THESE DE DOCTORAT DE
SORBONNE UNIVERSITE
préparée à EURECOM

École doctorale EDITE de Paris n° ED130
Spécialité: «Informatique, Télécommunications et Électronique»

Sujet de la thèse:

Reliability of Application-Generated Data for Security Evidence

Thèse présentée et soutenue à Biot, le 04/12/2025, par

Afiqah Azahari

Président	Prof. Tamara Rezk	INRIA
Rapporteur	Prof. Mohd Nazri Ismail	National Defence University of Malaysia
Examinateur	Dr. Merve Sahin	SAP
	Dr. Andrea Oliveri	EURECOM
Directeur de thèse	Prof. Davide Balzarotti	EURECOM

Preface

Acknowledgements

This thesis is dedicated to my parents, Mohammad Azahari bin Kamaruzaman and Zarina binti Kassim, for their love and Du'a. It is also dedicated to my sisters, Along and Alang, and my brothers, Adek and Baby, for their endless support. Finally, I thank my dear friends Nisa, Pipah, Eifa, and Hanan for their constant encouragement.

Abstract

Digital forensic investigators rely on data records generated by applications to understand what happened during security incidents. However, the reliability of such data for forensic purposes is not always guaranteed, particularly when it comes to application-generated data. For example, application logs often miss important details needed to explain security-related events because they are usually designed for debugging rather than forensic analysis. Mobile applications are also frequently updated, and each update can significantly changes the source of evidence from the application database. Such changes may resulted in critical evidence being lost for extraction or new introduce evidence that previous acquisition missed to capture. These factors raise an important question: to what extent can application-generated data be considered reliable for forensic purposes, and under what conditions does it miss newly introduced artifacts that could serve as evidence? To address these questions, this research examined application-generated data as a source of forensic evidence through three complementary studies. The first study explored how effectively application logs support security-focused analysis by assessing their usefulness for forensic tasks. The second study examined whether log statements are consistently placed along critical execution paths according to defined security logging practices. It also introduced an automated tool to detect missing or inconsistently placed logs that may affect the reliability of application log data. The third study investigates the reliability of forensic acquisition and evidential completeness under database schema changes across application versions, assessing how this changes affects the forensic data collection. These studies provide a systematic understanding of application-produced digital data, which often falls short as reliable security evidence, and present the challenges that can be addressed to improve its evidential value.

Résumé

Les enquêteurs en criminalistique numérique s'appuient sur les enregistrements de données générés par les applications pour comprendre ce qui s'est passé lors d'incidents de sécurité. Cependant, la fiabilité de ces données à des fins d'enquête n'est pas toujours garantie, en particulier lorsqu'il s'agit de données générées par des applications. Par exemple, les journaux d'application omettent souvent des détails importants nécessaires pour expliquer les événements liés à la sécurité, car ils sont généralement conçus pour le débogage plutôt que pour l'analyse criminalistique. Les applications mobiles sont également fréquemment mises à jour, et chaque mise à jour peut modifier considérablement la source des preuves provenant de la base de données de l'application. Ces modifications peuvent entraîner la perte de preuves essentielles pour l'extraction ou introduire de nouvelles preuves qui n'avaient pas été capturées lors de l'acquisition précédente. Ces facteurs soulèvent une question importante : dans quelle mesure les données générées par les applications peuvent-elles être considérées comme fiables à des fins d'investigation, et dans quelles conditions manquent-elles des artefacts nouvellement introduits qui pourraient servir de preuves ? Pour répondre à ces questions, cette recherche a examiné les données générées par les applications en tant que source de preuves judiciaires à travers trois études complémentaires. La première étude a exploré l'efficacité des journaux d'application dans le cadre d'analyses axées sur la sécurité en évaluant leur utilité pour les tâches judiciaires. La deuxième étude a examiné si les déclarations de journalisation sont systématiquement placées le long des chemins d'exécution critiques, conformément aux pratiques définies en matière de journalisation de sécurité. Elle a également présenté un outil automatisé permettant de détecter les journaux manquants ou placés de manière incohérente qui peuvent affecter la fiabilité des données des journaux d'application. La troisième étude examine la fiabilité de l'acquisition judiciaire et l'exhaustivité des preuves dans le cadre de modifications du schéma de base de données entre les différentes versions de l'application, en évaluant comment ces modifications affectent la collecte de données judiciaires. Ces études fournissent une compréhension systématique des données numériques produites par les applications, qui sont souvent insuffisantes en tant que preuves de sécurité fiables, et présentent les défis qui peuvent être relevés pour améliorer leur valeur probante.

Contents

1 Introduction **1**
- 1.0.1 Logs as Security-Relevant Evidence . . . 2
- 1.0.2 Mobile Application as Evidence . . . 3
- 1.1 Research Questions . . . 4
- 1.2 Contributions . . . 5

2 Related Works **9**
- 2.0.1 Application Logs: From Debugging to Security-Relevant Evidence . 9
- 2.0.2 Mobile Application: Database Evolution and Forensic Reliability . . 11

3 Defining and Assessing Security-Relevant Application Logs **13**
- 3.1 Application Logs for Forensic . . . 13
  - 3.1.1 Terminology . . . 14
  - 3.1.2 Standards and Guidelines . . . 14
  - 3.1.3 Forensic Use of Application Logs . . . 15
- 3.2 Methodology . . . 18
- 3.3 Inadequacy of Open-Source Application Logs for Digital Forensics . . . 26
  - 3.3.1 Application Logs for Timeline Development . . . 31
  - 3.3.2 Application Logs for Event Correlation . . . 31
  - 3.3.3 Application Logs for Execution Partitioning . . . 32
  - 3.3.4 Application Logs for Misuse Detection . . . 34
  - 3.3.5 Application Logs for Attack Detection . . . 35
  - 3.3.6 Assessment Overview . . . 36
- 3.4 Improving Practices Based on Observations . . . 37

4 Ensuring Security-Focused Log Placement and Consistency **39**
- 4.1 Background Scenario and Research Questions . . . 40
- 4.2 Methodology . . . 42
  - 4.2.1 Classification of Critical Calls . . . 43
  - 4.2.2 Log Placement . . . 43
  - 4.2.3 Summary . . . 45
- 4.3 Measurement of Logging Completeness and Behavior Across GTK Applications . . . 45
  - 4.3.1 Forensic Implications of Logging Inconsistencies . . . 49
- 4.4 Threat to Validity . . . 51

5 Reliability of Mobile Forensic Acquisition under Database Drift **53**
5.1 Motivation . . . 54
5.2 Methodology . . . 55
5.2.1 Environment Setup . . . 55
5.2.2 Applications Download (Step 1) . . . 56
5.2.3 Application Installation and Interaction (Step 2) . . . 57
5.2.4 Database Extraction (Step 3) . . . 57
5.2.5 Schema Drift Analysis (Step 4) . . . 57
5.2.6 Query Robustness Evaluation (Step 5) . . . 58
5.3 Results . . . 59
5.3.1 Update Frequency and Release Patterns . . . 59
5.3.2 Schema Drift . . . 60
5.4 Stability of Forensic Extraction Under Drift . . . 64
5.4.1 Cases Study: Query Update and Fallback . . . 65

6 Conclusion **69**
6.1 Summary of Findings . . . 69
6.2 Future Work . . . 70

Appendices **73**
.1 Appendices for Chapter 4 . . . 75

Résumé en français **93**
.1.1 Journaux comme preuves pertinentes pour la sécurité . . . 95
.1.2 Applications mobiles comme sources de preuves . . . 95
.2 Questions de recherche . . . 97
.3 Contributions . . . 98
.3.1 Journaux applicatifs : de l'outil de débogage à la preuve numérique pour la sécurité . . . 101
.3.2 Application mobile : évolution des bases de données et fiabilité forensique . . . 102

List of Figures

3.1 Usage of log level. 30

4.1 Pseudocode and data-flow graph of a motivation example. 40

5.1 LINE direct message acquisition query adaptation due to schema drift. . . 67
5.2 Telegram group messages query adaptations due to schema drift 68

1 The distribution of log percentages with user, network, and file data, as well as text-only events. 91

Chapter 1

Introduction

Digital evidence plays a crucial role in modern forensic investigations, serving as the foundation for uncovering events, actions, and intent within digital systems. Almost every interaction with digital systems leaves behind digital traces that can be analyzed to reconstruct events and attribute digital actions. These traces may present data of both the technical operations of a system and the actions of the users. While the data trace can serve as a valuable source of digital evidence, its evidential value largely depends on the platform where they are generated and how that platform records, stores, and preserves such data.

In desktop and server environments, for example, forensic investigations commonly rely on operating system artifacts and kernel-level events, such as system logs, file system metadata, and process records. In server environments, forensic analysis often includes examining application and network logs that record system and user activities during runtime. While these sources have been well explored in prior forensic research, application-level data on desktop systems has received far less attention.

In contrast, mobile forensics has evolved around application-level data. Because full physical acquisition of mobile devices is often challenging or restricted, forensic tools primarily focus on extracting data generated and stored by applications. This layer of data typically includes messages, call logs, media files, and cloud-synchronized records. As a result, application data have become the central focus of mobile forensic analysis, and understanding the completeness and reliability of evidence acquisition depends heavily on the consistency of this underlying application-level data.

Applications serve as the primary interface between users and systems, continuously generating digital traces that describe both user behavior and internal system processes. Yet, what data applications record, where they store it, and how long they retain it depend on design choices made during development and maintenance. These decisions directly shape the availability, consistency, and completeness of potential forensic evidence, influencing how effectively investigators can reconstruct activities or detect incidents over time. For example, a log entry critical for tracing an intrusion within an application may never be created, may be recorded only under specific conditions, or may later be altered or deleted. When such log data, particularly desktop application logs, that still remain understudied, the availability, consistency, and completeness of its data for forensic tasks become even more uncertain. Moreover, application databases that contain key identifiers may be restructured or removed as part of feature updates, security enhancements, or privacy-related revisions. This is particularly evident in mobile applications,

where database structures often evolve as developers add, modify, or remove data fields. Since mobile forensics relies heavily on application-level data, such changes can directly influence the consistency and reliability of forensic analysis.

This reflects the unpredictable nature of data produced by applications, where evidence-related information may appear inconsistently in logs, change location across database tables, or even be completely removed from the application's database. Such variability raises fundamental uncertainties about whether the digital evidence produced by an application can remain adequate and forensically reliable over time.

This thesis addresses these uncertainties through the lens of the reliability of application-produced data evidence. Here, reliability refers to an application's ability to provide digital data that remains stable and forensically sound despite software updates or uncertain conditions (such as logs intended only for debugging). This reliability is assessed through three critical factors that directly reflect those uncertainties:
(1) the presence (availability) and conditions of application logs, which determine whether security-relevant events are recorded and contain sufficient detail for a forensic task;
(2) the consistency of log coverage around execution of critical actions, defining the best placement of logs in the application's source code for security-relevant log data; and
(3) the persistence and structural stability of database-stored evidence, which determines whether forensic queries can continue (completeness) to retrieve evidence data as the application evolves.
Together, these factors define the core reliability of application-produced data for security and capture the key sources of variability that affect it, thus forming the foundation for the research questions explored in this thesis.

The remainder of the thesis turns these two particularly important sources of such evidence, logs and databases, further explaining the ways data evidence can present and preserve security-relevant data and to evaluate and strengthen the reliability of application in producing the data for forensic investigations.

1.0.1 Logs as Security-Relevant Evidence

Logs provide a chronological view of how software and services behave, capturing internal and external events that reveal a system's operation. Although introduced mainly for debugging and performance monitoring, logging has become equally critical for security because the same event records can expose attacks, policy violations, and abnormal behaviors. These records supply the contextual data needed to detect threats, prevent compromise, and reconstruct malicious activity. Records from operating-system logs describe kernel activities and process creation, network logs expose communication patterns and anomalous traffic, and application logs document fine-grained user actions and application-specific state changes. When these diverse data sources are correlated, investigators can use them as "digital eyewitnesses" to reconstruct the who, what, when, where, why, and how of digital events. For instance, linking a suspicious login at the operating-system level to subsequent data exfiltration through an application's network calls. For such log data to serve as reliable digital evidence, their integrity, provenance, and proper collection and preservation must be assured to ensure legal admissibility and effective incident reconstruction [Ken04]. Established practices such as cryptographic hashing and digital signatures, secure time-stamping, centralized log aggregation over encrypted channels, and append-only storage, together with international guidelines like

NIST SP 800-92 and ISO/IEC 27037, provide technical and procedural foundations for collecting and preserving logs in a legally defensible manner. Equally critical to their forensic value are the availability and consistency of these logs. Availability means that significant security events are recorded and remain accessible for analysis, while consistency ensures that these records are complete and dependable over time. Any absence or inconsistency directly undermines the ability of logs to serve as reliable digital evidence. This challenge is especially acute at the application layer. Application logging is frequently ad hoc, with developers focusing on debugging and performance rather than security. As a result, security-sensitive operations may remain partially or entirely unlogged, particularly during application failures or security attacks. Therefore, understanding and improving the adequacy of application logs data for security-relevant evidence is central to this thesis.

1.0.2 Mobile Application as Evidence

A study of 12,763 appeal judgments in the UK (2006–2011) showed that prosecutions involving mobile digital evidence increased from 51 to 157 cases [MGB13]. Similarly, research in China found that 3.3% of 66,552 criminal cases (2013–2018) involved mobile phone evidence, with call records dominating and WeChat data growing from 1% to 25%, especially in drug-related and terrorism cases [ZBMN22]. These findings highlight how mobile devices have become critical sources of digital evidence, offering data such as call logs, text messages, chat histories, location traces, and media files. Mobile artifacts can be acquired at multiple levels, from user-accessible file systems and application-private storage to bit-level extractions to uncover deleted files, login data, or passwords. A variety of methods and tools, both commercial and open-source, support mobile data forensic acquisition, which enable investigators to recover diverse artifacts across different acquisition levels, such as physical, file-system, and logical extraction. Physical extraction creates a full bit-for-bit image of the device, file-system extraction retrieves the complete directory structure and files, and logical extraction acquires active application and user data through system interfaces. Logical acquisition [ABJ14], which involves extracting files and databases through the operating system's standard interfaces without modifying the device, has become a fundamental technique for retrieving an application's internal storage. Because physical extraction is complex and often restricted, most research has focused on logical acquisition. With the operating system's native support for sandboxed storage, mobile applications now manage their data within isolated directories. Over time, this data has typically been stored in structured databases such as SQLite. These databases capture interactions between the user and the application, including messages, contacts, call and location histories, media records, authentication tokens, and other activity logs. The results of a logical acquisition can serve as key forensic evidence, particularly when the application stores its data in SQLite databases. However, evidence stored in these databases presents significant challenges for forensic acquisition, especially when application updates modify how data is structured or stored. Developers may change what is recorded in the databases, including which tables exist, what columns are included, and how long data is retained. These modifications mean that the results of forensic acquisition can vary each time a new version is released. A clear understanding of these internal storage processes and their evolution is therefore essential for reliable forensic acquisition. The completeness of extracted evidence is a decisive factor in ensuring overall reliability.

This thesis addresses this issue by examining how changes in application database schemas affect the completeness and repeatability of forensic extractions in mobile applications.

The value of that evidence as a digital eyewitness ultimately depends on the availability, consistency, and completeness of the application-produced data. In the case of application logs, their ad-hoc placement and the absence of clear guidelines for security logging raise doubts about whether such logs are adequately prepared for forensic use, and whether their placement truly reflects the best locations for capturing security-relevant events. To date, no prior work has defined what constitutes logs specifically for security purposes, nor established criteria for where such logs are used for security events. Furthermore, another factor that can compromise the reliability of evidence is the evolution of the underlying platform. Frequent application updates can change data structures of where evidence has been placed (and also present new evidence), potentially disrupting established forensic acquisition flows and put the completeness and reliability of digital evidence into a risk.

1.1 Research Questions

Applications generate digital data that can serve as a source of evidence for forensic. Among the most important sources are logs, which record backtraces to explain what, how, and why events occurred, and databases, which store information about users and application activity. For these sources to provide forensic value, the evidence they contain must remain available, consistent, and complete over time. Yet in practice, this stability is difficult to guarantee: logs are commonly designed for debugging or error detection rather than forensics, and application databases often change structure as updates introduce new data or remove existing data evidence. Such variability introduces fundamental uncertainty in whether application-generated data can reliably support forensic reconstruction and security investigations. These uncertainties create critical gaps in defining and assuring application-produced data as a dependable and consistent source of digital evidence. To address these gaps, this thesis aims to answer the following three research questions:

Question #1: What constitutes forensic-relevant application logs, and to what extent are current logging practices prepared to support security and forensic investigations? This question examines the reliability and uncertainty of application logs, focusing on the first critical factor: the presence and conditions of logs. It investigates what makes application logs forensically relevant and evaluates whether their presence and their condition meet the requirements for a forensic task. Building on prior high-level discussions such as [SD24], this study identifies and formalizes the essential requirements for what must be logged and how log entries should be structured to ensure their forensic value. It then examines to what extent application developers meet these conditions in practice, particularly in desktop environments where application-level records have received limited attention. This further assesses whether current desktop application logging practices are adequate to support security and forensic investigations task.

Question #2: How the presence and placement of log statements be defined to ensure their log coverage for security-relevant forensic investigations? This question focuses on the consistency of log coverage around critical actions, ensuring that

every significant operation is captured across all possible execution paths for complete and consistency of data for evidence. Building on prior work that largely targets log placement for debugging [YPH+12a, ZHF+15a, FZH+14a, ZRL+17b], this question shifts the prior study focus on security, investigating and presenting the placement and consistency of logs along critical path for security-event investigation. Plus, in exploring RQ2, it is important to consider that poor design choices in log placement (described in the literature as log smells [SSB24]) can create gaps or inconsistencies that compromise the completeness of log data for security-relevant evidence. RQ2 aims to identify missing or inconsistent log coverage and assess how well real-world applications meet the requirements for reliable, security-focused application logging.

Question #3: To what extent does forensic acquisition remain complete and accurate as application databases evolve across versions? Mobile applications are updated frequently to patch vulnerabilities, introduce new features, and improve privacy. These updates often change the structure of internal databases by adding, removing, or modifying tables and columns, which can shift where evidence is stored and make established forensic queries incomplete or invalid. This question focuses on the persistence and structural stability of application database-stored evidence, evaluating whether forensic queries consistently retrieve all required evidence data as applications add, remove, or restructure the data evidence. It aims to measure how database changes affect the completeness of evidence. It also aims to determine the conditions under which forensic acquisition remains reliable across application updates.

1.2 Contributions

This thesis makes several key contributions to the field, as outlined below.

Contribution I (Chapter 3)

The first contribution of this thesis is a systematic investigation on the role of application logs in supporting forensic analysis. This directly addresses **Research Question 1 (RQ1)**, which asks *what constitutes forensic-relevant application logs and to what extent current logging practices support security and forensic investigations*. This study begins by outlining the key tasks that forensic investigations typically require, including timeline reconstruction, event correlation, execution partitioning, misuse detection, and attack detection. Building on this, the study identifies the specific types of information that logs must contain to support these tasks effectively. These include timestamps, user actions, event identifiers, and warnings related to unusual or malformed data. To assess how well these logging requirements are met in practice, a detailed analysis of 60 open-source desktop applications was conducted. As part of this evaluation, the existing log implemented within each application were collected and examined to determine whether they included the parameters necessary to support forensic tasks. The results reveal widespread limitations in current logging practices for security purposes, with many applications failing to consistently capture elements such as timestamps, user actions, and error conditions. By formalising the requirements for forensic-relevant logs and empirically quantifying current practice, this contribution provides a concrete answer to RQ1 and establishes a foundation for designing security-aware application logging systems.

This study and its findings are published in:

- Azahari Afiqah and Davide Balzarotti
  "On the Inadequacy of Open-Source Application Logs for Digital Forensics."
  Forensic Science International: Digital Investigation, Volume 49, June 2024. [AB24]

Contribution II (Chapter 4)

The second contribution of this thesis is the analysis of log statement placement in relation to critical operations. This contribution responds to **Research Question 2 (RQ2)**, which examines *how the presence and placement of log statements be defined to ensure their log coverage for security-relevant forensic investigations.* A consistent logging strategy is essential around operations that are security-sensitive or prone to failure, as these may compromise system integrity, cause unexpected crashes, or create potential entry points for attacks. This study examines whether log messages are issued before or after critical actions, and whether conditional branches may prevent their execution. By reconstructing full execution paths, it emphasizes the importance of missing logs along every path from user input to the critical call and beyond. Missing logs at any point reduce their forensic value, as they fail to capture what occurred during normal execution, or worst during crashes, or security incidents. To perform this analysis at scale, a tool named BlindSpot was developed. Built on top of Joern, BlindSpot combines data-flow analysis to trace untrusted input, control-flow traversal to reconstruct interprocedural execution paths, and dominance checks to evaluate whether log statements are executed reliably. This makes it possible to automatically determine whether logs provide reliable coverage of critical operations throughout the application's execution. BlindSpot was applied to ten open-source GTK-based desktop applications. The results revealed widespread inconsistencies: operations influenced by external input were often left unlogged or were only logged under specific conditions. By defining the requirements for proper log presence and placement and identifying another criterion of log smells in terms of presence and placement with security impact, this contribution establishes a framework for assessing security-relevant logging. It then demonstrates these gaps in real-world applications, providing findings for RQ2.

This chapter and its finding is under revision and re-review at *Digital Threats: Research and Practice* journal.

Contribution III (Chapter 5)

The third contribution of this thesis focuses on the impact of database evolution on mobile forensics, particularly on how the availability of evidence in Android application databases when version evolved. This contribution directly addresses **Research Question 3 (RQ3)**, which asks *to what extent does forensic acquisition remain complete and accurate as application databases evolve across versions?*. To address this question, a longitudinal analysis was carried out on 20 popular Android applications covering 320 APK versions. This provides one of the first systematic views of how application updates alter database structures and how such changes affect the process of forensic acquisition.

The study introduces a classification of schema drift, grouping applications into high-, moderate-, and low-drift categories. Communication and social media apps were found to have the highest levels of drift, often introducing changes that disrupted existing forensic methods. Even small adjustments to the schema were enough to break queries, change the return results, or reduce the amount of evidence that could be collected. Detailed case studies further show that database changes can weaken established forensic workflows but also add new sources of evidence that did not exist in earlier versions. These effects highlights both the risks and opportunities that come with continuous application updates. This contribution strengthens the field by providing a longitudinal perspective on schema drift in mobile applications and by outlining its consequences for forensic investigations. By quantifying schema drift and demonstrating its impact on the reliability of evidence extraction, this contribution provides a direct response to RQ3.

The paper entitled **Resilience of Forensic Evidence Acquisition Under Database Schema Drift** has been accepted for presentation at DFRWS Europe 2026 and for publication in *Forensic Science International: Digital Investigation*. It is currently in the production phase.

Chapter 2

Related Works

Significant research over the past decade has focused on extracting and analyzing the digital traces that applications leave behind to support forensic investigation. These traces are can be found in two places: the application logs that record system and user events, and the internal databases that store persistent application data. Both have been studied in depth, yet in different ways and with different priorities. Prior work on logs has addressed improving the quality and placement of log statements for debugging and performance monitoring, and more recently for security compliance and attack detection. Work on mobile applications, meanwhile, has documented how apps evolve quickly across versions and how this affects their functionality, permissions, and vulnerabilities. Despite these advances, important gaps remain: what exactly should count as security-relevant log content and where such logs must be placed are still not well defined, and the long-term impact of database schema drift on the completeness and reliability of forensic evidence has not yet received attention.

2.0.1 Application Logs: From Debugging to Security-Relevant Evidence

Logs have served as a foundational source for various types of analysis, enabling tasks such as event reconstruction, anomaly detection, and incident investigation. These analyses can be derived from a wide range of sources, including system event logs data, IoT device logs data, and network audit logs data. A significant body of literature has highlighted the importance and utility of this data. For instance, surveys by He et al. [HHC+20] present works on automated techniques on log data for anomaly detection, failure prediction, and system diagnosis. Additionally, the survey by Studiawan et al. [SSP19] presents a comprehensive review of how event logs have been utilized to support forensic investigations. Collectively, these two works underscore the significance of log data as a foundation for extracting meaningful information and facilitating in-depth analysis. Also, the work by [Kar21] introduces the term 'forensic log records' that refer as log entries that are relevant to incident analysis. This implies the role of logs as a crucial evidence source for capturing actions, supporting event reconstruction, and providing key insights for security and forensic purposes. Several studies have explored logging practices from a software development perspective, focusing on what information should be logged, which variables are most relevant, and where to place logging statements within the code. The goal is to improve the quality of logs to better support tasks such as system analysis and

debugging. In addressing the question of what to log, Li et al. [LHZ+24] introduced SCLogger, a contextualized approach to automatic logging statement generation. Their method leverages inter-method static analysis and language models to identify relevant variables for logging. In addition, to assist developers in determining which variables to log, Liu et al. [LXL+21] proposed a technique that identifies the most important variables to log. Complementing this, Li et al. [LSH18] employed ordinal regression models to automatically recommend the most appropriate log level for each newly-added logging statement. Concerning where to log, foundational work by Fu et al. [FZH+14b] presented an empirical study of industrial logging practices. Building on this, Zhao et al. [ZRL+17a] applied information theory to learn optimal log placement by identifying program points that yield the most insight into runtime behavior. Besides, Zhu et al. [ZHF+15b] introduced LogAdvisor, a logging suggestion tool that automatically learns common log placement practices from existing code and provides actionable suggestions to developers. Further efforts by Zhao et al. [ZRL+17c] have focused on placing logs only in the most informative locations to help developers understand failures with minimal performance overhead. In addition, Saarimäki et al. [SSB24] introduced the concept of log smells, a taxonomy of poor design choices in logging implementation and log statements. Their work emphasis on quality issues that can reduce logs reliability. Most existing studies here take a developer-centric view, aiming to improve logs for debugging, performance monitoring, and understanding system behavior. However, they often overlook the design and implementation of logs specifically to support security and forensic analysis. Several studies propose systematic methods to improve log design with a focus on security monitoring and forensic readiness. Shen et al. [SSZ23] perform static analysis to identify missing access-deny logs, emphasizing the importance of capturing failure cases for security auditing. Rivera-Ortiz and Pasquale [ROP19] present a proactive approach for forensic readiness by defining logging requirements early in system design, and later extend their work [ROP20] to automate log placement based on attack scenarios. Olegård et al. [OAL25] introduce a method to trace activity by recording causal links between events, enabling sufficient log information to trace attacker movement across multiple system components. These studies prioritize the placement and intent of logs to meet forensic goals; however, they often stop short of defining what concrete data each log entry must contain for security. On top of this, some studies have focused on implementing logging practices that align with existing compliance requirements or standards. King and Williams [KW13] conducted foundational work in the healthcare domain by cataloging critical log types in Electronic Health Record (EHR) systems. They identify essential categories such as data transactions and security events, revealing the need for systematic log coverage. Follow-up studies by King et al. [KPW15, KSRW17] propose heuristic methods, such as verb-object pairing, to suggest what should be logged, and contrast resource-driven and standard-driven logging strategies to assist analysts. On top of that, Sahin and Daniele [SD24] analyzed Java-based web applications and uncovered significant gaps in logging practices when benchmarked against security guidelines from compliance frameworks and governmental recommendations. While previous studies have identified gaps and proposed improvements in logging practices for security, their work is often guided by compliance standards or government-issued guidelines. What remains largely unaddressed is a clear definition of log content (such as variables, inputs, or contextual data) that is valuable for forensic investigation, as well as systematic guidance on where logs should be placed to ensure forensic completeness and consistency for secu-

rity. This thesis aims to address these critical gaps by proposing a forensic-driven model of logging. It defines what constitutes forensic-relevant log content, establishes guidelines for log placement in source code to capture critical actions, therefore improve the completeness of log coverage for security.

2.0.2 Mobile Application: Database Evolution and Forensic Reliability

Mobile application forensics has been widely studied from different angles, with researchers examining how evidence can be extracted, how applications evolve over time, and how these changes affect the reliability of forensic tools. Numerous research works has focused on the development of different techniques to extract forensics evidence from mobile applications databases and its recoverability in function of changes in databases' layout. An initial work in this direction was performed in 2015 by Sgaras et al. [SKLK15] that have shown how to practically acquire forensic evidence from popular communication apps like WhatsApp and Viber, but without studying structural changes over time. At the same time, several open-source tools has been proposed to extract evidences from applications such as ALEAPP [Bri23] works with plugins to parse evidence from Android file-system dumps or archives and generates reports on user and application activity, Chatistics [SkS23] that parses exported chat logs, and Whapa [B1623] that decrypts and parsed evidence from WhatsApp databases. A first work that try to address the problem of apps' database schema evolution has been proposed by Shimmi et al. [SDKA20]. In their study, they analyzed schema evolution in iOS backups, proposing automated comparison of table and column structures to keep updated with where the evidence can be extracted. While they showed that schema drift disrupts query reliability, their scope was limited to native iOS applications. Another interisting methodology was developed by Lee et al. [LCL23] that presented a predictive approach to identify schema layouts in messaging apps, without however, study how that layout change in different application versions. Recently, AnForA [ACG20] and Argus [BDJH25] have been proposed to automate detection of app-generated artifacts through snapshotting and scripted app interactions. Two of these tools, internally use SQLDiff to highlight database changes but did not evaluate how schema changes impacts forensic evidence extraction.

Beyond forensics, researchers has examined how Android applications evolve from different point of view. Prior works investigated code complexity and maintainability [GLBK19], accessibility features [DSODJ+23, OdSAE24], and topic modeling of release changes [TAHB14]. Hecht et al. [HBR+15] highlighted how poor design choices in updates can degrade performance and quality. Security- and privacy-focused works have shown that permission requests often evolve dangerously [CG17, TM17], while large-scale analyses of millions of app records have revealed persistent issues such as third-party tracking and malicious behaviors [WLG19]. Vulnerability research further shown that insecure dependencies can persist across multiple updates [GLK+19]. While these studies collectively highlight the rapid evolution of mobile applications, they focus primarily on code, permissions, or functionality. To date, no empirical work has examined application version evolution through the lens of database schema drift and its direct impact on the reliability of forensic extraction and acquisition.
This thesis builds on these prior works and addresses the research gaps they reveal. Although logs and mobile databases differ in structure and purpose, both provide digital

traces that can serve as valuable security evidence. For application logs, however, the definition of security-relevant content remains underexplored, leaving the adequacy of such logs for forensic use remain unclear. On top of that, the presence and placement of log statements that can be defined for a forensic task remains an open challenge. In terms of log placement, existing studies (e.g., [FZH+14b, ZRL+17a, ZHF+15b, ZRL+17c]) focus mainly on improving debugging and performance monitoring. While this definition is important, prior work does not establish security-driven criteria for where logs should be positioned or how their presence should be guaranteed. This is important for security events, where the availability, consistency and completeness of evidence data, in this case application logs must be carefully considered. In mobile application domain, prior studies show that applications evolve rapidly; however, little is known about how the underlying database schemas that store forensic evidence change over time and how such changes affect the reliability and completeness of the evidence. This gap leaves open critical questions about whether forensic tools can consistently capture all relevant data as applications and their internal structures evolve.

Chapter 3

Defining and Assessing Security-Relevant Application Logs

Logs provide post-action data, capturing the trail of activities that occur within an application. The ability to backtrace events is essential and has long been recognized as valuable for system monitoring, software development, and troubleshooting. Prior studies have shown their importance for performance analysis and failure diagnosis [LCHX23, LLC+22, LLC+23, HLS+23], and other work has focused on refining logs for specific security goals such as intrusion or attack detection [KW13, KPW15, KSRW17, ROP19, ROP20]. Despite this progress, the readiness of logs for broader security use, especially for digital forensics, has not been systematically examined. Understanding what makes logs ready for security is an important first step toward ensuring that logs can support investigations when needed. This chapter addresses the uncertainties surrounding the use of application data for security by focusing on the first key factor: the presence and conditions of application logs, which determine whether security-relevant events are recorded and contain sufficient detail for forensic tasks. This chapter begins by defining the fundamental forensic requirements that application logs should meet, offering a practical reference for developers and compliance teams to understand what constitutes forensic-ready logging. It then presents a methodology for evaluating these requirements through an empirical analysis of log statements collected from 60 open-source applications. Finally, it discusses the reasons why existing logs often fall short of supporting security and forensic needs. By assessing both the quality and coverage of application logs, this chapter establishes a foundation for improving the forensic readiness of logging in modern software systems.

3.1 Application Logs for Forensic

The term *log* originally referred to the process of keeping a record of the progress of a ship, an operation that was performed by using a wooden chip log attached to a knotted rope [Whe04]. In the context of computing and information technology, the term is used to describe the recording of events that capture the usage, the activity, and the operation of a computer system.

3.1.1 Terminology

Logs are routinely used to record information about the runtime operations of operating systems, servers, applications, and network devices. This data is usually broken down in a sequence of individual *entries*, each reporting information on a particular event that occurred within the source that originates the log. For example, operating system logs capture a range of events from startup messages and system modifications to unexpected shutdowns. In contrast, application logs provide a comprehensive overview of activities, which encompass a broader scope of actions and user interactions within the application. These logs might include detailed records such as authorization processes, descriptions of actions taken, identifications of users involved, explanations for any failures, and records of activities between the application and its users, as well as insights into the internal dynamics of the application. Logs can be stored in various formats and locations, such as files, databases, or cloud-based storage systems, and serve as a primary source of information that is essential for detecting and troubleshooting problems but also for verifying the operational performance and efficiency of a system. Furthermore, logs also play a crucial role for security, both as a way to **detect** attacks or compromise and as a way to **investigate** them in a post-mortem scenario.

3.1.2 Standards and Guidelines

Various standards and guidelines exist to guide the developers and organizations to design and streamline their logging process. For instance, ISO 27002 provides essential logging requirements, such as the fact that audit logs should be generated and stored for a certain period of time to assist future analysis. At the same time, ISO provides very few details on how such requirements needs to be implemented. The Control Objectives for Information and Related Technologies (COBIT) framework offers additional information about logging management, including logging configuration, changes, and backup processes. Nevertheless, COBIT provides little details regarding the requirement of logs for security purposes. Another industry standard that specifies the requirement to log is PCI. In fact, PCI v3.2.1 provides much more detailed logging guidance that any relevant applications could apply. These include logs' details and properties, the type of events to logs, the importance of time synchronization, important security requirement, logs locations, log retention, and the need for a routine review [PCI18]. While the PCI standard provides a complex methodology, it is specifically designed for software applications, which require different logging provisions and practices. Additionally, many technical blog posts provide logging recommendations [Rob, Bes, Loga, Logc, Con, Logb, Dev, OWA]. Industry players, support engineers, the development community, and security experts often discuss logging and share their opinions and perspectives on proper logging techniques. These posts provide tips, including what to log, what 'not' to log, which verbosity level and logging format to use, and which log retention approach to adopt. One major limitation is that the vast majority of log standards and guidelines focus their requirements and recommendations on logs for debugging rather than for security. In 2010 Chuvakin and Peterson [CP10] compared the two classes (debugging vs security) and discussed how the second should be intended specifically for security and audit personnel. Security-relevant logs should always be available and contain messages reporting information about attacks, activities, and faults. The authors also defined what to log and the retention of log

files. Moreover, Brouwer and Mertens [BM15] listed five categories of logging requirement that are relevant for forensic investigation. These requirements include logs retention, correlation, content, integrity and consideration. The authors mention that logs for security should contain information on *"who did what, where, when, how, and the results"*. However, these suggestions and recommendations were based on the authors personal experience, without any justification or experiment to support the discussions. Finally, it is important to mention that the Cyber-Investigation Analysis Standard Expression (CASE) introduces a significant effort to standardize the representation of forensic data, thus simplifying the investigation and analysis process through enhanced provenance and traceability [CBG+17]. CASE considerably improves the existing standards and guidelines for logging in security and forensic contexts. By breaking down data silos and enhancing tool interoperability, CASE supports the automated normalization, combination, and correlation of data from diverse sources. Its community-driven API eases integration into various tools, allowing developers to tailor CASE to fit their specific data structures and formats [CBG+18, CNH19]. This flexibility in mapping and converting data to comply with the CASE standard makes it a valuable asset in advancing digital forensic investigations and facilitating collaboration among cyber-investigators. However, integrating CASE without properly understanding the essential information required for forensic logging may lead to inadequate evidence collection. Additionally, the use of CASE to merge multiple data sources, especially those lacking comprehensive data, might backfire and actually impede effective evidence extraction. Thus, while CASE presents a comprehensive framework for integrating diverse data sources, the accuracy and completeness of the input data remain critical for effective forensic evidence gathering and analysis.

3.1.3 Forensic Use of Application Logs

Application logs can provide a broad range of information that can be used as part of an incident response or forensic investigations. However, both the number and the type of tasks that an analyst can perform depend on the available data.

In this section, five main categories are defined to organize common forensic analysis tasks:

1. Activity Timelines.
   Timelines play a very important role in digital forensics, as they allow an investigator to easily aggregate and navigate the sequence of events that took place on a computer system. For instance, a timeline can be used to identify the cause of an incident by locating co-occurring events (such as an external drive plugged in just before a document was accessed) or by identifying patterns of sequential behaviors. The ability to restrict the analysis to particular time ranges, to pinpoint relevant entries, and to put them in context with respect to all other pieces of evidence collected in the target system makes timelines an invaluable tool. In order to be integrated into a timeline, log entries must include a well-identified timestamp. Their absence make it difficult or impossible to synchronize the logs with other data sources and correlate events. In addition, relevant events that capture user behavior need to be logged by the application (these events vary depending on the type of application and will be discussed in more detail in Section 3.2).

2. Events Correlation.
   The ability to correlate events from different sources is paramount in a forensic investigation. For instance, a broad range of information can be extracted from emails, server logs, network appliances, antivirus products, and operating systems. All these sources can be useful in isolation, but the ability to 'connect the dots' by linking together events across the entire target system, or across different systems, can help an investigator to better understand what is relevant to the case and what parts of the system have been compromised. For instance, a succesful password bruteforce against a dormant employee account can be difficult to investigate if the application does not log the username associated to each attempt, or the network address from which the connections originated. The representation of unique characteristics in log message plays a crucial role in event correlation. In particular, the presence of unique identifiers (such as URLs, user names and IDs, network addresses, domain names, and complete file and directory paths) helps to reconstruct complex timelines of events and to transform raw logs into coherent event sequences, such as workflows. For instance, a recent study by Li et al. [LLC+23] leverages commonly available IDs within logs to tackle challenges associated with intermixed event sequences. Moreover, unique events in logs enrich the analysis by tracing activity sequences within the application, providing additional context. Each unique event acts as a distinct marker or footprint, allowing the tracing of specific actions or occurrences. The unique values provided by dynamic variables in logs aid in differentiating between normal and anomalous system behavior. Developers have observed that various categories of these dynamic variables capture crucial information, enhancing log analysis. Thus, these unique identifiers in logs can be harnessed to correlate events and identify system components that may require in-depth scrutiny.

3. Execution Partitioning.
   Audit logs are among the most precise and fine-grained source of information available to an investigator. When enabled, in combination with execution and application logs, they allow tracking of each system event's complex dependencies using *data provenance graphs* [HNDB20]. However, the presence of long-running processes often undermine the effectiveness of the analysis by introducing the provenance graph a large numbers of irrelevant or erroneous causal dependencies. For instance, a new file created by a browser would depend on all URLs that were ever opened by the same process in the past. This dependency explosion is a well-known problem in digital forensics, and over the years researchers have proposed several mitigations based on the idea of *execution unit partitioning* [MZW+17, HGL+19, HSS20]. In a nutshell, partitioning consists in breaking down audit log events in small segments, based on the presence of particular application events. For instance, whenever an application closes a document and opens a new one, causal dependencies from the content of the former document should not be propagated to the latter.

   To support partitioning, a user application needs to generate a recognizable log entry in correspondence to a unit boundary (e.g., when serving a new user requests or processing a new document). This idea was recently described by Yu et al. [YMZ+21] who in 2021 analyzed 32 Linux applications and found that 1) the vast majority were long-running and 2) that all of them had log entries that could be used for the

purpose of execution partitioning.

4. Misuses detection.
   Misuse detection refers to the ability to use application logs to identify unauthorized behaviors, typically associated with insider threats or attackers who have already gained access to the system. For instance, a legitimate user can try to misuse an application to perform lateral movements inside the network, to impersonate another user, or to gain access to sensitive information. In order to support misuse detection an application needs to log any operation that affect the state of the system (such as change of privileges or service configurations). In addition logs should report operations on sensitive data as well as any action which failed or which was denied, based on insufficient user permissions. A typical example is provided by the `sudo` setuid application, which logs any failed attempt to execute privileged commands. This information can serve as a way to support future investigations, but also as an effective way to deter users against poking around and trying to guess passwords.

5. Attack detection.
   After an attack, it is crucial for an investigator to be able to retrieve the attack vector, the source of the attack, and all actions that the attacker performed on the system. Security logs from both network and host intrusion detection systems are designed specifically for this purpose but they often lack the necessary visibility into each application state. For instance, they may be able to detect a malicious file created in the `/tmp/` directory but they might fail to provide information on which process created the file and, more importantly, which malicious input caused the process to misbehave. In the lack of audit logs, application logs could help to cover this blind spot by providing information about any unusual, suspicious, or malformed piece of data the application encounters. For instance, exceptions due to unexpected behaviors or triggered by malformed data and input that fails validation routines should be reported in the logs, together with enough information to identify the source of the data and/or its correspondent user request.

The aforementioned five categories cover many different uses of application logs for forensics and incident response. Each category introduces specific requirements (e.g., the presence of timestamps and unique object identifiers) and guidelines for developers (e.g., the need to log exceptions and malformed or invalid inputs). An orthogonal dimension, which crosses all possible use cases, is related to the availability and integrity of log files. In practice, application logs are often written to multiple locations, including remote servers, consoles, files, and databases. Some applications print relevant error information only to the standard error, making them difficult to collect and store for later use. Log availability also need to consider other log properties such as log rotation, log archival, log retention period, and security properties such as access privileges or the use of append-only storage. Centralized log daemons simplify this procedure but they are typically adopted only to manage system services and servers, leaving other userspace applications to their own custom log management. Creating a comprehensive list of user actions that an application should log to support security and forensic investigations is very difficult.

Different applications serve different purposes, thus with different requirements, features, and development methodologies. For instance, Office Suites differ from a communication chats or file sharing applications. To this extent, different log frameworks, formats, and default configurations are chosen by application developers. In general, logs are created primarily for debugging purposes, and it remains unclear to what extent they also fulfill the requirements of logs for forensic analysis tasks as described above. This question will be addressed in the next section.

3.2 Methodology

The following presents the selected applications and the methods used to evaluate the usefulness of their logs for common forensic analysis tasks.

To conduct the study, a set of 60 open-source applications was selected across various categories, including file managers, chat and communication tools, office software, file sharing applications, screen capture programs, antivirus software, text editors, remote desktop applications, window managers, and application launchers. The aim was to ensure broad coverage of applications that may process untrusted data or interact with a system in ways potentially relevant to forensic investigations. Open-source applications were chosen to allow access to source code, thereby enabling a thorough examination of the nature and content of individual log statements.

Table 3.1: List of analyzed applications

Application	Description
File Managers	
Nautilus	Default file manager of the GNOME desktop
Disks	Default storage management of the GNOME desktop
Dolphin	KDE's file manager
nnn	Terminal file manager
recoll	Full-text search tool for Unix/Linux
Konqueror	File management and preview based on web browsing
Thunar	File manager for Linux and other Unix-like systems
Chat and Communication	
IceChat (Windows)	IRC client
BeeBEEP	Peer-to-peer office messenger
Pidgin	Multi-platform instant messaging client
signal-desktop	A private messenger application
Mattermost-Desktop	Platform for secure collaboration
Konversation	User-friendly and fully featured IRC client
jitsi	VoIP, video conferencing, and instant messaging
wire-desktop	Encrypted communication and collaboration app
session-desktop	Onion routing based messenger
HexChat	IRC client
Terminal-Irssi	Terminal based IRC client
tox	Peer-to-peer instant messaging and video-calling app
Office Tools	
Calligra	Office and graphic art suite by KDE
Xournal++	Handwriting note-taking software with PDF annotation support
SumatraPDF	PDF viewer

Continued on next page

Table 3.1 — *continued from previous page*

Application	Description
zael	Offline documentation browser
Okular	KDE document viewer
xpdf	PDF viewer and toolkit
Apache Open Office	Office productivity software suite
Scribus	Libre desktop publishing
Document Viewer	PDF viewer for GNOME
Torrent and File Sharing Platforms	
qBittorrent	BitTorrent client
Deluge	Cross-platform BitTorrent client
transmissiongtk	Transmission BitTorrent client repository
frostwire	BitTorrent client
warpinator	File sharing across the LAN
Screen Capture Tools	
ShareX	Capture or record any area of the screen
OBS-Studio	Live streaming and screen recording
Greenshot (Windows)	Screenshot manager
Security and Antivirus	
Seahorse	Password and encryption key manager for GNOME
hashcat	Password recovery utility
Clam AntiVirus (Linux)	Open-source antivirus
Text Editors	
Zettlr	Markdown editor
gnome-text-editor	Default editor of GNOME
Vim	Highly configurable text editor
jrnl	Simple journal application for the command line
Utilities	
Console	GNOME's default terminal emulator
Cheese	GNOME's webcam application
Guvcview	Webcam application
XDM	Download manager
Media Players	
VLC Media Player	Video player
musikcube	Terminal based music player
Kodi	Media player and entertainment hub
Window Managers	
IceWM	Open window manager
tmux	Terminal multiplexer
bspwm	Tiling window manager
Qtile	Hackable tiling window manager
Remote Desktop Tools	
Remotely	A remote control and remote scripting solution
xrdp	Open-source RDP server
App Launchers	
albert	A fast and flexible keyboard launcher
launchy	Linux launcher
ulauncher	Linux launcher
Wox	Launcher for Windows

Table 3.1 presents an overview of the study dataset. The applications that are consid-

ered developed in various programming languages, including Java, Python, C, C++, C#, and Typescript. Although some applications may contain components written in multiple languages (e.g., in the case of plugins or extensions), the analysis focused on the primary language used for the development of the core application.

The latest source code of each application was selected, and the SLOCCOUNT tool[1] was used to count the lines of code (SLOC). A set of custom regular expressions was then employed to identify and count the lines of log code (LOLC). Although this approach is similar to those adopted in previous studies (e.g., [FZH+14c], [CJ17b], [YPZ12], and [CJ17a]), particular attention was paid to improving the accuracy of the results by manually curating and customizing the process according to each application's logging framework or methods. As a result, custom regular expressions were tailored for each application, allowing all log messages to be collected and multi-line log statements to be properly recognized.

A number of features and parameters related to each log statement were identified. Based on the collected information, it was verified whether the application could support the five forensic tasks outlined in Section 3.1.3.

Timestamps

To support the process of timeline development and visual representation of the sequence of events, it is crucial for an application to provide log entries containing timestamp data.

Log2timeline is a very popular tool that significantly improves forensic analysis by allowing the creation of comprehensive timelines[2]. It is the de facto tool for parsing a variety of log files from different data sources, thereby creating detailed timelines that present a chronological sequence of events within the system. However, application logs lacking timestamp data become less useful for forensic analysis, especially when using tools like Log2timeline. In such scenarios, Log2timeline treats these logs as basic 'raw data' with limited analytical potential. This limitation impacts the ability to establish events in a chronological order, detect anomalies and patterns, and assess the duration and frequency of events.

Timestamps can be introduced explicitly by the developer into the logged message, or they can be implicitly added (and often configured) by the logging framework adopted by the application.

To determine whether timestamp data was present in the logs of the studied applications, a series of steps was followed. First, the programming language and framework used in the application's source code were identified, as logging implementation can vary depending on these factors. Second, the logging configuration files or code were analyzed to determine which logging libraries or tools were employed, providing insight into the available logging syntax and features. Third, logging statements in the source code were examined for the presence of timestamp-related parameters or functions. For example, in Python's logging library, the format specifier was used to indicate the inclusion of timestamp data in log messages. Additionally, the time zones applied by the application during the logging process were observed.

It is usually recommended to standardize timestamp in UTC (Coordinated Universal Time) for logging across all systems. This approach aligns with ISO 8601 standards [ISO],

[1] https://dwheeler.com/sloccount/

[2] https://github.com/log2timeline/plaso

which prescribe the format for timestamps and time zones. According to these standards, timestamps should either be set to UTC or to local time with an offset to UTC. Adopting this practice ensures consistency, reduces complexity in analysis, and assists in accurately correlating events.

To confirm the presence of timestamps and timezone information, the applications were executed and the logs generated during their operation were observed. During this process, it was also verified whether logs related to user actions were available. Overall, a systematic approach was applied to determine whether timestamp data was present in the logs of the applications under study.

Unique Identifier

The presence of unique identifiers allow analysts to filter log entries and focus on specific aspects of interest but also to connect a certain action with other logs or events collected elsewhere. For instance, a log entry that records a specific name of file being accessed by a specific user account can be linked to a network log entry which shows the corresponding user connection being established from the specific IP address.

```
1  //BeeBEEP code snippet
2  bool Core::downloadFile( VNumber user_id, const FileInfo& fi, bool
     show_message )
3  {
4    ...
5    qDebug() << "Downloading file" << qPrintable( fi.path() ) << "from user" <<
     qPrintable( u.path() );
6    mp_fileTransfer->downloadFile( u.id(), fi );
7    return true;
8  }
```

Listing 3.1: Implementation of BeeBeep's log in source code

The code extracted from BeeBEEP in Listing 3.1 is used to log any successful file downloads within the application. The code uses the `qDebug()` function to print a log message at the debug level, which includes the file path and the user's path. This information is useful for detecting specific events, such as the downloading file and execution of potentially malicious files. By keeping track of this data, the application can detect and mitigate any security risks to such file path or user's path.

```
1  //Deluge code snippet
2  def on_download_fail(failure):
3  log.warning(
4  'Error occurred downloading file from "%s": %s',
5  url,
6  failure.getErrorMessage(),
7  )
8  result = failure
9  return result
10
11 def on_download_success(result):
```

```
12  log.debug('Download success!')
13  return result
14
15  d = _download_file(
16  url,
17  filename,
18  callback=callback,
19  headers=headers,
20  force_filename=force_filename,
21  allow_compression=allow_compression,
22  handle_redirects=handle_redirects,
23  )
24  d.addCallbacks(on_download_success, on_download_fail)
25  return d
```

Listing 3.2: Handling successful and failed download events in Deluge

Listing 3.2 showcases an excerpt from Deluge, a Python-based torrent application. The `on_download_success` function acts as a callback, triggered by the deferred object resulting from the `_download_file` function upon successful file download. In this instance, when `on_download_success` is invoked, the application logs a debug message noting the download is success and then returns the input result argument. Merely logging a message like "Download success!" lacks detailed information about the download, rendering it challenging to correlate events with specific actions. It is crucial to incorporate more specifics, such as the file's name or download URL, to enhance the context of a successful download event.

On the other hand, the `on_download_fail` function performs a warning log with the URL and error message of the failure event, which provides more information about the failure and makes it easier to diagnose problems or understand the behavior of the system. With this information in hand, it is easier to determine the action and event identifier related to the logged event.

For this study, the number and percentage of log messages containing three types of unique identifiers (file and path names, user identifiers, and network endpoints such as host names or IP addresses) are investigated. A high number of unique identifiers in application logs is considered beneficial, as it can ease the process of filtering log entries and focusing on specific aspects of event interest, which is crucial for security-related tasks..

To gather this information, each extracted log message from the applications was analyzed to identify the unique identifiers expected to be printed alongside the log content. This process involved examining common names such as 'ipAddress', 'srcIP', or 'url', which are typically used to represent unique network-related identifiers. In cases where naming conventions were not immediately identifiable, the unique identifier was manually traced back to its source declaration in the application's source code. Through this approach, each unique identifier included in the log message was accurately classified.

The amount of text-only log entries – i.e., those reporting fixed messages without any parameters or identifiers was also counted. Such messages are often used by developers as a simple and convenient way to mark specific points in the execution (e.g., errors in data processing) for debugging purposes. However, for security-related tasks, text-only event

logs may fail to provide sufficient granularity and context to detect security incidents or to understand the relationships between log entries. A high percentage of text-only log entries can therefore be interpreted as an indication that the application's logs are not well-suited for event correlation or security investigations.

Partitioning Waypoints

Previous research has shown that by fusing both application and audit logs it is possible to achieve a high level of precision in identifying the origin of an attack [YMZ+21]. This operation requires the presence of particular messages that act as waypoint to partition the provenance graph. Complex application consist of a set of units, such as separate windows, conversation threads, and processing of network connections [YMZ+21, MZW+17, HNDB20]. A unit is made up of a series of transactions, each representing a single, sequential step in the execution of the unit. Therefore information such as the 'TabID', 'FolderID', 'messageID' or 'chatID' can be used to identify the current execution unit, which is otherwise difficult to extrapolate at the `syscall` level. In order for application logs to be used for event partitioning, each execution unit performed and logged by the application should have a unique identifier to indicate the completion of an execution context and the start of another one. Thus, the availability of any unique identifiers that can act as waypoints is investigated. Specifically, it is examined whether logs contain information about when a new document begins processing or when an old one is closed. For applications that do not involve files, the presence of logs containing unique identifiers related to the processing of new network connections or new conversation threads is assessed. If such identifiers are present, they can be considered as potential waypoints that allow an analyst (or an automated system) to partition the application's activity into distinct units.

User Actions

Actions such as downloading an infected attachment, clicking a malicious link, (un)consciously changing settings, and allowing the application to execute untrusted documents can lead to security incidents. Thus, an applications should log any actions which can later be used to investigate or detect attacks or attempts to misuse the software. However, despite the importance of logging sensitive user actions (which is suggested by many guidelines and standards such as ISO 27002), there is no precise checklist on how developers should identify such actions.

In this study, a list of user actions that should be logged for each category of application is proposed, as summarized in Table 3.2. The list includes both the interactions performed by end-users with the application and the interactions initiated by the application with the external environment. It should be noted that the list is not exhaustive and is intended only as a starting point to assess the current state of application logging.

Table 3.2: List of user actions to log for security analysis across various application categories

Application Category	Actions to Log
File Managers	
	Search queries
	Changes to file or document permissions
	File or document executions
	Archive operations
Chat and Communication	
	Login activities
	Clicks on shared links
	Unique, randomized, non-identifiable IDs for conversation activities
	Attachment downloads
Office	
	File executions with embedded scripts
	URL clicks within documents
	Attempts to access restricted files
	Macro executions
	Changes to settings
Torrent and File Transfer Clients	
	Files downloaded
	Seeding activities
	Search queries
	Peer connections
Screen Capture	
	Screen capture activities
	Saved screen captures
Security / Antivirus	
	Login activities
	Acceptances to run quarantined files
	Setting changes (e.g., enabling/disabling scanners)
Text Editors	
	Installation and usage of external plugins
	Link or URL clicks
	Document openings and edits
Utilities	
	Service starts
	Download or save activities
	Installation and usage of external plugins
Media Players	
	Opened files
	Data exports
	External subtitle imports
	Media play starts
	External plugin or add-on installations
	Runs of external plugins or add-ons
Window Managers	
	Setting changes

Continued on next page

Table 3.2 — *continued from previous page*

Application Category	Actions to Log
	Session starts
Remote Desktop	
	Login activities
	Remote connection starts
	Failed connections
Application Launchers	
	Search queries
	Applications or documents launched
	Setting changes

Log Levels

Log levels are used to configure the amount of recorded information and organize log data. For instance, `trace`, `debug`, and `info` levels usually provide detailed data that can be used for debugging and application performance analysis. On the other hand, `warn` and `error` level logs can be centralized to provide information on unusual events, which might be associated to security threats. From a security perspective, the presence of a log level can help to determine whether an event needs immediate investigation, could be deferred for a later analysis, or can be safely ignored. This prioritization, results in more efficient and effective security monitoring and analysis.

Therefore, to evaluate the effectiveness of application logs for detecting misuse, the default log-level configuration is assessed, along with the consistency and availability of log levels associated with specific user actions and events.

Exploitation

While system-level monitoring can be capable of detecting when an application is compromised, it is often insufficient to determine the cause (i.e., the input associated to the attack) which is a valuable information for post-incident analysis and incident response. In principle, application logs can help security analysts to better understand how the attack occurred and identify the specific actions that led to the security incident. This information can be used to identify the root cause and remediate vulnerabilities in the application. Furthermore, these logs can help in forensic investigations to understand how the attacker gained access to the system, the actions they took, and what data they managed to access.

Since logging all external inputs is considered impractical and embedding a custom anomaly-detection approach to identify unusual cases is recognized as highly complex and error-prone, it becomes challenging to assess whether sufficient information is included in application logs to capture malicious inputs. As an initial step, the extent to which logs cover all exception blocks in the program is measured, along with whether the data responsible for the exceptions is reported.

In addition, five case studies are conducted in which Proof of Concept (PoC) exploits, collected from the Common Vulnerabilities and Exposures (CVE) database, are executed against vulnerable versions of five applications in the dataset. After each successful exploitation, the generated logs are analyzed to identify any entries that may be used to

detect and characterize the attack. The availability of these log entries and their relevance to incident response and forensic investigations are also evaluated.

Availability

The availability of log information depends on the application's logging configuration. Some applications provide debug logs or runtime logs by default to improve the application or to facilitate the process of reporting unexpected crashes to the developers. Others may only log errors and require users or administrators to manually edit configuration files to tune the amount of logged information. Finally, some applications do not enable logging by default nor provide options to start logs in the deployment.

To determine the default availability and configuration of logs for each application, the methodology is initiated with a thorough review of the application's documentation, where references to the default availability of logs or any user-configurable options for logging output are examined. For applications in which the documentation does not provide clear information regarding log availability or configurability, the applications are executed in a host environment. This enables the manual inspection of any log-related information or settings that may be enabled or configurable.

3.3 Inadequacy of Open-Source Application Logs for Digital Forensics

Table 3.3: Logs properties of studied applications.

Application	Logging utilities	Language	SLOC	LOLC	% Log statements
File managers					
Files — Nautilus	GLib Message Logging	C	98 739	128	0.13
Disks	GLib Message Logging	C	20 724	68	0.33
Dolphin	Qt — QMessageLogger	C++	40 003	47	0.12
nnn	Own logging facilities	C	11 953	118	1.00
recoll	Own logging facilities	C++	93 972	1 758	2.34
Konqueror	Qt — QMessageLogger	C++	52 191	370	0.72
Thunar	GLib Message Logging	C	65 835	105	0.16
Chat and Communication					
IceChat (Windows)	Windows EventLog Class	C#	61 981	261	0.42
BeeBEEP	Qt — QMessageLogger	C++	61 392	1 147	1.89
Pidgin	GLib Message Logging	C	169 056	1 320	0.78
signal-desktop	Pino Logging	Typescript	208 740	1 384	0.66
mattermost desktop	electron-log	Typescript	10 945	224	2.05
Konversation	Qt — QMessageLogger	C++	47 393	259	0.55
jitsi	SLF4J	Java	301 383	2 858	0.95
wire-desktop	electron-log	Typescript	5 677	212	3.73
session-desktop	Bunyan logging API	Typescript	57 841	609	1.05
HexChat	Own logging facilities	C	62 090	611	0.98
Terminal- Irssi	GLib Message Logging	C	63 755	440	0.69
tox	Own logging facilities	C	48 274	442	0.92
Office					
Calligra	Qt — QMessageLogger	C++	660 504	698	0.11
Xournal++	GLib Message Logging	C++	52 221	234	0.46
SumatraPDF	Own logging facilities	C++	1 059 517	391	0.05
zael	Qt — QMessageLogger	C++	13 886	36	0.26
Okular	Qt — QMessageLogger	C++	91 398	434	0.52
xpdf	Own logging facilities	C++	125 529	1 069	0.85
ApacheOpenOffice	Own debug macro	C++	4 591 378	15 235	0.33
Scribus	Qt — QMessageLogger	C++	390 442	515	0.13

Continued on next page

Table 3.3 — *continued from previous page*

Application	Logging utilities	Language	SLOC	LOLC	% Log statements
Evince-DocumentViewer	GLib Message Logging	C	81 644	572	0.70
Torrent and file sharing platform					
qBittorrent	Qt — QMessageLogger	C++	58 813	437	0.74
Deluge	Python logging facilities	Python	43 956	1 266	2.88
transmissiongtk	Own logging facilities	C++	87 299	193	0.22
frostwire	Java logging utilities	Java	148 930	942	0.63
warpinator	Python logging facilities	Python	4 111	134	3.26
Screen-capture					
ShareX	.Net logging utilities	C#	115 016	236	0.21
OBS-Studio	Own logging facilities	C	340 593	924	0.27
Greenshot (Windows)	Log4net	C#	53 293	684	1.28
Security or Antivirus					
Seahorse	Qt — QMessageLogger	C	16 901	88	0.52
hashcat	Own logging facilities	C	198 410	1 519	0.77
ClamAntiVirus (Linux)	Own logging facilities	C	215 996	1 492	0.69
Text editor					
Zettlr	Own logging facilities	Typescript	23 770	89	0.37
gnome-text-editor	Qt — QMessageLogger	C	24 903	48	0.19
Vim	Own logging facilities	C	399 870	140	0.04
jrnl	Python logging facilities	Python	4 000	22	0.55
Utilities					
Console	GLib Message Logging	C	5 856	23	0.39
Cheese	Gstreamer debugging tool	C	6 624	17	0.26
Guvcview	Own logging facilities	C	30 320	901	2.97
XDM	Java basic log utilities	Java	27 955	522	1.87
Media player					
VLCMediaplayer	Own logging facilities	C	607 435	454	0.07
musikcube	Own logging facilities	C++	365 945	122	0.03
Kodi	spdlog	C++	732 891	5 619	0.77

Continued on next page

Table 3.3 — *continued from previous page*

Application	Logging utilities	Language	SLOC	LOLC	% Log statements
Window manager					
IceWM	Own logging facilities	C	70 029	621	0.89
tmux	Libevent	C	63 656	647	1.02
bspwm	Own logging facilities	C	12 147	198	1.63
Qtile	Python logging facilities	Python	38 947	259	0.67
Remote desktop					
Remotely	.Net Logging Basics	C#	58 550	256	0.44
xrdp	Own logging facilities	C	84 133	1 089	1.29
App launcher					
albert	Qt — QMessageLogger	C++	4 264	96	2.25
launchy	Qt — QMessageLogger	C++	11 735	33	0.28
ulauncher	Python logging facilities	Python	6 035	96	1.59
Wox	Own logging facilities	C#	15 367	150	0.98

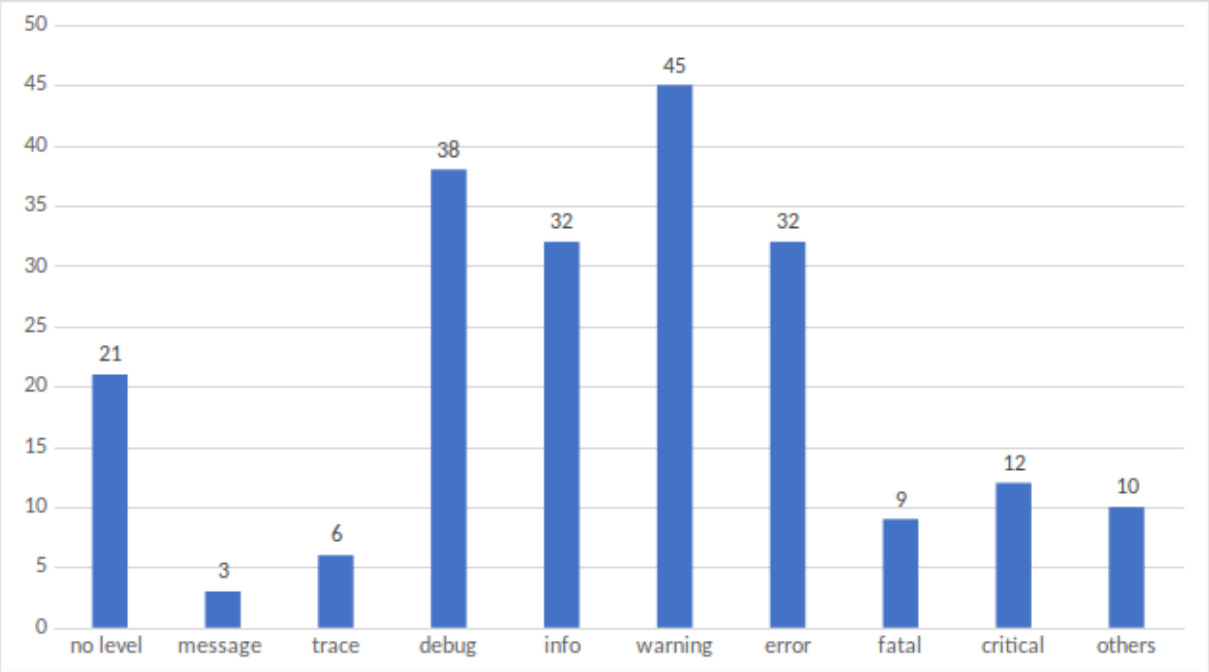

Figure 3.1: Usage of log level.

In this section, the results of the measurement study are presented. Table 3.3 is provided to give an overview of the overall size and amount of logging statements in each application, along with the primary programming language and the library or framework (if any) used for logging.

As a rough indication of the amount of logging, the last column shows the ratio between the number of lines of code and the number of lines dedicated to logging. The value varies between 3.7% for the wire-desktop encrypted communication software to 0.03% for the musikcube application (with a median of 0.68% and a mean of 0.88%)

It was found that application logs were enabled by default in 11 of the studied applications, while 49 did not provide logs by default, as indicated in Table 2. However, among the applications that do not have the feature of logging enabled by default, many provide a configuration or options for users to enable logging. In fact, out of the 49 studied applications that did not provide logs by default at runtime, 47 offers users the capability to enable logging through configuration or options at runtime. This capability can be leveraged for debugging, system analysis, or even execution partitioning. However, it was also observed that the xpdf and XDM applications do not have the capability to produce runtime logs or provide any options for logging to be enabled.

Figure 3.1 shows the number of applications that use the different log levels. It is interesting to observe that 21 of them generated messages without any specific level. 'Warning' was identified as the most commonly adopted level by the applications in the dataset, but a number of unconventional levels, such as 'silly', 'advice', and 'chatter', which has been classified as 'others'.

Additionally, inconsistent log levels used for various user actions were revealed through the analysis of the examined applications. For example, in the 'Utilities' category, the applications produced different log levels for the same user action. For instance, when a user starts the application, Cheese generates logs with the 'Info' level, while Guvcview

and XDM produce logs without any level. This variability in log levels used by different applications can make it difficult to generalize the analysis, thus prolonging the process of security investigations.

3.3.1 Application Logs for Timeline Development

Timestamps were included in every log entry in only 31 out of the 60 applications analyzed in the study. Moreover, the implementation of timestamps varied among different application categories and is dependent on the implementation of the application's logging utilities. Of the 29 applications that do not include timestamps, 12 implemented their own logging utility, 9 used Qt's `QMessageLogging`, five `GLib` Message Logs, one the default `Java`, one with Windows Event Log and one with `.Net` logging libraries, as presented in Table 2. It is interesting to observe that some applications do not include timestamps also when the logging frameworks they use support this feature. Even worse, in these cases, no configuration variable was found to have been defined by the developers to include timestamps in all logs. In the investigation of timestamp granularity, as presented in Table 3, it was observed that while a logging module is used by most applications, the date and time format is often defined by the developers themselves. Due to this flexibility and lack of standardized guidelines, applications can differ significantly in their timestamp presentations. The majority of the applications log timestamps up to the granularity of seconds. However, 17 of them extend this to milliseconds. Notably, tmux logs timestamps up to the microsecond level. It was noted that the date and time are logged by hashcat application solely at the commencement and conclusion of particular operations, such as the initiation of a hashing process. Regarding the availability of date data, 11 applications lacked date information. 20 logged both the month and date, but 12 out of 31 did not include the year in their logs. While the exact year might not always be crucial for security analysis, this inconsistency in timestamp presentation could complicate log analyses, especially when constructing detailed timelines using application logs. According to the data, the timestamp timezone was set to UTC (Coordinated Universal Time) in only four applications, while the others used the system's local timezone for storing log entries. Storing logs in the UTC format or local time with an offset to UTC is a widely adopted practice, in line with ISO 8601 standards, in both software development and system administration. This universal standard is not affected by changes in time zones or daylight saving time adjustments, making it a crucial consistency factor when dealing with systems or users across multiple time zones. This ensures that log timestamps remain uniform and unchanged across different geographical locations. Furthermore, in the process of analyzing logs, particularly during incident response, having a single, consistent time reference like UTC simplifies the process, as there is no need to convert times from different time zones to understand the sequence of events.

This emphasizes the importance of providing developers with clear guidelines for implementing timestamps in application logs.

3.3.2 Application Logs for Event Correlation

As shown in Figure 1, it was found that 77% of the applications in the dataset contain more literal log messages (i.e., hardcoded strings with no variable parts) than log with data containing user, file, and network identifiers combined. The information provided

by text-only messages is limited, which diminishes the effectiveness of logs for practical investigation. Table 1 shows the actual amount of application log messages that contain unique identifiers related to users (username, account IDs, or nick names), network (IP addresses, domain names, and URIs), and file information (file and directory names). A considerable variation in the value was observed among the applications in the dataset. For instance, 15.4% of the log messages of BeeBEEP report the associated user identifier, 17.4% of the messages of Konversation contains unique network identifiers, and 36.1/% of the messages of Zael have an associated filename. Conversely, 11 applications didn't record any network-related unique identifiers (UIDs) in their logs and four applications did not log any file UIDs at all. However, the decision to log UIDs associated with network or files largely depends on an application's core functionality and in some cases it might not be necessary or not apply at all. Applications categorized under File Manager, Office, and Security/Antivirus often integrate features like cloud functionality, updates via network, cloud-based storage, or collaboration tools. Specifically, for Security and Antivirus applications, logging is indispensable for tracing malicious activities, updating databases, or connecting to central databases. When examining the number of log statements in each application, the lowest counts in the dataset were observed in Console (a terminal emulator) and Cheese (a webcam tool for capturing photos and videos), with only 23 and 17 log statements in their source code, respectively. In both cases, no unique identifiers were found across the three examined areas. This prevents any possible correlation with other sources of information and severely limit the use of these logs for forensic purposes. Finally, it should be stressed that the study only assesses the broad availability of unique identifiers and further research is necessary to understand the usefulness of these UIDs in actual investigations.

3.3.3 Application Logs for Execution Partitioning

Application-generated logs can offer valuable insights on the usage patterns and behaviors of users, as well as detailed information about the inner workings of the application, including its interactions with other systems. Previous research has shown that these logs can be effectively leveraged for execution partitioning to derive precise attack provenance graphs. Logs that are not enabled at runtime may not be accessible for security analysis, and this can greatly decrease the possibility of security monitoring and analysis. Previous research on utilizing application logs for execution partitioning highlighted the importance of including unique identifiers such as file names, IPs, or URIs to represent object of data in a single transaction of logs. These types of information can be leveraged to identify or gain insight into the execution recorded by the application. Thus, the presence of unique identifiers was examined in relation to operations such as opening or closing a file, or establishing or terminating a network connection, as a means to indicate the beginning of a new execution unit in the logs. The outcomes derived from the use of application logs for execution partitioning are presented in Table 2 in the Appendix. It was revealed through the analysis that, out of the examined applications, 37 produced log entries containing at least one unique identifier associated with the ongoing execution unit. The distribution of these unique identifiers varied among the different application categories. For instance, among the file management applications, only one included a unique identifier in its log entries. Meanwhile, two out of the applications in the chat and communication category did not include any unique identifiers. In contrast, all of the

applications in the screen capture, media player, and remote desktop categories contained at least one unique identifier in their log entries.

```
--
(16:06:22) account: Connecting to account ###@irc.###.com.
(16:06:22) connection: Connecting. gc = 0x556421bc17d0
(16:06:22) dnsquery: Performing DNS lookup for irc.ubuntu.com
(16:06:22) Session Management: Received first save_yourself
(16:06:22) dns: Created new DNS child 5341, there are now 1 children.
---
---
(16:11:22) prefs: /pidgin/conversations/toolbar/wide changed, scheduling save.
(16:11:22) gtkconv: setting active conversation on toolbar 0x55df14133d90
(16:11:28) util: Writing file prefs.xml to directory /###/###/.purple
(16:11:28) util: Writing file /###/###/.purple/prefs.xml
(16:11:31) gtkconv: setting active conversation on toolbar 0x55df13ae2000
(16:11:32) gtkconv: setting active conversation on toolbar 0x55df14133d90
(16:11:33) server: Leaving room: [name of chat room]
(16:11:33) gtkconv: setting active conversation on toolbar 0x55df13ae2000
(16:11:33) irc: Got a PART on [name of chat room], which doesn't exist --
  probably closed
(16:11:39) util: Writing file blist.xml to directory /###/###/.purple
(16:11:39) util: Writing file /###/###/.purple/blist.xml
(16:11:42) roomlist: unreffing list, ref count now 0
(16:11:42) roomlist: destroying list 0x55df13f35580
(16:11:44) account: Disconnecting account ###@irc.###.com
  (0x55df139226f0)
(16:11:44) connection: Disconnecting connection 0x55df13d97a60
(16:11:44) connection: Deactivating keepalive.
(16:11:44) connection: Destroying connection 0x55df13d97a60
(16:11:44) certificate: CertificateVerifier tls_cached unregistered
(16:11:44) certificate: CertificateVerifier singleuse unregistered
(16:11:44) certificate: CertificatePool tls_peers unregistered
(16:11:44) certificate: CertificatePool ca unregistered
(16:11:44) main: Unloading normal plugins
(16:11:44) plugins: Unloading plugin Message Notification
---
```

Listing 3.3: Pidgin log entries

Listing 3.3 shows a Pidgin log entries when user make login to the application, changing the preference of the application, leaving chat room and logging out from the chat application. The logs generated by the applications provide a comprehensive view of the actions taken within the application, including user login information, file save events, and changes in preferences. For instance, there are a log entries shows changes in preference for the chat program (prefs: /pidgin/conversations/toolbar/wide changed, scheduling save), leaving a chat room with the name of room (server: Leaving room: [name of chat room]), and connecting and disconnecting from the account. The log also shows that the program is writing and saving files, such as `prefs.xml` and `blist.xml`, to the user's file directory.

When the log entries were examined, it shows that for each new unit of execution, there are log entries that contain unique identifiers such as the application account UID when user connections were made or server names when user connected to chat server. Additionally, the application logs provide clear execution units for every log entry, making it easier to identify and categorize actions. For example, actions related to a user's account will be categorized as 'account' and actions related to a server will be categorized as 'server'. The clear representation of this log entries can facilitate the process of parsing and partitioning of execution units. On the other hand, Listing 3.4 shows instead the negative example of Calligra's logs, where entries are primarily related to errors and configuration issues. These entries indicate problems like missing icon themes, inability to access files or directories, and challenges initializing features or setting shortcuts. Though these log details are essential for enhancing the application's performance, they don't help in execution partitioning because they lack data on the made execution units. To use this log for partitioning, one would need a deeper analysis to spot usage patterns and object allocation in the application.

```
1  ---
2  Icon theme "breeze" not found.
3  connect failed: No such file or directory
4  Hspell: can't open /usr/share/hspell/hebrew.wgz.sizes.
5  kf.sonnet.clients.hspell: HSpellDict::HSpellDict: Init failed
6  CalloutPathTool::CalloutPathTool(KoCanvasBase*) QAction(0x55e2cc8079b0
     text="To Path" toolTip="To Path" shortcut=QKeySequence("P")
     menuRole=TextHeuristicRole visible=true)
7  kf.xmlgui: Shortcut for action  "object_order_raise" "&Raise" set with
     QAction::setShortcut()! Use KActionCollection::setDefaultShortcut(s) instead.
8  kf.xmlgui: Shortcut for action  "object_order_lower" "&Lower" set with
     QAction::setShortcut()! Use KActionCollection::setDefaultShortcut(s) instead.
9  kf.xmlgui: Shortcut for action  "object_order_front" "Bring to &Front" set with
     QAction::setShortcut()! Use KActionCollection::setDefaultShortcut(s) instead.
10 ---
```

Listing 3.4: Caligra log entries

3.3.4 Application Logs for Misuse Detection

To understand the use of application logs for misuse detection, a list of user actions across different categories of applications was compiled. The list includes both events related to the end-user's interactions with the application and the events related to the application's interactions with the external environment. This list of actions guide us in identifying what the application should log to enable post-attack analysis. Logging user actions can provide a record of what actions were taken within application, allowing to identify patterns of behavior that may indicate misuse, such as repeated access to unauthorized resources. Each action was performed on the studied applications, and the logs they produced were analyzed. The results as presented in Table 4. Inconsistencies were revealed through the analysis of log content and levels among applications within the same category, particularly in how user actions were logged. For example, in the chat and communication category, BeeBEEP was found to generate detailed logs of all

studied user actions at the Debug log level, while Irrsi-Client and Jitsi did not produce any logs for the tested user actions. In particular, as part of the testing of chat and communication applications, it was examined whether users were given the option to record or log conversations. All five applications are tested in this category provided the option for users to save conversation logs on their host computers. This can be useful for future reference or to recall important details from earlier conversations. However, this raises significant privacy concerns as these logs are stored in plain text and include names and body of messages. On the other hand, it have been discovered that most user actions which can be used for forensic purposes, such as downloading files and clicking on shared links, are also logged in the conversation log. Moreover, in the office application category, most of the studied applications did not generate any logs regarding user actions. Okular was the only exception, producing a 'Debug' log when a user clicks on an embedded URL.

3.3.5 Application Logs for Attack Detection

As discussed in the previous section, it is very difficult for an application to detect exploitation attempts. One aspect that, while insufficient, might help in this regard is to log any anomalous or error condition. Logging, particularly at these detected error junctures, is crucial as it sheds light on potential issues before they manifest as overt failures [YPH+12b]. One way for programs to detected unexpected errors is through exceptions and therefore it is believed that logging these conditions (and the data that might have been responsible to trigger them) can be a first step for an applications to record malicious undertakings or potential attacks. Thus, a consistent logging approach in exception blocks yields profound insights into an application's behavior, facilitating the identification of any unusual patterns suggestive of security threats or breaches. However, Table 5 reveals that only 25 out of 60 applications implemented exception handling, due to the fact that exceptions are not supported in all programming languages. Among them, Open Office has the highest number of exception blocks. GreenShot is instead the application with the largest fraction (77.4%) of logging statements in exception blocks, followed by Mattermost-Desktop at 74.51%. Only five application (Transmission, Ulauncher, IceChat, Warpinator, and Wox) logged all exception statements, aiding in providing valuable information for troubleshooting and resolving issues. In contrast, three apps (Jrnl, Qtile, and Recoll) did not produce any log related to exceptions. It is concluded that this inconsistency is due to the absence of standard guidelines for logging in application development. Whether application logs are sufficient to detect and investigate a successful exploitation is a very complicated question, as it vastly depends on the type of vulnerability and the exploit itself. Therefore, it is difficult to answer this point by simply looking at the log messages themselves. Instead, a separate study was conducted in which five applications from the dataset were selected and tested using public exploits information. The application version, platform, and exploitation type, are shown in Table 3.4.

Table 3.4: Logs generated during exploitation of selected applications

Application	Version	Tested platform	Type of exploitation
Evince	3.17.1	Ubuntu 16.04.6	Execution of arbitrary command
Nautilus	3.4.2	Kali 1.0.6	Execution of arbitrary command

Continued on next page

Table 3.4 — *continued from previous page*

Application	Version	Tested platform	Type of exploitation
LibreOffice Writer	6.2.5	Ubuntu 18.04	Directory traversal attack
VLC Media Player	2.2.8	Windows 10	Execution of arbitrary code via MKV files
HexChat	2.10.0	Ubuntu 22.04	Directory traversal attack

Considerable variance in the utility of application logs for attack detection was indicated by the study, subject to the specific application and the nature of the attack. While three applications logged events during the exploitation, these logs lacked definitive indicators of successful breaches. The absence of logs that show any exploitation makes it harder for analysts to use them for detecting security threats. For example, in the study involving VLC Media Player, even when the logging has been turned to the highest logging level, no logs were produced during an attack. This lack of logs, including user actions, highlights challenges in using application logs from VLC for security purposes. In contrast, with HexChat, when users enable logging, the application records vital details like the connected server's IP, connection status, and chat history. This can offer insights into potential threats. Yet, this logging is only active when a user permits it. Many might not do so due to privacy concerns.

3.3.6 Assessment Overview

Based on the analysis conducted on 60 popular desktop applications, it was determined that their logs are not well suited for forensic purposes. In fact, the findings showed that:

1. 29/60 of the applications did not include timestamps in their log entries, making it challenging to determine the chronological order of events.

2. More than half of the applications produce log messages consisting mainly in constant text strings. Compared to messages that contain identifiers like user, network, and file data, this emphasis on text-only logs complicates event correlation.

3. 23/60 of the applications did not log any unique identifier at the beginning of a new execution unit, making it challenging to define new execution units and objects involved in those units.

4. Inconsistent logging of specific user actions across different applications makes it difficult for application and security analysts to understand patterns of behavior that may indicate misuse.

5. 35/60 of the studied applications did not use exception handling. Out of the 25 remaining applications, only five applications thoroughly logged exception statements. Thus, valuable information about unexpected errors and potentially vulnerable features or code could go unnoticed.

6. Successful exploitation often does not result in any relevant log data, and even when such data is produced, it is not available by default.

In conclusion, the results of the study highlight several significant challenges in leveraging application logs for security analysis. Even when logs are enabled, the deficiency of timestamps in 29 of the studied applications, the high proportion of text-only log events in over half of the applications, and the inconsistent logging of user actions and unique identifiers across different applications make it difficult for security analysts in performing thorough assessments. These findings highlight the need for improved logging practices for better logging to aid security analysis and incident detection.

3.4 Improving Practices Based on Observations

The implications of the findings are discussed, along with potential directions for future work.

Needs of Application's Logs for Forensic. Based on the analysis conducted on 60 popular open-source applications, it was found that the current logging practices of desktop software are largely inadequate, limiting the effectiveness of their logs for forensic purposes. In particular, it was observed that while these logs are rich in details, they often lack critical data, such as consistent timestamps and unique identifiers, which are essential for effective forensic analysis. This findings highlight the necessity to align application logs with best practices already adopted in other sources, such as operating systems and networks.

Investigators frequently compile comprehensive super-timelines during incidents or attack investigations. These timelines typically involve data collected from a variety of sources, each offering distinct insights and perspectives, and contributing to a more extensive and fine-grained analysis. It is crucial to understand that this research does not suggest to rely solely on application logs. Instead, it emphasizes their potential use to complement existing data sources, thanks to the unique contextual information application logs can provide.

In fact, it is believed that the application logs can play a significant and specific role in the context of an investigation. The availability of their data can be crucial to fully understand the the interaction of users with external data or events – connecting the dots provided by other logs.

Sadly, due to their inadequacy, these logs are still under-represented in broader forensic studies. Understanding their limitations, which is the goal of this study, is crucial to mitigate this problem.

Solutions to Enhance Application Logging. This chapter emphasizes a number of issues and inconsistencies with application logs and emphasizes the importance of proposing actionable solutions. Unfortunately, there are limited guidelines and practical solutions proposed in the literature that focus specifically on the forensic use case. Even worse, there is a lack of static analysis tools that can help developers to identify (and improve) logging issues during the development lifecycle. It is believed that the proposed taxonomy can be used by researchers to advance in both directions: to first create guidelines of *what* needs to be logged, and then to develop tools that can automatically enforce or verify that the implementation follows these guidelines.

Certain things are easy to fix, such as inconsistencies in the timestamp formats or the use of predefined log levels. These cases can be fixed by adopting a proper and

standardized log framework or library. Other issues, like the lack of unique identifiers or the absence of log statements that precisely report error and exceptions, are more complex. Finally, some of the aspects are specific to a certain class of applications and thus need to be addresses on a case-by-case basis. This is the case, for instance, of the need to report when an application parses new external and user-provided data or when (and how) it interacts with the rest of the environment.

Finally, new forensic tools need to be developed to take advantage of this information and help investigators to combine and correlate application logs with other sources.

Overall, the collaboration between software engineering, cybersecurity, digital forensics, industry, and academia is essential for developing these guidelines and tools.

Chapter 4

Ensuring Security-Focused Log Placement and Consistency

Application logs play a critical role in forensic investigations, supporting tasks such as timeline reconstruction, event correlation, execution partitioning, misuse detection, and attack detection (as discussed in Chapter 3). However, the value of logs as forensic evidence depends not only on their existence but also on where they are placed within a program's execution.

This chapter addresses the second research objective of this thesis: evaluating the consistency of log coverage around the execution of critical actions. The goal is to determine whether application logs are placed consistently and effectively within the source code to capture security-relevant data along critical paths. Logs positioned along critical paths, those that connect user inputs to security-sensitive operations are vital. These are the paths most often exploited by attackers. Placement matters because logs recorded after a critical operation may never be reached in the event of crashes, compromises, or unexpected shutdowns. For example, if an application executes a shell command where part of the command-line is constructed from external input, a possible command injection can be used to kill the application, preventing the post-execution log statement from being reached. Another important aspect related to log placement is the fact that the generation of logs can be, and often is, guarded by a conditional branch. This means that the log is generated only under particular conditions, for instance, on the content of the parameter for pre-execution logs or on the result of the action for post-execution cases. All these placement aspects can lead to silent failures or to missed security-relevant events. This oversight is particularly concerning in the context of security and forensics, where logs should be placed on the same control-flow path as the operation they refer to, in order to ensure that sensitive operations always leave behind traces that reflect the actions performed within the critical operation.

To address these challenges, this chapter investigates the strategic placement of logs along critical execution paths. Building on the measurement study in Section 3.2, which examined 60 open-source applications, the focus here shifts from evaluating what information is logged to where logs are positioned within relevant control flows. This shift in perspective is crucial for effective attack detection, as identifying an attack requires logs that capture not only abnormal outcomes but also the triggering causes and contextual data along all possible execution paths.

Because reasoning about all possible execution paths is complex and error-prone when

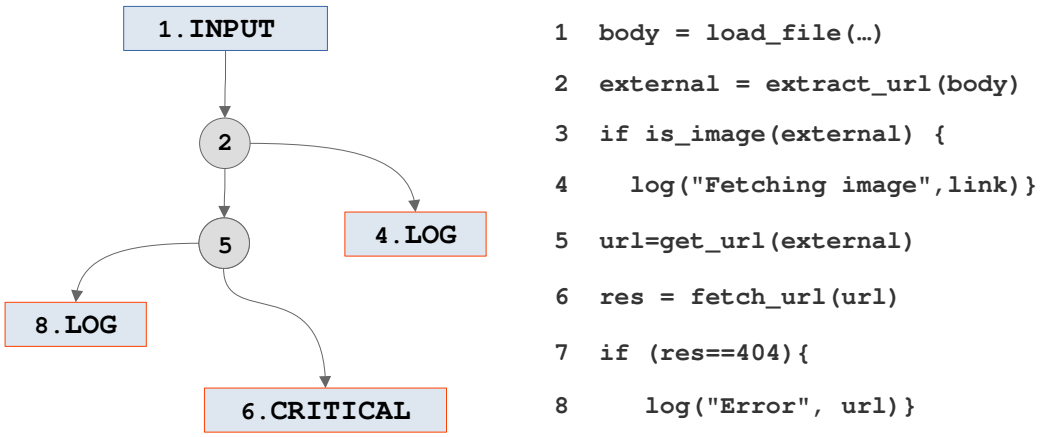

Figure 4.1: Pseudocode and data-flow graph of a motivation example.

done manually, a custom analysis tool, BlindSpot was developed, for this study. Built on the Joern static analysis framework, BlindSpot identifies inter-procedural data-flow dependencies between untrusted inputs and the parameters of critical function calls. It then automatically inspects both preceding and following execution paths to determine whether logs are positioned in ways that reliably capture these sensitive actions. By leveraging information from the dominator tree and control-flow graph, BlindSpot identifies log statements that are guaranteed to execute, either before or after a critical operation.

This chapter concludes that, despite the importance of pre-execution and post-execution visibility, logs are often missing, inconsistently placed, or guarded by conditions that allow execution to proceed without leaving a reliable trace. Many critical operations are completely overlooked or, in the best-case scenario, only partially and inconsistently logged. This inconsistency finds limitation in the consistency of log coverage around the execution of critical actions. In particular, sensitive operations influenced by external input often proceed without leaving observable traces, emphasizing the need for clearer guidance on the best placement of logs within source code to ensure reliable, security-relevant data capture.

4.1 Background Scenario and Research Questions

This chapter presents a systematic study of log placement to capture meaningful security and operational insights, particularly in relation to the execution of critical calls influenced by user input. To achieve this, the study considers three types of instructions in the program. **Input nodes** represent points where external, potentially attacker-controlled input enters the program (e.g., by reading an untrusted file). **Critical call nodes** are points in which the program executes significant actions on the system, e.g., by executing a system command or by fetching data from a given URL. Finally, **Log nodes** are responsible for recording a new entry in the application log. To understand the rela-

tionship among these nodes, the study constructs an inter-procedural data-flow graph. For instance, lets consider a simple example in which a program loads a document from disk, extracts the URL of an external element from it, and fetches the remote content. A simplified pseudo-code, along with its data-flow graph, is shown in Figure 4.1. In the graph, the three types of nodes mentioned above can be easily identified: one input node where untrusted data is read by the application, one critical node where the data is used in a sensitive operation, and two associated log statements. Now the flow of information along the program and the different nodes have been identified, the goal is to study the placement of the log nodes with respect to these critical paths and how branches in the control flow affect the availability of logs upon the execution of an untrusted critical call.

In particular, this chapter want to address these three research questions:

Q1 Presence – Does the program log the fact that it is executing a critical action? Does it also log the parameters?

Q2 Position – Are the log statements placed *before* or *after* the execution of the critical call? Is this placement consistent across the program?

Q3 Gaps – Are these critical log statements independent of the control flow, or is there a path in which the log calls can be avoided?

This allows the assessment of logging coverage for critical actions and the identification of potential monitoring gaps, focusing on cases where user input influences the behavior of security-sensitive operations. In the toy example, that the program does log the parameter of the critical call in two places is presented, one before the call is executed and one after. However, the first log is only performed if the external element is an image, and the second only in case of errors. Thus, there are paths in the program in which the code fetches the remote component without leaving any corresponding trace in the application logs.

Threat Model – The work is motivated by adversaries who exploit untrusted input to trigger sensitive operations while avoiding reliable forensic traces. The model determines whether, and in which locations, the application contains execution paths that allow attacks to proceed without generating log entries. Post-execution logs also create opportunities for such concealment, especially in cases of crashes or forced termination. From a defender's perspective, logs form the foundation for incident response and forensic reconstruction. Analysts must be able to attribute sensitive operations to user input and reconstruct the execution trail from the information of the logs. When log statements are absent, or when they exist only after execution, the forensic record becomes incomplete. BlindSpot aims to expose precisely these blind spots in logging coverage, helping developers strengthen log placement and enabling analysts to better understand the limitations of available evidence.

While this observation is simple to make, it is important to note that the data-flow graph alone is not sufficient to answer the questions. In fact, the dataflow in Figure 4.1 does not address what is the control-flow relationship among the various nodes, which one executes first, and whether there are paths in the program that can circumvent any of the instructions. For this, a more sophisticated program analysis phase is required, which is detailed in the next section.

4.2 Methodology

The approach is built on top of the Joern [Joe] open-source static analysis platform. **Joern** is a static analysis tool designed for vulnerability research and program auditing. It builds a *Code Property Graph* (CPG), which combines code structure, control flow, and data flow into a unified representation. Joern tracks how data moves from *sources* (e.g., user input functions) to *sinks* (e.g., critical or sensitive operations) using taint analysis. This reveals all possible data flow paths, including method calls, variable assignments, and value propagation. We leverage this source-to-sink tracking concept to understand how input data reaches critical operations which allowing us to reconstruct control flow, dominance relationships then to understand logging coverage along valid execution paths.

Input Nodes. To systematically define the program input, functions that retrieve data originating from users or external sources are identified and analyzed. For instance, a GUI application could collect standard user input by calling functions such as `gtk_entry_get_text` for text entry fields, `gtk_text_buffer_get_text` for buffer-based input, and `gtk_file_chooser_get_filename` for file selections. In addition to direct user interaction, input definition extends to data received via indirect mechanisms, including drag-and-drop events (`gtk_drag_get_data`), hyperlink interactions (`gtk_link_button_get_uri`), and network-based input from sockets or streams (e.g., `read`, `g_input_stream_read`, `g_socket_receive`). This categorization ensures that both explicit user actions and implicit data receptions are consistently captured as potential sources influencing critical operations.

Critical Nodes. Zhang et al. and Li et al. offer a broad definition of critical operations, describing them as function calls, assignments, or control statements that an attacker can directly or indirectly influence by manipulating certain variables or conditions [LZL+20, ZLP+21]. While their definition is provided in the context of vulnerability detection, it provides a conceptual foundation to understand critical operations in for this study. Building on this foundation, the concept of critical points and define them as any functions responsible for executing security-sensitive operations were adopted, serving as potential points of vulnerability or critical functionality in the application code. These operations represent security-relevant actions that may serve as potential failure points (e.g., application crashes) or reasons of adversarial behavior (e.g., external command execution). Based on the characteristics of the critical actions, critical calls are categorized into three types: file operations, URL handling, and system execution. File-operation critical calls include any method that interacting with the filesystem, such as reading, deleting, moving files, or managing directories. URL-related critical calls refer to functions responsible for launching or handling interactions with external or shared URLs. Finally, system-execution critical calls involve invoking shell commands through functions like `system` or `popen`. Functions are categorized as critical calls based on the actions summarized in Table 4.1, reflecting their role in executing security-relevant operations.

Log Nodes. Log nodes represent log statements inserted by the developers to capture information about the application's behavior during its execution. These logs may contain static messages, dynamic runtime data, or both, helping developers monitor, debug, or audit the application's actions. For each application we studied, we systematically identified the set of log nodes by analyzing the specific logging functions used by developers, such as `g_warning`, `g_info`, etc.

Table 4.1: Categorization of critical calls

Category	Actions
URL Actions	Clicking an external URL, clicking a shared URL, opening a URL from external sources (e.g., PDF viewer)
File Operations	Opening a file, navigating directories, creating a new directory, accessing a directory, deleting a file
System	Command execution

4.2.1 Classification of Critical Calls

To systematically evaluate the security risks associated with critical operations, Joern's data-flow information is used to classify critical nodes based on the trustworthiness of their input sources. This classification helps distinguish operations that process untrusted user input from those that handle static or configuration-driven data, providing a clear framework for analyzing potential vulnerabilities and logging requirements. The sensitive function calls are divided into three categories:

Trusted, which operate on data sourced from fixed internal values (e.g., hardcoded or constant input), **Untrusted**, which operate on data from user-controlled input; and **Configuration**, which operate on data retrieved from configuration files or functions call that are under the control of the user, but whose content cannot be easily controlled by an attacker.

To separate the different categories, Joern's data-flow analysis is leveraged by performing a `reachableByDetailed` query, with user input nodes defined as sources and critical call nodes set as sinks. This analysis returns complete interprocedural data flow paths that capture how data originating from user inputs propagates to critical operations, including through variable assignments, function arguments, return values, and any intermediate data transformations. A critical call is classified as **untrusted** if any portion of its arguments can be traced back to the input node, whether through a direct assignment or a series of propagation steps. Distinguishing trusted from untrusted input is more complex in GUI applications, where programs rely extensively on event handling implemented through a callback mechanism. In this case, applications register event-handling functions, known as callbacks, and visual elements emit signals representing events like a button click or a key press. When a signal is emitted, the associated callback function is invoked. These functions further process user inputs and manage the transition of input data to the rest of the application code. In this case, if data for a critical call is sourced from a signal-connected callback function that does not handle any input nodes, its content can be considered trusted.

4.2.2 Log Placement

So far, the relevant nodes in the program dependency graph have been identified, and their data-flow relationships have been examined to narrow the analysis to critical nodes that process untrusted input. To address the three research questions, the relative position of the corresponding log statements needs to be computed. While the data flow analysis helps to identify if (and how) input data reaches critical calls, it does not capture the actual sequence of instructions executed along the data-flow path or their control relationship.

For instance, Figure 4.1 shows the existence of a data-dependency between the input at line 1 and the critical node at line 6, and between the input and the log statements at lines 4 and 8. It is however unclear whether the instruction associated with the log statements are *guaranteed* to be executed, either before or after, the critical call. Since the objective is to evaluate whether log statements are placed along the execution path leading to untrusted critical calls, the program elements that are necessarily encountered during execution must be identified. The analysis is carried out in two phases to separately detect log statements placed before and after the critical call.

Pre-Logs. To identify whether information is logged before the critical call, *dominator tree* is computed. The dominator tree provides a hierarchical representation of control-flow relationships that captures which program nodes always precede the execution of others. If a log node strictly dominates a critical call, this ensures that every path from the entry node to the critical call must pass through the log statement. Therefore, if a log statement belongs to the dominance tree (computed by using the `dominatedBy` Joern call), it can be concluded that the log is *always* executed before the call is performed. If it does not, however, it can only be concluded that there exists at least one execution path in which this is not the case. However, to verify this, the `cfgPrev` API is used to check whether at least one path exists in which the log statement precedes the critical call. For instance, in the following snippet:

```
1 if (condition) {
2    log_message(param);
3 }
4 critical_call(param);
```

the `log_message` call is not a dominator of the `critical_call`, but there is at least one control flow path that goes from the first to the second, showing that this is a case of **inconsistent** pre-log.

This method of computing the dominance tree and verifying the log placement on the control-flow graph is first applied at the intra-procedural level, within the function where the critical call is located. This is due to a limitation of Joern, which does not provide support for the computation of inter-procedural `dominatedBy` relationships. However, to capture realistic execution paths that originate from user input, the analysis must be extended across function boundaries. This is achieved by combining interprocedural data-flow analysis with dominance-based control analysis. Specifically, interprocedural data flow is used to trace the propagation of user-controlled input through the call graph, identifying the chain of functions that ultimately pass this input to the critical call. For each function along this path, two-phase analysis were reapplied: (1) the computation dominance tree rooted at the relevant call site to the next function, and (if failed) (2) the reachability validation of any log statements via control-flow traversal.

Consider the following critical call from GIMP:

```
1 g_file_delete(local_file, NULL, NULL);
```

Through data-flow analysis, we determine that the argument `local_file` originates from user input obtained via:

```
1 gtk_entry_get_text(GTK_ENTRY(entry));
```

This input is initially handled by the function `file_open_location_response`, which processes user-provided URIs. Then the data propagated through a sequence of func-

tion calls starting from — `file_open_location_response` to `file_open_with_proc_and_display`, and finally to `file_open_image`, where the critical call to `g_file_delete` is made. This interprocedural data-flow path enables the focus on dominance analysis on functions the input actually flows through, avoiding irrelevant call sites and enabling a realistic dominance tree from input to sink. For instance, while `file_open_image` is invoked by several other functions, only one (`file_open_with_proc_and_display`) is involved in the propagation of the input node. The dominance tree is therefore extended to `file_open_with_proc_and_display`, specifically checking whether any log statements dominate the call to `file_open_image` call. This process is applied recursively: since `file_open_with_proc_and_display` is itself called by `file_open_location_response`, that function is examined for log placement before its invocation of downstream functions. The combination of interprocedural data-flow analysis (to identify relevant function call chains) with intra-procedural dominance tree (to determine whether logging occurs along these chains) is applied at each step.

Post-Logs. For log statements placed after the untrusted critical call, the same method is applied in reverse. This time, the log node is treated as a critical call node (the end of the execution path), while the critical call is considered the input node. The goal is to determine whether any logs are placed after the execution of the critical call, either directly following it or within conditional branches that handle its outcome. Unlike the construction of the dominance tree for pre-execution analysis, the control-flow traversal in this phase is not performed, as a part of the goal is to identify where log statements are placed after the untrusted critical call. A data-flow query was first performed from the critical call to the log node using data flow analysis. If successful, this allowed the reconstruction of interprocedural data flow from the critical call to the log statements. The log statement was then analyzed to determine whether it dominates the critical call, thereby confirming if it is always executed afterward. If this condition was not met, the control flow was inspected to assess whether the log is conditionally executed, indicating that it runs only under specific post-operation conditions.

4.2.3 Summary

By performing a data-flow analysis we determine whether untrusted input is used as part of a critical operation, and whether (part of) that input also appears in a log statement (research question Q1). If so, we study the domination relationship, extended to a interprocedural setting, to tell whether the log statement always precedes or always follows the critical node (research question Q2). Finally, if no dominance relationship is detected, we perform a control-flow analysis to identify whether the log nodes are only conditionally executed in some cases but not others (thus answering question Q3).

4.3 Measurement of Logging Completeness and Behavior Across GTK Applications

Techniques presented in the previous section are applied in a tool called **BlindSpot**. Once configured with a set of regular expressions to identify input, logs, and sensitive nodes, BlindSpot then uses Joern to analyze the program dataflow graph and compute the set of critical calls. For each of them, it then relies on a combination of control-flow

and dominance analysis, as explained in Section 5.2, to locate the corresponding pre-execution and post-execution log statements, as well as to compute whether there exists an execution path that can avoid such calls. To analyze logging behavior with BlindSpot, a set of open-source, GTK-based desktop applications was selected, as they offer a consistent use of standardized functions to handle user input and perform critical operations. This consistency simplifies the systematic identification of input and critical call patterns across different projects – which is the input required by BlindSpot to perform its analysis. To conduct the measurement study in practice, applications were selected from a variety of categories, including image editors, document viewers, messaging clients, email clients, multimedia players, and file managers. All the applications that were selected may be used to process untrusted files or receive messages and commands over the network. This dataset allows to demostrate logging practices across different types of user interactions. Table 4.2 provides an overview of the different applications, including the version of the source code analyzed, the size (expressed as lines of code (LOC)), logging-specific lines of code (Log LOC), and their overall logging density.

Table 4.2: Overview of studied applications with lines of code (LOC), version, and logging density

Application	**Description**	**Version**	**LOC**	**Log LOC**	**Log Density (%)**
Pidgin	Messaging client	2.14.13	234,392	1,857	0.79
Evolution	Email client	3.56.0	482,029	756	0.16
Evince	Document reader	2.7.21	73,688	90	0.12
Darktable	Photo editor	5.0.0	318,400	2,916	0.92
GIMP	Image editor	3.0 (stable)	771,557	1,270	0.16
Chatty	Messaging app	0.27-beta5	34,384	273	0.79
Balsa	Lightweight mail client	2.6.4	93,939	409	0.44
Xreader	Document reader	4.2.6	65,115	135	0.20
Rhythmbox	Music player	3.4.8	106,343	355	0.33
Thunar	File manager	4.20pre1	72,651	86	0.12

Table 4.3: Call counts for file, URL, and system actions categorized by trust level and configuration across applications.

Application	**File**				**URL**				**System**			
	Total	Trusted	Config	Untrusted	Total	Trusted	Config	Untrusted	Total	Trusted	Config	Untrusted
Pidgin	133	25	101	7	12	8	2	2	6	2	1	3
Evolution	182	53	119	10	4	2	-	2	11	2	9	-
Evince	79	32	40	7	7	1	3	3	1	-	1	-
Darktable	193	91	88	14	1	1	-	-	2	-	-	2
GIMP	513	186	274	53	6	4	2	-	5	2	3	-
Chatty	40	17	21	2	4	-	4	-	3	2	-	1
Balsa	87	19	67	1	14	12	1	1	3	-	3	-

Continued on next page

Table 4.3 — *continued from previous page*

Application	File				URL				System			
	Total	Trusted	Config	Untrusted	Total	Trusted	Config	Untrusted	Total	Trusted	Config	Untrusted
Xreader	80	21	51	7	7	1	3	3	1	-	1	-
Rhythmbox	140	45	92	3	6	4	1	1	1	1	-	-
Thunar	77	37	38	2	1	1	-	-	2	-	2	-

Variations in the ratio of logging statements over the total source code result in noticeable differences in logging density across the studied applications. Overall, across the projects, it is observed that a minimum of 86 and a maximum of 2916 unique calls to log information – ranging from 1 to 9 log statements for each 1000 lines of code.

By performing a data-flow analysis, BlindSpot first breaks down these statements in the three different categories: Trusted, Config, and Untrusted (see Table 4.3). The numbers already show some interesting takeaways. For instance, the lack of logging for system-level actions is especially worrying. System command executions (rightmost section in Table 4.3) are highly sensitive from a forensic point of view, and prone to command injections in case the input is not properly sanitized. However, many applications either do not log them or log them only under specific outcomes. For example, *pidgin* includes three calls to `system(..)`, but none of them is associated to either pre- or post-execution logs. *Chatty*, in contrast, logs one post-execution outcome that is executed only if the execution succeeds. This makes untrusted system calls undetectable if they do not raise an error or trigger a success message, potentially introducing a forensic blind spot.

The tool then discards 'trusted' and 'config' operations and focuses its analysis to study the **124** (106 file-related, 12 URL, and 6 system commands) critical actions that processes 'untrusted' input. For those, the analysis reveals a significant variation in logging placement across applications, particularly when comparing pre-execution and post-execution logging. This inconsistency creates challenges for forensic traceability and security observability, especially in the context of actions resulting from user inputs.

Table 4.4: Logging behavior across file, URL, and system command actions in various applications.

App	File			URL			System		
	Count	Pre	Post	Count	Pre	Post	Count	Pre	Post
Pidgin	7	3	5	2	-	-	3	-	-
Evolution	10	-	1	2	-	-	-	-	-
Evince	7	-	-	3	-	-	-	-	-
Darktable	14	-	4	-	-	-	2	-	-
GIMP	53	-	18	-	-	-	-	-	-
Chatty	2	2	2	-	-	-	1	-	1
Balsa	1	-	-	1	1	-	-	-	-
Xreader	7	-	-	3	-	-	-	-	-
Rhythmbox	3	2	-	1	-	1	-	-	-
Thunar	2	-	-	-	-	-	-	-	-

Table 4.4 summarizes the logging behavior for untrusted calls across applications. For each case, the analysis includes the total number of untrusted calls and identifies whether log statements appear before or after the corresponding critical call. A dash ('–') indicates the absence of logging either in pre- or post-execution. For example, *Pidgin* performs seven file operations where the filename depend on untrusted input. Out of them, three have one or more corresponding pre-execution log statements, and five have post-execution logs. The fact that the sum is greater than seven signifies that some operations are associated with both a pre- and a post-execution log statement.

RQ1 - Log Presence. The answer to the first research question confirms the finding of study in Chapter 1 (refer to 3) regarding the general lack of log statements in desktop applications. While in the previous chapter it showed this to be the case in general, the tool found this malpractice to also affect critical actions. In fact, out of 124 critical actions in the dataset, only 37 (29.8%) have at least one associated log operation. Strangely, the percentage is slightly higher for file operations (31.5%) and lower (16.7%) for both URLs and system commands. The absence of logs capturing data and the execution context of untrusted critical calls limits the effectiveness of logs for security auditing and accountability. A key goal of security-relevant logging is to ensure traceability around operations involving untrusted input. However, the result of analysis reveals that such coverage is inconsistent at best and many of these calls are not accompanied by consistent logging. For instance, *gimp* contains 53 untrusted file operations, yet none are preceded by logs, and 35 of them are not associated to any log operation at all. This difference between the volume of untrusted critical calls and the presence of logs introduces serious observability gaps.

RQ2 - Log Placement. Blindspot emphasizes the imbalance between pre- and post-execution logging practice.

Only 8 of the 37 log statements associated to critical calls are placed before the action is executed – showing that developers prefer to log *after* the operation is completed, typically along with the result of the operation itself. This design choice aligns well with debugging and operational diagnostics, when post-execution logs can provide feedback on what the system actually did, but in case a critical operation results in an application crash or in a compromise (e.g., if an attacker can inject a command into the system operation), post-execution logs might be absent or unreliable. Only one application in the dataset (*Chatty*) logged both before and after critical actions, thus providing a more reliable forensic trail. However, most of the other applications failed to maintain both pre- and post-execution logging, resulting in limited visibility into the behavior of untrusted critical calls.

RQ3 - Gaps. The most interesting finding of BlindSpot is related to the conditional execution of log statements. The analysis shows that post-execution logging is very often conditional. In fact, even in the rare cases in which the application logs sensitive parameters, it does so within error-handling paths or conditionally to the success of the operation. As a result, a successful execution may not be logged at all, which leaves gaps in forensic reconstruction, especially if the critical action caused unintended system behavior.

Table 4.5 breaks down the pre- and post-execution logs based on whether they are part of conditional or unconditional branches. This emphasizes one of the most important results of this study: even when developers include a log statement to record the parameters of a critical call, these statements are very often conditional and are therefore executed only in certain paths. This is the case of four out of eight of the pre-execution

Table 4.5: Distribution of Conditional and Unconditional Logging for Pre- and Post-Execution

Application	Pre-Execution			Post-Execution		
	Total	Cond.	Uncond.	Total	Cond.	Uncond.
File						
Pidgin	3	2	1	5	5	0
Evolution	-	-	-	1	1	0
Darktable	-	-	-	4	3	1
GIMP	-	-	-	18	18	0
Chatty	2	2	0	2	2	0
Rhythmbox	2	0	2	-	-	-
URL						
Balsa	1	0	1	-	-	-
Rhythmbox	-	-	-	1	1	0
System Operations						
Chatty	-	-	-	1	1	0

logs, and for a stunning 31 out of 32 of the post-execution ones.

In summary, BlindSpot draws a dire picture for the applications in the dataset:

- Log entries exist for less than 30% of the critical actions.
- Pre-execution logs (which should be favored from a forensics perspective) are rarely used.
- Post-execution logs are almost always conditional on the result of the operation.
- In total, only 5 out of 124 critical operations (4%) are associated with an unconditional log entry.

This prevents building a clear and continuous trail of what occurred during the lifecycle of a critical operation, raising serious questions about the practicality of application logs in a forensic investigation.

4.3.1 Forensic Implications of Logging Inconsistencies

The divergence of results shown in Tables 4.3 and 4.4 reveals a critical gap in the reliability of log statements for security monitoring and forensic analysis.

Two representative cases from the dataset, Pidgin and GIMP, help illustrate this gap in more detail. Pidgin constructs a system command using untrusted input and executes it without any log statements, neither before nor after the execution. This command originates within a GTK callback function registered using:

```
1 g_signal_connect(G_OBJECT(combo_box), "drag_data_received",
    G_CALLBACK(theme_dnd_recv), (gpointer) type);
```

Within the corresponding handler, `theme_dnd_recv`, a filename is extracted from a dropped item, an external TAR archieve (e.g., a `.tar.gz` file) that user drags into the combo-box of the application to install a custom theme using `GtkSelectionData` and converted from a URI to a local path:

```
tmp = g_filename_from_uri(name, NULL, &converr);
```

This path is then embedded into a shell command:

```
command = g_strdup_printf("tar > /dev/null xzf %s -C %s", path_escaped,
  destdir_escaped);
system(command);
```

The application performs a system-level command constructed from user input, but no log statement is present to capture the initiation or result of the action. While this does not currently represent an exploitable vulnerability in isolation (the filename itself is restricted to a safe format by calling the `g_shell_quote` function) the unavailability of logs severely limits the ability to trace user actions or system behavior of the application and is overall just a bad security practice in case of a poor sanitization or an error in the code. This example illustrates a broader issue identified in the findings: the application executes three shell commands which are partially controlled by user input, yet none are captured in the logs. Any potentially malicious use of these unlogged commands, e.g., due to a bug in the sanitization process, would be invisible to analysts and leave no trace for a post-mortem analysis.

The next example focus on inconsistent logging practices by developers. This case examines four different points in the same application where the program saves data to a file. All four cases open the destination file, whose name is controlled by the user, by calling `g_fopen` with write-related modes (e.g., `"wb"`) and then proceed to write the file content. Even though the four operations are similar in nature and rely on the exact same API calls, it is observed that there inconsistent in logging behaviors. As shown in the table below, in two cases, there are post-execution log statements that are executed if the operation fails (i.e., if the file cannot be created), while in the other two, there are no log statements at all to capture the same failure scenario:

Call	Method	Log	Conditional?
`g_fopen(filename, "wb")`	`save_preset_response`	Post-Execution	Yes, only if operation fails
`g_fopen(filename, "wb")`	`file_response_callback`	Post-Execution	Yes, only if operation fails
`g_fopen(name, "wb")`	`dialog_save`	None	-
`g_fopen(filename, "wb")`	`ifsfile_save_response`	None	-

In this example, inconsistency in post-execution log placement shows that some methods record the outcome of an operation, while others performing the same operation with the same input source do not. As a result, one method may appear responsible for a failure or unwanted behavior, with the leaves of trace in a form of post-execution logs, while the other leaves no forensic trail. For example, in a case of unintended file modifications or overwrites triggered by a write operation, an analyst might find the cause of reasons of a failed file save in `save_preset_response`, but no trace of similar activity in `dialog_save` or `ifsfile_save_response`. This inconsistency makes it difficult to determine whether those operations succeeded, failed silently, or were never triggered at all. Due to the inconsistent placement of log statements after critical operations, it becomes difficult to trace the outcomes of identical actions. When the same input triggers the same method at different points in the program, the absence of consistent logging results in incomplete trace. This inconsistency complicates the process of post-analysis in the case when a failure occurs or in the worst case, when an unintended file modification happens silently.

4.4 Threat to Validity

This study has several limitation that may affect the findings. Although the approach and tool are generic, the experiments were focused on desktop applications, as GUI applications were less studied in the past and they present particular challenges from a static analysis point of view. As a result, the findings may not fully reflect the logging practices or constraints of other domains such as mobile apps, web services, or embedded systems.

Adapting BlindSpot to a different domain would require the analyst to define what the input sources, critical operations, and logging methods are for the new environment.

A second limitation of the work is that it is subject to the limits of static analysis. While this is also an advantage, as a static approach simplifies the deployment and it is easier to integrate into a development pipeline, future work could complement the solution with a more costly (but potentially more precise) dynamic tracing. This would allow to investigate runtime observability across different real-world execution scenarios. Furthermore, BlindSpot works on application that statically defined logs in their source code. This shows when in the applications that were studied, logging was implemented using hardcoded method, where we were able to detect and label log nodes clearly. The tool currently do not account for applications that generate logs dynamically through macro expansion or runtime code generation/instrumentation.

Although this study examines log presence and placement for forensic visibility, it does not consider the trade-offs of excessive logging. In practice, extensive logging may lead to performance overhead, increased I/O activity, and large log footprints.

BlindSpot identifies cases where logging is inconsistent, conditionally applied, or absent around critical operations, raising concerns about the reliability of logs as a source of evidence. In the context of forensic readiness in logs, the cost, volume and availability of logs may be unavoidable especially when the completeness and forensic visibility were take into account. However, to manage these trade-offs, practical deployments may include strategies such as log reduction, pruning, or log anonymization to address privacy concerns which also beyond the scope of this study.

Chapter 5

Reliability of Mobile Forensic Acquisition under Database Drift

Mobile applications have become a necessity platform for personal productivity, communication, and social networking. Most data on mobile devices is generated and stored within applications, where user activities like messaging, calling, and browsing are recorded in application databases. Thanks to the operating system native support, these databases are often implemented using SQLite and provide one of the richest sources of digital evidence available during forensic investigation. Their value is immense: Conversations can be constructed from message histories, movements can be mapped from location records, and online behaviour can be re-constructed from the browser artifact. The access to these precious pieces of evidence depends on the correct acquisition and interpretation of data records stored in the different databases that each application create and manage according to its own unique schema. However, updated versions of the applications are released at short intervals, sometimes weekly or even daily. Along with new features and bug fixes, these updates might also introduce changes in the database schema. Such changes, referred in this chapter as **schema drift**, can involve the addition of a new table or field, its removal, or the restructuring of some existing table's columns. To extract this information from SQL databases, existing forensic tools use relational queries that are tailored for a specific database schema. As schemas drift, forensic acquisition queries may fail or overlook newly introduced evidence data, increasing the risk of incomplete forensic results. This problem is not hypothetical. For example, the extraction queries described by Anglano et al. [ACG17] for Telegram have become nowadays (2025) obsolete, as the proposed queries to extract blocked users and messages no longer work due to table drops and table renaming. Furthermore, open-source forensic tools [abs25, sep25] regularly require manual human revisions, updates, and testing of the implemented queries to remain compatible with schema drift. This variability in how application schemas evolve, coupled with frequent software updates, raises important questions about the persistence and structural stability of database-stored evidence. Specifically, it challenges whether existing forensic acquisition processes can continue to retrieve complete and reliable evidence data as applications evolve. Despite the importance of application data, this variability and its implications for forensic acquisition remain largely unexplored by researchers. Therefore, this chapter investigates how the changes of database structures influences the persistence, completeness, and repeatability of forensic extractions, with a focus on schema drift across multiple application versions.

5.1 Motivation

The forensic value of application databases is not only determined by the richness of the data they store, but also by the stability of the structures that govern how such data can be retrieved. While the growing dependence of investigations on these databases has already been highlighted, this section focuses on the specific factors that drive schema drift.

A first driver is the release cadence of modern mobile applications. Many popular apps do not only perform cosmetic improvements: in new versions they frequently introduce new features that can alter the underlying storage structures. A second driver lies in the mismatch between how forensic tools evolve and how applications evolve. Commercial and open-source tools are maintained by human developers who must monitor app releases, inspect database layouts, and rewrite SQL queries when incompatibilities arise. As shown in the case of open-source projects, like the Whatsapp Msgstore Viewer [abs25] and IPED [sep25], the frequency of application updates often overtake the capacity of maintainers, resulting in tools that quickly become outdated. This creates a dangerous situation in which investigators may unknowingly rely on queries that execute successfully but no longer capture all relevant artifacts. Finally, the reliability of digital evidence is a legal as well as a technical concern. Silent query degrading, where fields such as deleted messages, attachment metadata, or group membership are no longer retrieved, risk to produce partial reconstructions of a user activity. Such gaps are difficult to detect in practice and can undermine both investigative completeness and the admissibility of evidence in court. Schema drift therefore represents not just a technical inconvenience, but a direct threat to the repeatability and defensibility of forensic practice.

These reasons motivate the central research questions of this work:

- **Q1: How often are popular Android applications updated, and how are releases distributed across major, minor, and patch versions?** By studying update intervals and the distribution of release types, the level of activity in application updates was measured. Frequent update suggest that core features of application might be redefined or redesigned, and this may suggest that the database where the evidence reside might be modified as well.

- **Q2: To what extent do these updates alter the underlying database schemas?** In particular, how often are tables renamed or dropped, columns added or restructured, and schema layouts changed across versions? Such changes can break existing queries, result in incomplete data extractions, or introduce silent degradation in forensic workflows. This is particularly problematic when investigators unknowingly rely on outdated SQL queries, potentially leading to missed evidence or partial reconstructions.

- **Q3: How do schema changes impact the robustness of forensic acquisition queries?** When database schemas are modified, the change can affect the ability to extract evidence from the database. Particularly, if queries must be updated to recover forensic casework information after a version update, does schema evolution preserve or challenge the reliability and completeness of evidence collection?

By addressing these questions, this paper provides a longitudinal perspective on how Android application evolution shapes the durability of forensic workflows. Understanding

schema drift at scale is a necessary step toward the development of adaptive, drift-aware tools capable of ensuring evidential integrity despite the rapid pace of mobile application change.

5.2 Methodology

This approach of understanding the stability and reliability of SQL-based forensic extraction across different versions of popular applications starts with the process of selecting 20 widely used applications that are widely supported by commercial forensic extraction tools such as Cellebrite [Cel18, Cel20], MSAB [MSA14, MSA22], and Belkasoft [Bel18, Bel25] and reflect realistic forensic challenges across multiple domains.

For each application in the dataset, a preliminary analysis was performed to identify relevant databases (Step 1), followed by execution in a controlled environment and interaction to populate the databases with realistic data (Step 2). The next steps involved systematically extracting the SQL databases produced by each application version (Step 3), comparing their schemas to identify changes (Step 4), and finally assessing how schema evolution affected the recovery of forensic artifacts (Step 5).

5.2.1 Environment Setup

In Android systems, each installed application is assigned a private directory (only accessible with root privileges) that it can use to store persistent internal data, including databases, configuration files, shared preferences, and cached content. To access this information, the experiments were conducted in a rooted Android 16.0 (API level 34) `x86_64` emulator. Some application versions, however, were incompatible with this Android release or enforced compatibility checks that prevented execution. In such cases, older emulator images were used, specifically Android 11.0 (API level 30) or Android 8.1 (API level 27), which allowed the restrictions to be circumvented and enabled execution of the applications and collection of their databases.

Table 5.1: Collected Databases and Required Interactions per Application Version

Application	DB Creation/Fill Trigger Actions	Extracted DBs	DB Missing Reason
Communication Applications			
WhatsApp	Load messages	5/16	Forced update
Kakao Talk	Load messages	15/16	Forced update
Telegram	Load messages	16/16	
Viber	Login + startup	16/16	
LINE	Load messages	16/16	
Google Chat	Send a message	16/16	
Discord	Login + message	14/16	DBs absent in previous versions
GroupMe	Login + message	16/16	
Truecaller	Login + startup	16/16	
Browser Applications			
Firefox	Browse	16/16	
DuckDuckGo	Browse	15/16	Forced update
Brave	Browse	16/16	
Tor Browser	Browse	16/16	
Opera	Browse	16/16	
Social Media Applications			
Twitter/X	Post + message	16/16	
Instagram	Login + message	15/16	Forced update
TikTok	Login + message	16/16	
Note-taking Applications			
Evernote	Login, create notes, organize event	7/16	Forced update
Location/Navigation Applications			
Google Maps	Query + navigate	16/16	
Waze	Query + navigate	16/16	

5.2.2 Applications Download (Step 1)

To study the evolution of database schemas across different applications, the approach requires collecting multiple versions of each application. For each application, the APK was downloaded from APKMirror [APK25], a publicly available repository that archives application packages across different release versions. All monthly releases between May 2024 and May 2025 were retrieved. In addition, to capture longer-term changes, one version from six months prior to May 2024 and one version per year for the two preceding years were included.

This allowed us to track schema drift both in the short term (month-to-month changes) and in the long term (yearly evolution). In months with multiple updates, the earliest available version was selected to maintain temporal consistency across the dataset. We manage to collect up to 16 APK versions per application, with total of 320 APKs for the entire dataset. This structured versioning strategy enables a uniform basis for longitudinal schema comparison and helps ensure that any observed changes in the database structures were resulted from the version evolution and not because of inconsistencies in sampling.

Table 5.1 summarizes the applications under studied, along with their categories and description. For a visual overview of the timeline, versions and application update cate-

gories included in this study, see this link[1].

5.2.3 Application Installation and Interaction (Step 2)

Each application version was installed on the emulator according to its update timeline, and a predefined sequence of manual interactions was carried out to both trigger database creation and populate it with forensically relevant data. Repeating the same interaction sequence across all versions ensured that the resulting datasets were directly comparable. This step was particularly important for applications like WhatsApp and Telegram, which only instantiate their databases after user activity (such as sending messages or initiating conversations) has occurred. The interaction steps executed prior to data extraction are detailed in Table 5.1.

5.2.4 Database Extraction (Step 3)

Next, each application's private data directory were copied from the emulator to the analysis machine to extract the SQL databases generated by each app. However, not all databases containing forensic artifacts were recoverable. In fact, in some rare cases the application refused to start because it required a mandatory update. Despite numerous attempts with compatible emulator images, date/time changes, offline mode, and dynamic instrumentation (Frida) to trigger the required interactions, these versions refused to create the local database until a more recent version was installed.

The study concentrated on databases containing forensically relevant evidence, such as message histories, posting activity, location queries, and event records. These databases were then examined to detect instances of schema drift. Table 5.1 summarizes the number of databases successfully collected for each application and highlights the challenges that limited recovery.

5.2.5 Schema Drift Analysis (Step 4)

To examine how database schemas changed across versions, we relied on the `sqldiff` utility provided with SQLite. This tool outputs DB schema differences as SQL statements (e.g., `CREATE TABLE`, `DROP TABLE`, `ALTER TABLE`), enabling a global structural comparison between databases. To capture finer-grained changes, this process was complemented with metadata inspection using `PRAGMA table_info`, which enabled analysis of column definitions within tables. By combining these approaches, schema drift was classified into table creation, deletion, restructuring, and column-level modifications such as column removal and addition. Tracking these changes provided a longitudinal view of how evidence storage evolved. Newly introduced tables could expand the scope of forensic artifacts, whereas dropped or restructured tables could result in data loss, relocation, or reinterpretation. This analysis formed the basis for evaluating the stability of database structures and their implications for forensic extraction.

[1]Timeline available at: https://anonymous.4open.science/r/app-version-timelines-C6D3/

Table 5.2: Forensic evidence extracted per application

Application	Forensic Evidence
Communication Applications	
WhatsApp	DM; GM; AM; Call Logs; Group Call Logs; Deleted Messages
Kakao Talk	DM; GM; AM; Deleted Messages
Telegram	DM; GM; AM
Viber	DM; GM; Call Logs
LINE	DM; Group Info; AM; Call Logs
Google Chat	DM; GM; Search History; Deleted Messages
Discord	DM
GroupMe	GM; AM; Event Conversations; Deleted Messages
Truecaller	Call Logs; Message Content; AM; Favorite Contacts
Browser Applications	
Firefox	Search Queries; Bookmarks; Recently Closed Tabs
DuckDuckGo	DF; Bookmarks
Brave	Visited Pages; DF; Recently Closed Tabs
Tor Browser	Open Tabs; Bookmarks; Recently Closed Tabs
Opera	Visited Pages; DF; Location Browsing; Search titles and URLs
Social Media Applications	
Twitter/X	DM; Bookmarked Posts
Instagram	DM; Followers List; Recent Searches
TikTok	DM; User Interactions; Deleted Messages
Note-taking Applications	
Evernote	Deleted Notes; All Notes; Reminder List; Calendar Events; Tasks
Location/Navigation Applications	
Google Maps	Current Location History
Waze	List of Favorite Locations; Search Queries; Navigation History

DM Direct Messages
GM Group Messages
AM Attachment Metadata
DF Downloaded Files Metadata

5.2.6 Query Robustness Evaluation (Step 5)

To evaluate the resilience of SQL queries on the application's databases, this step examined how schema drift influenced their effectiveness in extracting evidence across different application versions. Our objective was to estimate how often forensic tools must be updated to remain accurate. For this purpose, a set of SQL queries is designed for each application, targeting the database schema of versions released in May 2024- a point chosen as the approximate midpoint of the dataset. Query design was guided by the forensic features listed in Table 5.2, which capture evidence types of practical relevance to investigators, such as messaging records, posting activity, event logs, and location history. Once developed on the reference schema, these queries were executed across all earlier and later versions. This approach enabled us to assess both backward compatibility with older releases and forward robustness against newer updates. By recording query failures and their causes, this step managed to quantified the impact of schema drift on the reliability of forensic data extraction.

5.3 Results

5.3.1 Update Frequency and Release Patterns

Table 5.3 highlights the substantial variation observed in the update frequency across application categories. Communication and social platforms, including TikTok, WhatsApp, Telegram, and Instagram, update most frequently, with median intervals ranging from one to eight days. These applications also release both minor and major updates at a rapid pace, reflecting a continuous user-driven improvements. This observation is consistent with the high schema drift that will be discussed in Section 5.3.2, as shorter release cycles with significant version changes increase the likelihood of modifications in database evidence storage.

In contrast, browser and navigation applications exhibit more moderate update cycles, typically ranging from 7 to 24 days. Browsers such as Firefox, DuckDuckGo, Tor Browser, and Opera primarily release updates as scheduled minor or major versions rather than frequent patches. Their development strategy emphasizes bundled releases, in which security fixes, bug resolutions, and feature enhancements are integrated into larger, less frequent updates.

Finally, note-taking applications, such as Evernote, update approximately every five days. However, these updates are limited to patches and minor revisions, with no major redesigns observed.

Table 5.3: Application Update Intervals and Types, Grouped by Category

Application	Update Interval (days)	Major	Minor	Patch
Communication Applications				
WhatsApp	2	4	11	–
Kakao Talk	12	4	8	3
Telegram	8	3	11	1
Viber	8	7	8	–
LINE	8	4	11	–
Google Chat	3	13	2	–
Discord	3	15	–	–
GroupMe	20	2	13	–
Truecaller	8	2	13	–
Browser Applications				
Firefox	10	15	–	–
DuckDuckGo	7	–	15	–
Brave	3	–	13	2
Tor Browser	22	4	11	–
Opera	8	10	5	–
Social Media Applications				
Twitter/X	3	2	13	–
Instagram	5	15	–	–
TikTok	1	8	7	–
Note-taking Applications				
Evernote	5	–	13	2
Location/Navigation Applications				
Google Maps	3	2	13	–
Waze	24	1	13	1

5.3.2 Schema Drift

A longitudinal analysis of 16 versions for each application in the dataset reveals that database schemas evolved over time at an application-dependent rate. Table 5.4 summarizes the databases that been examined to capture this evolution, according to the criteria presented in Section 5.2.

As shown in the table, several applications exhibited substantial schema drift, including table restructuring through the addition or removal of columns, as well as the creation and deletion of entire tables. In contrast, some databases maintained stable structures, showing no observable changes across versions. This study categorize the applications into three groups based on the extent of schema drift.

High-drift applications – Applications such as WhatsApp, KakaoTalk, Truecaller, Viber, LINE, Google Chat, TikTok, and GroupMe exhibited substantial schema changes across application updates. Most of these belong to the communication category, where databases evolve in tandem with the introduction of new features. Messaging applications, in particular, frequently expand functionality to support richer interactions, adding capabilities such as group messaging, community announcements, one-time-view attachments, and deleted messages. Also notable from the query coverage analysis (see Table 5.5, Telegram row of 'Deleted Message' Query Type and Notes) is

that, for now, only Telegram handles deleted messages by physically dropping the corresponding database rows. In contrast, other messaging applications that allow message deletion, such as WhatsApp, Instagram, Google Chat, TikTok, and GroupMe remove only the cell data containing the message text while retaining related metadata such as the timestamp, sender, and receiver. This approach in handling acquisition of deleted evidence may change in the other messaging applications if they want to adopt Telegram's privacy-oriented feature. When that happens, proving that a message existed must rely on version-aware techniques rather than current direct content reads.

As shown in Table 5.3, new features are often introduced within short release intervals, requiring developers to add new tables or columns to store the associated data. As a result, forensic acquisition queries designed for previous versions may miss newly created data evidence from the database. If the acquisition queries are not updated, reflecting the schema drift, investigator may risk overlooking this available data evidence.

Moderate-drift applications – These includes applications such as Telegram, X (Twitter), Waze, one database of DuckDuckGo, Brave, Instagram, and two of the databases in Opera. While these applications did not experience extensive changes, they still showed a noticeable structural evolution.

Within this category, communication and social platforms displayed more frequent schema changes than non-communication applications. In contrast, applications not primarily focused on communication or social interaction, that probably does not strictly depends on new user eye-capturing features, such as Waze, DuckDuckGo, Brave, and Opera, demonstrated only minor schema drift, typically occurring once a year or approximately every six months.

Low-drift applications – No schema drift was observed in the database of Evernote, Google Maps, and Discord, as well as in one DuckDuckGo database, a Tor Browser database, a LINE database, two Instagram databases, and an Opera database. This chapter observed that the core database structures that store essential artifacts, such as tab history, downloaded data, or search queries in browsers, and searching location, have remained constant and have not required structural modification, even as these applications continue releasing updates. As a result, there are no drift in this category of applications.

Drift Direction and Nature of Changes – With respect to table and column additions or removals, the analysis shows that schema drift across all studied applications was predominantly additive, with far fewer instances of table or column deletion. This trend likely reflects a common development practice: preserving backward compatibility for existing user data while extending database functionality.

When removals did occur, they were limited to a small subset of applications. Table drops were observed in Telegram, Waze, Viber, LINE, TikTok, Opera, DuckDuckGo, and Brave, ranging from a single table removal to a maximum of three in a single update. However, three-table drops were exceptionally rare, occurring only twice across all studied versions, one instance even within a minor update. Two-table drops occurred instead three times, all associated with major releases.

For example, in TikTok a major update removed the tables `IM_RES_CACHE` and `USER_EXTRA`. The first table dropped stored media cache information, while the second one contained metadata about user-interaction features, such as logging interaction prompts between profiles. These tables not only reflects a change in feature design but also dropped the availability of related forensic artifacts, including cached media traces

and records of user-to-user interaction events.

Cases of columns dropped were observed in X (Twitter), Truecaller, Viber, Brave, GroupMe, Opera, and WhatsApp, with instances ranging from a single dropped column to a maximum of two. For example, X dropped the column `users.has_nft_avatar`; Truecaller dropped `msg_im_quick_actions.action_value`; Viber dropped `conversations.to_number` as well as `hidden_gems.countries` and `hidden_gems.monetized_phrase`; GroupMe dropped `messages.has_image_url` and `chats.is_muted`; and WhatsApp dropped `optimised_delivery_info.msg_timestamp`.

These table and column drops were not seen in the monthly incremental updates but mainly appeared in the yearly or six-month snapshots. This pattern suggests that destructive schema changes are more likely to occur during major feature overhauls or redesigns rather than routine updates. The removal of tables or columns containing forensic evidence has direct implications for forensic practice, as it necessitates adapting tools and queries to ensure consistent data collection across different application versions.

More details on the specific versions and databases affected by tables and columns removals are provided in Tables on the dataset repository [Ano25].

Table 5.4: Schema drift across applications and time ranges (Nov 21 – May 25)

Application	Database	11/21–11/22	11/22–11/23	11/23–05/24	05/24–06/24	06/24–07/24	07/24–08/24	08/24–09/24	09/24–10/24	10/24–11/24	11/24–12/24	12/24–01/25	01/25–02/25	02/25–03/25	03/25–04/25	04/25–05/25
WhatsApp	msgstore.db												4/0/3 (6/0)	3/0/8 (26/0)	6/0/10 (34/0)	4/0/6 (11/1)
KakaoTalk	KakaoTalk.db								0/0/1 (8/0)						0/0/1 (2/0)	
	KakaoTalk2.db						1/0/2 (3/0)		1/0/0 (0/0)			1/0/0 (0/0)		1/0/0 (0/0)	0/0/1 (2/0)	
Telegram	cache4.db	14/0/4 (16/0)	12/1/6 (14/0)	7/1/4 (10/0)	2/0/0 (0/0)				2/0/0 (0/0)	0/0/1 (2/0)	0/0/2 (6/0)					
Viber	viber_messages	3/0/7 (14/0)	5/0/11 (12/0)	2/1/3 (7/1)	0/0/2 (4/0)		1/0/1 (2/0)	0/0/3 (8/0)		0/0/2 (4/0)		0/0/1 (2/0)	0/0/1 (4/0)		3/0/0 (0/0)	1/0/1 (0/2)
LINE	naver_line	0/0/2 (13/14)	0/2/2 (6/0)	0/0/1 (2/0)	2/1/2 (4/0)							0/0/2 (4/0)				0/0/1 (4/0)
Google Chat	dynamite.db	0/0/7 (32/0)	0/0/5 (26/0)	0/0/4 (18/0)		0/0/2 (4/0)	0/0/1 (4/0)						0/0/1 (2/0)	0/0/1 (4/0)	0/0/2 (16/0)	0/0/1 (2/0)
GroupMe	groupme.db	2/0/4 (12/1)	1/0/4 (12/0)	1/0/2 (4/0)		0/0/1 (4/0)				1/0/2 (0/2)		0/0/2 (14/0)	0/0/1 (0/0)			1/0/2 (3/0)
Truecaller	tc.db	3/0/3 (12/0)	2/0/11 (29/1)	0/0/3 (4/0)			0/0/1 (1/0)			0/0/1 (8/0)	0/1/0 (0/0)	0/5/2 (3/0)		0/0/2 (3/0)		0/0/1 (2/0)
DuckDuckGo	app.db	2/0/0 (0/0)	2/3/1 (2/0)	2/0/0 (0/0)						1/0/0 (0/0)	0/0/1 (2/0)			1/0/0 (0/0)	0/1/0 (0/0)	1/0/0 (0/0)
Brave	History	2/0/6 (48/1)	2/2/5 (14/0)	0/0/1 (2/0)												
Tor Browser	places.sqlite		0/0/3 (6/0)													
Opera	History	4/0/4 (30/1)	2/1/5 (14/0)	1/1/1 (4/0)												
	newsfeed.db	0/3/1 (2/0)				0/0/1 (2/0)				0/0/1 (4/0)				0/0/1 (2/0)		
Twitter/X	0-66.db			0/0/4 (9/1)						0/0/1 (1/0)					0/0/1 (4/0)	0/0/1 (0/1)
Instagram	direct.db			1/0/0 (0/0)												0/0/1 (3/0)
TikTok	{account_id}_im.db	0/0/4 (9/0)		0/0/3 (6/0)			3/0/2 (6/0)			0/0/1 (2/0)			0/0/1 (1/0)	0/0/1 (2/0)	0/0/1 (1/0)	
	db_im_xx	3/2/1 (10/0)	0/0/1 (3/0)									0/1/0 (0/0)				
Waze	user.db		0/0/1 (2/0)	0/1/0 (0/0)	0/0/1 (2/0)											

Values indicate database schema drift over time: new / dropped / restructured (columns added / columns removed).

We have studied also the following DBs that however do not show any modification along the timeline: LINE: `call_history`, Discord: `{account_id}/a`, Firefox: `places.sqlite`, DuckDuckGo: `downloads.db`, Tor Browser: `tab_collections`, Opera: `reading.db`, Instagram: `ranked_user_{account_id}` and `recent_searches.db`, Evernote: `UDB-{account_id}+RemoteGraph.sql`, GoogleMaps `gmm_sync.db`.

5.4 Stability of Forensic Extraction Under Drift

After categorizing the types of schema drift observed across application versions, the next step is to evaluate the stability of forensic data extraction in the presence of such drift. The objective is to determine how robust forensic queries remain when applications introduce schema changes and whether extraction tools can continue to function without modification.

To this end, a sequence of SQL queries were constructed targeting the forensic features defined in Table 5.2 within each application's database. For example, in Telegram, direct message information was extracted, including sender and receiver identifiers, timestamps, and, when available, geolocation data. All queries were initially developed for the May 2024 version, which served as the baseline.

The baseline queries were subsequently applied across all other collected versions to evaluate extraction stability under both forward drift (monthly updates from May 2024 to May 2025) and backward drift (yearly and six-month snapshots preceding May 2024, specifically November 2021, November 2022, and November 2023). This methodology made it possible to assess not only whether queries continued to execute successfully, but also how the scope and content of available evidence changed over time.

For backward drift, any query failures were addressed by rewriting the queries to restore functionality, with the results then compared to the May 2024 baseline. This comparison revealed whether earlier versions provided equivalent evidence, reduced evidence, or, in some cases, additional evidence.

For forward drift, the evaluation considered whether the baseline queries remained valid or required modification, and whether the extracted evidence differed from that of May 2024. This process made it possible to determine whether schema evolution over time led to an expansion, reduction, or consistency in forensic evidence.

Table 5.5: Query Coverage – Communication Applications

Application	**Query Type**	**Ver. Matched**	**Notes**
Telegram	Direct Messages	15/16	1 update (schema change)
	Group Messages	13/16	2 updates (schema change)
	Deleted Message	–	No row available
LINE	Direct Messages	4/16	1 update (schema change)
	Attachment Data	4/16	1 update (schema change)
Discord	Direct Message	14/16	No DB in later versions
GroupMe	Events/Conversations	13/16	1 update (schema change)
Truecaller	Call Logs	15/16	1 update (schema change)
	Message Content	14/16	1 update (schema change)
	Attachment Data	14/16	1 update (schema change)

Table 5.6: Query Coverage – Social Media Applications

Application	Query Type	Ver. Matched	Notes
Twitter/X	Direct Messages (DM)	14/16	2 updates (database name change)
	Bookmarked Posts	14/16	2 updates (database name change)
Instagram	Deleted Message	–	No row available

The results of the experiments show that messaging applications frequently experienced query failures due to schema changes, as summarized in Table 5.5. For instance, Telegram, LINE, GroupMe, and Truecaller all required at least one query update across the 16 versions.

In contrast, non-communication applications, including browsers, note-taking tools, and navigation apps, showed no query failures. This finding aligns with their classification in the moderate- and low-drift categories: applications for which the database schemas changed more often over time were also more likely to require adjustments in the forensic extraction queries.

Social media applications exhibited mixed levels of extraction stability. For example, Instagram and TikTok required no modifications across all collected versions, despite frequent minor schema changes. This indicates that the database structures storing forensic-relevant data remained unchanged.

Conversely, Twitter/X proved more fragile: queries targeting direct messages and bookmarks failed in 2/16 versions due to database name changes, as shown in Table 5.6. Although these changes were relatively minor, they were sufficient to disrupt otherwise valid queries, necessitating updates to maintain forensic support.

Table 5.7 resume for all the applications and versions when forensic queries required modifications to remain functional. A clear pattern emerges: most query failures occurred during backward drift, particularly in the yearly and six-month snapshots preceding May 2024.

Forward drift testing, covering monthly updates from May 2024 to May 2025, showed far fewer failures, with the majority of queries remaining operational across consecutive versions. An exception was observed in LINE, where queries targeting direct messages and attachment data failed from June 2024 onward. This failure resulted from a table drop in that update, which removed the `contacts` table containing essential message and attachment information.

Overall, these findings emphasize the importance of designing forensic support tools that balance two strategies: maintaining resilience against routine incremental updates while remaining flexible enough to handle major schema changes that may occur unexpectedly during both backward and forward evolution.

5.4.1 Cases Study: Query Update and Fallback

The next step is to examine how query updates and fallback strategies influence the richness and completeness of evidential data. Schema changes do not always cause queries

Table 5.7: Query Adaptations Across Applications and Timeline Versions

Application	Query Type	11/21	11/22	11/23	05/24	06/24	07/24	08/24	09/24	10/24	11/24	12/24	01/25	02/25	03/25	04/25	05/25
Telegram	Direct Messages	•	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Group Messages	★	•	•	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Twitter/X	Direct Message	★	•	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Bookmarked Post	★	•	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Truecaller	Call Logs	•	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Message Content	•	•	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Attachment Data	•	•	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LINE	Direct Messages	✓	✓	✓	✓	•	•	•	•	•	•	•	•	•	•	•	•
	Attachment Data	✓	✓	✓	✓	•	•	•	•	•	•	•	•	•	•	•	•
Discord	Direct Message	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
GroupMe	Events Conversation	•	•	•	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

✓ = query (built at 05/24) works unchanged on this version; • = query does not work and requires adaptation; ★ = query was updated at this version to handle a schema change (new baseline); ✗ = not applicable / no query possible.

to completely fail, but they can return results which does not contain newly added fields that can contain relevant evidence. This highlight that forensic tools must do more than simply maintain functional queries, they must also monitor schema modifications that could introduce new sources of evidence or, if overlooked, result in data loss.

By analyzing examples of query adaptation, both the risks of reduced evidential value and the opportunities to capture additional artifacts as databases evolve across version updates are illustrated.

Query Update

Consider the messaging application LINE, the baseline SQL query developed for the May 2024 version, shown in query 5.1a, allows to extract direct messages by joining the `chat`, `chat_member`, and `contacts` tables. However, following the June 2024 app update (version 14.4.5), this query start to fail because the `contacts` table, which previously stored user information, was removed from the schema and its usage of on more recent application version (forward query adaptation) result impossible.

A revised query is then required (Figure 5.1b). Although the updated query still retrieves key elements such as the last message text, message owner, and identifiers, the removal of the `contacts` table prevents the extraction of contact names. This reduces

```
SELECT c.last_message,
       c.last_created_time,
       c.owner_mid,
       con.name
FROM chat c
LEFT JOIN chat_member cm
   ON c.owner_mid = cm.mid
LEFT JOIN contacts con
   ON cm.mid = con.m_id;
```

(a) Baseline query valid for 11/2021 to 05/2024, used to retrieve the last message, creation time, owner, and contact name.

```
SELECT c.last_message,
       c.last_created_time,
       c.owner_mid,
       ch.from_mid,
       cm.mid,
       cm.created_time
FROM chat_history ch
LEFT JOIN chat c
   ON c.chat_id = ch.chat_id
LEFT JOIN chat_member cm;
```

(b) Updated query valid from 06/2024 to 05/2025: the `contacts` table was dropped and replaced by `chat_history`, allowing extraction of message and participant data but no longer providing contact names.

Figure 5.1: LINE direct message acquisition query adaptation due to schema drift.

the amount of directly linkable identity information available in the results. For forensic acquisition, this illustrates that schema changes may not entirely remove evidence, but can limit the range of data points that can be retrieved for analysis.

Query Fallback

The following illustrates how application databases evolved to include richer data over time, introducing more detailed structures and increasing the amount of extractable evidence. As shown in the queries adaptation in query 5.2, schema drift in Telegram required progressive fallbacks to support group message extraction in older versions. The baseline query for May 2024-2025 (refer to Figure 5.2a) relied on `messages_topics` and `topics`, which provided both message content and contextual fields such as group labels and edit timestamps. However, in the database schemas from November 2022 and November 2023, the baseline query is no longer functional, as it fails to return the same results as the acquisition query on the baseline version. This is because the `edit_date` field did not yet exist in those versions of the application. In this version of the application's database, a forensic investigator must adjust the baseline query to extract only the message data, as the date of group creation or modified is missing (Refer to Figure 5.2b). The queries designed for the November 2022 and November 2023 versions were executed on the Telegram database from November 2021. These queries failed with

```sql
SELECT
    m.date,
    t.data AS topic_data,
  m.data AS message_data,
    t.edit_date,
    m.uid,
    u.name,
    u2.phone
FROM messages_topics m
LEFT JOIN topics t
    ON t.topic_id = m.topic_id
LEFT JOIN users u
    ON u.uid = m.uid
LEFT JOIN user_phones_v7
     u2
    ON u.uid = u2.key;
```

(a) Baseline query valid from 05/2024 to 05/2025, extracting message content, topic data, edit date, user ID, name, and phone number.

```sql
SELECT
    m.date,
    t.data AS topic_name,
    m.data AS message_data,
    m.uid,
    u.name,
    u2.phone
FROM messages_topics m
LEFT JOIN topics t
    ON t.topic_id = m.topic_id
LEFT JOIN users u
    ON u.uid = m.uid
LEFT JOIN user_phones_v7
     u2
    ON u.uid = u2.key;
```

(b) Fallback query valid from 11/2022 to 11/2023, extracting required data but losing `edit_date`, which records the group creation date.

```sql
SELECT
    m.date,
    t.data AS message_data,
    m.uid,
    u.name,
    u2.phone
FROM messages_v2 m
LEFT JOIN
     channel_users_v2 t
    ON t.uid = m.uid
LEFT JOIN users u
    ON u.uid = m.uid
LEFT JOIN user_phones_v7
     u2
    ON u.uid = u2.key;
```

(c) Fallback query valid for 11/2021. Extracting data from messages_v2 + channel_users_v2 data format, with no group name data entirely.

Figure 5.2: Telegram group messages query adaptations due to schema drift

missing `messages_topic` and `topics` errors. Examination of the database schema revealed that the table used for storing group message data was not yet present in the November 2021 version.

On top of that, if a forensic investigator is unaware of schema drift as the application evolves, newly introduced columns that may contain important evidence can be overlooked. In such cases, existing queries still function but suffer from unnoticed query degradation. Take acquisition query presented in 5.2b as an example. If the same query is applied to extract group message data in 05/2025, it will still execute successfully but does not extract another possible evidence that is important for forensic that is timestamp. This case shown that if the forensic investigator or tool developer is unaware of new columns introduced through schema drift during an application update, the outdated acquisition may miss newly available data.

In this case study, the schema drift shows the need for practitioners to monitor newly introduced features, as in this case where schema drift in the form of additional columns provides more data for forensic acquisition.

Chapter 6

Conclusion

This thesis examined the reliability and readiness of application-generated data as a source of digital forensic evidence, focusing on two major evidence types: application logs and mobile databases. Through three complementary studies, it explored how the design and placement of logs, as well as the evolution of database schemas, influence the availability, consistency, and completeness of digital evidence.

6.1 Summary of Findings

The first study examined the readiness of application logs for security. While standards and guidelines exist to define what should be logged, how logs should be stored, and where they should be placed, most of these recommendations are still oriented toward debugging rather than security. To frame the evaluation, five forensic tasks were identified including activity timeline, event correlation, execution partitioning, misuse detection, and attack detection. Then data from logs that required to support each task was outlined, such as the availability of timestamps to reconstruct timelines, unique identifiers to correlate events and partition executions, the logging of user actions to detect misuse, and the availability of log data during exploitation attempts. An analysis of 60 popular desktop applications showed that logs are inadequate for forensic tasks. 29 out of 60, lacked of timestamps data, which made it difficult to reconstruct the timeline of events. In more than half of the applications, the log entries consisted mainly of constant text strings without identifiers such as user, network, or file data, limiting their value for event correlation. A further 23 applications did not record unique identifiers when new execution units began, complicating the reconstruction of processes and the identification of objects involved. The logging of user actions was also inconsistent, reducing the ability to detect patterns of misuse across different applications. Further, 35 applications contained no exception logs at all, while only 5 provided detailed exception data. Finally, in tests where past exploitation scenarios were recreated, successful attacks frequently generated no relevant log entries, and when such data was produced, it was often not available by default.

The second study presents the lack of logs surrounding critical calls, where actions from user input through to the critical operation and beyond are often left unrecorded. The availability of logs around such calls is essential to capture data in cases where critical operations can resulted in crashes, compromises, or unexpected shutdowns. To reasons

about all execution possibilities from untrusted inputs, a tool named BlindSpot was developed on top of Joern. BlindSpot extends Joern's capabilities by reconstructing realistic execution flows and capturing both pre- and post-execution paths, enabling detailed assessment of the presence or absence of logs around sensitive operations. Its structural analysis makes it possible to identify log statements that are guaranteed to execute along specific control paths, thereby providing a precise view of logging coverage for high-risk operations. The study analyzed ten open-source desktop applications and identified 124 untrusted critical operations. Only 37 of these (29.8%) had an associated log statement, with even lower coverage for system command and URL operations, which fell below 17%. Logging was most often performed after execution, with only 8 of the 37 logged actions being preceded by a log statement. Most of these logs, both before and after execution, were conditionally executed. In total, only 5 out of 124 critical operations (4%) were associated with unconditional logs. This means that logs generated after a crash, compromise, or code injection at the critical call may never be executed, significantly reducing the likelihood of establishing a reliable forensic trail. This study highlight a serious observability gaps in the behaviour of existing applications, which in turns limit the role of application logs in supporting forensic or security analysis.

Lastly, in the third study, this thesis explores the completeness and reliability of forensic evidence extractions under a database schema that is not static but evolves with different releases of the application. Analyzing 320 versions of 20 Android applications, the study showed that schema drift directly affects the stability and reliability of forensic acquisition queries. The study further reveals that communication and social platforms exhibit the highest levels of drift, necessitating updates to forensic aquisation queries, whereas navigation, note-taking, and certain browser applications remain largely stable with no significant schema changes. As schemas drift, queries developed for a specific database version require continual updating, and investigators must remain aware of these changes to preserve evidential completeness and reliability to avoid the loss of evidence data. The finding highlights the need for forensic tools to incorporate adaptive query generation and schema-agnostic capabilities to mitigate the impact of schema drift on evidence extraction.

Collectively, the first two studies highlight critical concerns about the adequacy of application logs to meet the availability and consistency of data for forensic investigations. The results show that existing logs practice often fail to provide the information required for security-focused analysis. Also the critical execution paths are frequently left unrecorded, creating blind spots into the defined log placement for security. The third study extends this concern to the mobile domain, showing that application updates and schema changes can disrupt evidence acquisition, impacting the completeness of data evidence. As schemas evolve, new sources of evidence may be overlooked, and destructive changes may result in the loss of previously accessible data. Together, these studies show that both application logs and mobile databases face fundamental challenges that affect the availability, consistency, and completeness of their data for forensic purposes.

6.2 Future Work

This thesis presents a novel contribution and serves as a foundation for several future research directions. By presenting the importance of forensic readiness and the current

insufficiencies in application logging for security-related tasks, it opens the door for the next research work aimed at enhancing logging systems. One beneficial direction of future work is to develop an automated approach to detect inconsistencies and enforce consistent, forensic-ready logging practices across applications.

This thesis has presented the groundwork by identifying what to log, where logs should be placed, and why certain actions require logging to enable effective post-incident analysis. These insights provide a foundation for building logging systems aimed at forensic completeness especially in desktop environment. However, balancing comprehensive logging with optimization remains an open challenge. Future research can integrate strategies such as selective log summarization, log compression, or secure anonymization into logging techniques, and still making sure that forensic value in logs is preserved.

In Chapter 4, it presents a static method for identifying where to place logs for forensic purposes, based on statically determined execution paths. While this study relies on static analysis for its scalability and ease of integration into development workflows, future work could complement this solution with a more costly (but potentially more precise) dynamic tracing. Dynamic tracing allow runtime observability across different real-world execution scenarios. Plus, this way of tracing also allow to mark logs statement that dynamically generated such those introduced via macro expansion or runtime instrumentation. This future work will enabling a more complete and realistic understanding of logging behavior across diverse execution path in real world. Another direction for future work is to propose practical strategies for implementing logs that are forensic-ready while remaining mindful of system trade-offs such as performance overhead, storage consumption, and privacy concerns.

Future work could extend the final findings of this thesis to a wider range of application categories covering financial apps and other data-intensive services. Furthermore, developing prototype drift-aware forensic tools will allow automated schema monitoring and adaptive query regeneration. This future work could be crucial for reactive maintenance for a proactive and resilient evidence acquisition.

Appendices

.1 Appendices for Chapter 4

Percentages of User, Network, File, Open Connection and Open File Data in Studied Applications.

Table 1 illustrates the distribution of logs by identifier, including user, network, and file logs, with the corresponding number and percentage of each type of log. Additionally, it shows the number and percentage of log entries that include open network connections and open files.

Table 1: Number and percentage of user, network, file unique identifier in studied applications

Categories	**Applications**	**No. of User UID (Percentage)**	**No. of Network UID (Percentage)**	**No. of File UID (Percentage)**
File managers	Files-Nautilus	0 (0)	11 (8.59)	4 (3.13)
	Disks	0 (0)	2 (2.94)	5 (7.35)
	Dolphin	0 (0)	2 (4.26)	3 (6.38)
	nnn	0 (0)	0 (0)	7 (5.93)
	recoll	0 (0)	24 (1.37)	112 (6.37)
	Konqueror	0 (0)	36 (9.73)	41 (11.08)
	Thunar	0 (0)	4 (3.81)	7 (6.67)
Chat and Communication	IceChat(Windows)	2 (0.77)	30 (11.49)	14 (5.36)
	BeeBEEP	177 (15.43)	171 (14.91)	232 (20.23)
	Pidgin	65 (4.92)	74 (5.61)	40 (3.03)
	signal-desktop	34 (2.46)	36 (2.6)	59 (4.26)
	mattermost desktop	0 (0)	28 (12.5)	3 (1.34)
	Konversation	20 (7.72)	45 (17.37)	11 (4.25)
	jitsi	77 (2.69)	143 (5)	12 (0.42)
	wire-desktop	10 (4.72)	6 (2.83)	9 (4.25)
	session-desktop	0 (0)	28 (4.6)	3 (0.49)
	HexChat	18 (2.95)	44 (7.2)	28 (4.58)
	Terminal- Irssi	25 (5.68)	23 (5.23)	12 (2.73)
	tox	17 (3.85)	14 (3.17)	2 (0.45)
Office	Calligra	0 (0)	23 (3.3)	37 (5.3)
	Xournal	0 (0)	0 (0)	11 (4.7)
	SumatraPDF	0 (0)	7 (1.79)	57 (14.58)
	zael	0 (0)	0 (0)	13 (36.11)
	Okular	0 (0)	11 (2.53)	58 (13.36)

(Continued on next page)

76

Categories	Applications	No. of User UID (Percentage)	No. of Network UID (Percentage)	No. of File UID (Percentage)
	xpdf	0 (0)	0 (0)	94 (8.79)
	ApacheOpenOffice	8 (0.05)	227 (1.49)	80 (0.53)
	Scribus	0 (0)	0 (0)	25 (4.85)
	Evince-DocumentViewer	51 (8.92)	17 (2.97)	37 (6.47)
Torrent and file sharing platform	qBittorrent	3 (0.69)	44 (10.07)	62 (14.19)
	Deluge	0 (0)	5 (0.39)	52 (4.11)
	transmissiongtk	3 (1.55)	28 (14.51)	32 (16.58)
	frostwire	0 (0)	50 (5.31)	53 (5.63)
	warpinator	0 (0)	37 (27.61)	19 (14.18)
Screen-capture	ShareX	2 (0.85)	22 (9.32)	27 (11.44)
	OBS-Studio	0 (0)	1 (0.11)	55 (5.95)
	Greenshot(Windows)	0 (0)	24 (3.51)	34 (4.97)
Security or Antivirus	Seahorse	0 (0)	0 (0)	1 (1.14)
	hashcat	0 (0)	0 (0)	110 (7.24)
	ClamAntiVirus(Linux)	9 (0.6)	67 (4.49)	153 (10.25)
Text editor	Zettlr	0 (0)	0 (0)	30 (33.71)
	gnome-text-editor	0 (0)	5 (4.17)	0 (0)
	Vim	0 (0)	4 (2.86)	10 (7.14)
	jrnl	0 (0)	0 (0)	5 (22.73)
Utilities	Console	0 (0)	0 (0)	0 (0)
	Cheese	0 (0)	0 (0)	0 (0)
	Guvcview	0 (0)	51 (5.66)	56 (6.22)
	XDM	2 (0.38)	20 (3.83)	21 (4.02)
Media player	VLCMediaplayer	0 (0)	9 (1.98)	10 (2.2)
	Kodi	297 (5.29)	506 (9.01)	442 (7.87)
	musikcube	0 (0)	10 (8.2)	8 (6.56)
Window manager	IceWM	58 (9.37)	12 (1.94)	54 (8.72)

(Continued on next page)

Categories	**Applications**	**No. of User UID (Percentage)**	**No. of Network UID (Percentage)**	**No. of File UID (Percentage)**
	tmux	1 (0.15)	12 (1.85)	24 (3.71)
	bspwm	0 (0)	3 (1.52)	0 (0)
	Qtile	0 (0)	4 (1.54)	9 (3.47)
Remote desktop	Remotely	4 (1.56)	8 (3.13)	4 (1.56)
	xrdp	43 (3.95)	88 (8.08)	98 (9)
App launcher	albert	0 (0)	2 (2.08)	7 (7.29)
	launchy	0 (0)	1 (3.03)	4 (12.12)
	ulauncher	0 (0)	5 (5.21)	12 (12.5)
	Wox	0 (0)	6 (4)	35 (23.33)

78

Logs for Partitioning, Availability, Logs Configurable and Timestamped Entries

Table 2 provides a detailed overview of various applications and their log entries for partitioning, default availability, configurable logs, and timestamped entries.

Table 2: Execution partitioning, default availability, configurable logs, and timestamped entries

Categories	**Applications**	**Execution Partitioning**	**Log available by default**	**Log configuration**	**Date and Time Data**
File managers	Files - Nautilus	✗	✗	✓	✓
	Disks	✗	✗	✓	✗
	Dolphin	✗	✗	✓	✗
	nnn	✗	✗	✓	✗
	recoll	✓	✗	✓	✓
	Konqueror	✗	✗	✓	✗
	Thunar	✗	✗	✓	✗
Chat and Communication	IceChat(Windows)	✓	✓	✓	✗
	BeeBEEP	✓	✗	✓	✗
	Pidgin	✓	✗	✓	✗
	signal-desktop	✓	✓	✓	✓
	mattermost desktop	✓	✗	✓	✓
	Konversation	✓	✗	✓	✗
	jitsi	✓	✗	✓	✓
	wire-desktop	✓	✓	✓	✓
	session-desktop	✓	✗	✓	✓
	HexChat	✗	✗	✓	✓

(Continued on next page)

80

Categories	Applications	Execution Partitioning	Log available by default	Log configuration	Date and Time Data
	Terminal- Irssi	✗	✗	✓	✓
	tox	✓	✗	✓	✗
Office	Calligra	✗	✗	✓	✗
	Xournal++	✗	✗	✓	✓
	SumatraPDF	✓	✗	✓	✗
	zael	✓	✗	✓	✗
	Okular	✗	✗	✓	✗
	xpdf	✗	✗	✗	✗
	ApacheOpenOffice	✓	✗	✓	✗
	Scribus	✗	✗	✓	✗
	Evince-DocumentViewer	✗	✗	✓	✗
Torrent and file sharing platform	qBittorrent	✓	✗	✓	✓
	Deluge	✗	✗	✓	✓
	transmissiongtk	✓	✗	✓	✓
	frostwire	✓	✗	✓	✓
	warpinator	✓	✗	✓	✓
Screen-capture	ShareX	✓	✓	✗	✗
	OBS-Studio	✓	✓	✓	✓

(Continued on next page)

Categories	Applications	Execution Partitioning	Log available by default	Log configuration	Date and Time Data
	Greenshot(Windows)	✓	✓	✓	✓
Security or Antivirus	Seahorse	✗	✗	✓	✓
	hashcat	✗	✗	✓	✓
	ClamAntiVirus(Linux)	✓	✗	✓	✓
Text editor	Zettlr	✓	✗	✓	✓
	gnome-text-editor	✗	✗	✓	✓
	Vim	✓	✗	✓	✗
	jrnl	✓	✗	✓	✓
Utilities	Console	✗	✗	✓	✗
	Cheese	✗	✗	✓	✓
	Guvcview	✓	✗	✓	✗
	XDM	✗	✗	✗	✗
Media player	VLC Media Player	✓	✗	✓	✗
	musikcube	✓	✓	✓	✓
	Kodi	✓	✓	✓	✓
Window manager	IceWM	✓	✗	✓	✗
	tmux	✓	✗	✓	✓
	bspwm	✓	✗	✓	✗

(Continued on next page)

82

Categories	**Applications**	**Execution Partitioning**	**Log available by default**	**Log configuration**	**Date and Time Data**
	Qtile	✗	✓	✓	✓
Remote desktop	Remotely	✓	✓	✓	✓
	xrdp	✓	✗	✓	✗
App launcher	albert	✓	✗	✓	✗
	launchy	✓	✗	✓	✗
	ulauncher	✗	✗	✓	✓
	Wox	✓	✓	✗	✓
TOTAL		37	11	56	31

Analysis of Timestamp Granularity in Application Logs

Table 3 showcases the granularity of date and timestamps in logs across various applications, offering insights into how different software systems prioritize time-precise logging.

Table 3: Date and time granularity of logs across applications

Categories	**Applications**	**Year**	**Month**	**Date**	**Hour**	**Min**	**Sec-**	**Mili-sec-ond**	**UTC time-zone**
File managers	Files - Nautilus	✗	✗	✗	✓	✓	✓	✓	✗
	recoll	✓	✓	✓	✓	✓	✓	✗	✗
Chat and Communication	signal-desktop	✓	✓	✓	✓	✓	✓	✓	✓
	mattermost desktop	✓	✓	✓	✓	✓	✓	✓	✗
	jitsi	✓	✓	✓	✓	✓	✓	✓	✓
	wire-desktop	✓	✓	✓	✓	✓	✓	✓	✗
	session-desktop	✓	✓	✓	✓	✓	✓	✗	✓
	HexChat	✗	✓	✓	✓	✓	✓	✗	✗
	Terminal- Irssi	✗	✗	✗	✓	✓	✓	✓	✗
Office	Xournal++	✗	✗	✗	✓	✓	✓	✓	✗
Torrent and file sharing platform	qBittorrent	✓	✓	✓	✓	✓	✓	✓	✓
	Deluge	✓	✓	✓	✓	✓	✓	✓	✗
	transmissiongtk	✓	✓	✓	✓	✓	✓	✗	✗
	frostwire	✓	✓	✓	✓	✓	✓	✗	✗
	warpinator	✓	✓	✓	✓	✓	✓	✗	✗
Screen-capture	OBS-Studio	✗	✗	✗	✓	✓	✓	✓	✗

(Continued on next page)

Categories	**Applications**	**Year**	**Month**	**Date**	**Hour**	**Min**	**Sec-**	**Mili-sec-ond**	**UTC time-zone**
	Greenshot(Windows)	✓	✓	✓	✓	✓	✓	✓	✗
Security or Antivirus	Seahorse	✗	✗	✗	✓	✓	✓	✓	✗
	hashcat*	✓	✓	✓	✓	✓	✓	✗	✗
	ClamAntiVirus(Linux)	✓	✓	✓	✓	✓	✓	✗	✗
Text editor	Zettlr	✗	✗	✗	✓	✓	✓	✗	✗
	gnome-text-editor	✗	✗	✗	✓	✓	✓	✗	✗
	jrnl	✗	✗	✗	✓	✓	✓	✗	✗
Utilities	Cheese	✗	✗	✗	✓	✓	✓	✓	✗
Media player	musikcube	✗	✗	✗	✓	✓	✓	✗	✗
	Kodi	✓	✓	✓	✓	✓	✓	✓	✗
Window manager	tmux**	✗	✗	✗	✓	✓	✓	✓	✗
	Qtile	✓	✓	✓	✓	✓	✓	✗	✗
Remote desktop	Remotely	✓	✓	✓	✓	✓	✓	✓	✗
App launcher	ulauncher	✓	✓	✓	✓	✓	✓	✗	✗
	Wox	✓	✓	✓	✓	✓	✓	✓	✗
TOTAL		19	20	20	31	31	31	17	4

* Indicates the application that logs the date and time solely at the start and conclusion of an action's execution.

** Refers to the application showcasing granularity up to the microsecond level.

86

Presentation of logs on user actions and their level

Table 4 presents the availability of logs on user actions and their level for different applications. The table includes several categories of actions on different category of applications. Each user actions is listed with a "Yes" or "No" indicating whether they have the feature of logging user actions, and the level of the log is available in bracket. Dash (–) means the application does not provide the feature related to that user action. This table serves as a reference for understanding the availability of log entries for various user actions across different categories of applications then used to referred application logs for misuse detection.

Table 4: Logs availability on user actions and their level

Categories	Applications	Search Query	Permission changes	Assessing File	Zip file activity
File managers	Files - Nautilus	No	No	No	No
	GNOME disks	–	–	–	–
	Dolphin	No	No	No	No
	nnn	No	–	Yes (No level)	Yes (No level)
	recoll	Yes (Debug)	–	Yes (Debug)	Yes (Debug)
	Konqueror	No	No	No	No
	Thunar	No	No	No	No

Categories	Applications	Login activity	Shared link is clicked	Anonymized IDs of conversation participant	File download activity
Chat and Communication	IceChat (Windows)	No	No	Yes (In conversation log)	No
	HexChat-Client	Yes (No level)	Yes (In conversation log)	Yes (In conversation log)	Yes (In conversation log)
	Terminal- Irssi	No	No	No	No
	BeeBEEP	Yes (Debug)	Yes (Debug)	Yes (Debug)	Yes (Debug)
	Pidgin	Yes (Notice)	Yes (In conversation log)	Yes (In conversation log)	No
	Signal-Desktop	No	No	No	No
	tox	Partial (Debug)	No	No	Partial (Debug)
	mattermost-desktop	No	Yes (Debug)	Yes (In conversation log)	No
	konversation	No	Yes (In conversation log)	Yes (In conversation log)	No
	jitsi	No	No	No	No
	wire-desktop	Partial (Info)	No	No	No
	session-desktop	Partial (Info)	No	Partial (Info)	Partial (Warn)

Categories	Applications	Executing files with embedded scripts	User clicking on a URL within a document	Attemped access a restricted file	Executing macros	Making changes to settings
Office	Scribus	No	–	No	–	No
	Calligra Words	No	No	No	–	No
	Xournal++	No	–	No	–	No
	LibreOffice Writer	No	No	No	No	No
	SumatraPDF	No	No	No	–	No
	zael	No	No	–	–	No
	Evince	No	No	No	–	No
	Okular	No	Yes (Debug)	No	–	No
	xpdf	No	No	No	–	–

Categories	Applications	Downloading file shared	File start seeding	Search query	Peer connection successful
Torrent and file sharing platform	qBittorrent	Yes (No Level)	No	–	–
	Deluge	No	No	–	–
	transmissiongtk	Yes (Info)	Yes (Debug)	–	–
	frostwire	Yes (Info)	No	Partial (Severe)	–
	warpinator	No	–	No	Yes (Info)

Table 4 Continued: Logs availability and level

Categories	Applications	Login activity	Accept to run quarantine file	Changes in setting
Security or Antivirus	Seahorse/Keyring	No	–	No
	hashcat	–	–	–
	ClamAntiVirus(Linux)	–	No	No

Categories	Applications	Installing and using external plugin	Clicking links through the editor	Opening and editing documents
Text editor	Zettlr	No	–	The application does not allowed to run under root
	gnome-text-editor	No	No	No
	Vim	Partial (No level) - dir of plugin	No	Partial (no level)
	jrnl	–	No	The application does not allowed to run under root

Categories	Applications	Start application service	Document downloaded or saved	External plugin installed
Utilities	Console	No	–	No
	Cheese	Yes (Info)	No	–
	Guvcview	Yes (No Level)	Yes (No Level)	–
	XDM	Yes (No Level)	Yes (No Level)	–

Categories	Applications	Opened file	Exporting media	Importing subtitle	Playing media	Importing external plugins	Importing external plugins
Media player	VLC Media Player	Yes (Debug)	Yes (Debug)	Yes (Debug)	Yes (Debug)	Yes (Debug)	Yes (Debug)
	musikcube	Yes (Info)	No	–	Yes (Info)	No	No
	Kodi	Yes (Debug)	–	Yes (Debug)	Yes (Info)	Yes (Debug)	Yes (Debug)

Categories	Applications	Any setting changes	Start new session
Window manager	IceWM	Yes (Fail)	Yes (Message)
	tmux	No	Yes (Debug)
	bspwm	Yes (Error)	No
	Qtile	Yes (Error)	No

Categories	Applications	Login activity	Start remote connection	Failed connection
Remote desktop	Remotely	Yes (Info)	No	Yes (Warning)
	xrdp	Yes (Info)	Yes (Debug)	Yes (Error)

Categories	Applications	User search query	Name of application/document launched	Changes in setting
App launcher	albert	Yes (Debug)	No	Yes (Debug)
	launchy	Yes (Debug)	Yes (debug)	No
	ulauncher	Partial (Debug)	Yes (Info)	Yes (Info)
	Wox	No	No	No

Exception Block Distribution, Logs within Blocks, and Logs with Exceptions

The table (referenced as Table 5) displays the distribution of the number of exception blocks in the studied applications, the number of logs within those blocks, and the number of logs that contain exception data within the defined logs. The percentage representation of each data is included in parentheses.

Table 5: Number of exception blocks, logs within those blocks and logs with exceptions

Categories	**Applications**	**Number of Exception Block**	**Number of Logs Statement in Exception (Percentage)**	**Number of Logs with Exception Info (Percentage)**
File managers	recoll	48	9 (18.75)	0 (0.00)
Chat and Communication	IceChat(Windows)	231	52 (22.51)	52 (100.00)
	signal-desktop	395	218 (55.19)	181 (83.03)
	mattermost desktop	51	38 (74.51)	32 (84.21)
	jitsi	2015	1219 (60.50)	1194 (97.95)
	wire-desktop	63	23 (36.51)	21 (91.30)
	session-desktop	225	104 (46.22)	83 (79.81)
Office	Calligra	71	23 (32.39)	10 (43.48)
	Xournal	43	23 (53.49)	17 (73.91)
	ApacheOpenOffice	4410	1379 (31.27)	1027 (74.47)
Torrent and file sharing platform	qBittorrent	50	14 (28.00)	13 (92.86)
	Deluge	450	21 (4.67)	4 (19.05)
	transmissiongtk	18	2 (11.11)	2 (100.00)
	frostwire	1352	344 (25.44)	341 (99.13)
	warpinator	83	26 (31.33)	26 (100.00)
Screen-capture	ShareX	193	112 (58.03)	109 (97.32)
	Greenshot(Windows)	323	250 (77.40)	247 (98.80)
Text editor	Zettlr	97	17 (17.53)	16 (94.12)
	jrnl	38	0 (0.00)	0 (0.00)
Utilities	XDM	342	187 (54.68)	183 (97.86)
Media player	musikcube	100	32 (32.00)	2 (6.25)
Window manager	Qtile	228	30 (13.16)	0 (0.00)
Remote desktop	Remotely	198	128 (64.65)	123 (96.09)
App launcher	ulauncher	52	6 (11.54)	6 (100.00)
	Wox	105	56 (53.33)	56 (100.00)

Log percentages with user, network, and file data, as well as text-only events.

The Figure 1 provides a visual representation of the distribution of log percentages with user, network, and file data, as well as text-only events.

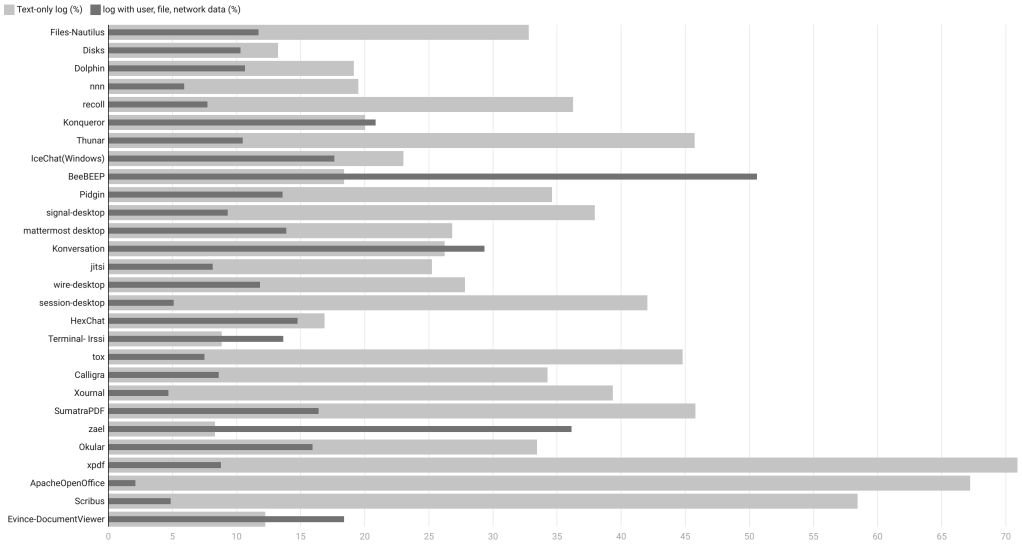

Figure 1: The distribution of log percentages with user, network, and file data, as well as text-only events.

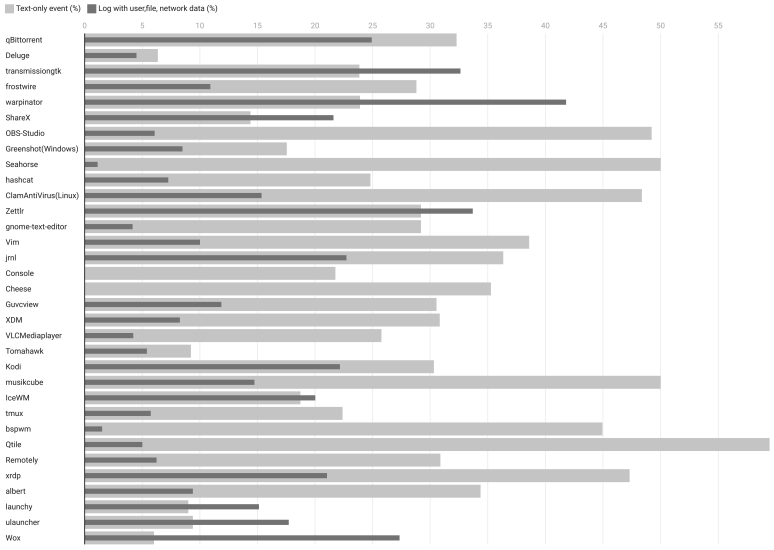

Figure 1 Continued: The distribution of log percentages with user, network, and file data, as well as text-only events.

Résumé en français

Fiabilité des données générées par les applications comme preuve de sécuritée

Les preuves numériques jouent un rôle crucial dans les enquêtes judiciaires modernes, servant de fondement pour révéler les événements, les actions et les intentions au sein des systèmes numériques. Presque chaque interaction avec un système numérique laisse derrière elle des traces numériques qui peuvent être analysées afin de reconstituer les événements et d'attribuer les actions numériques. Ces traces peuvent contenir à la fois des données issues du fonctionnement technique d'un système et des actions effectuées par les utilisateurs. Bien que ces traces constituent des sources précieuses de preuves numériques, leur valeur probante dépend largement de la plateforme sur laquelle elles sont générées ainsi que de la manière dont cette plateforme enregistre, stocke et préserve ces données.

Dans les environnements de bureau et les serveurs, par exemple, les enquêtes judiciaires reposent souvent sur les artefacts du système d'exploitation et les événements au niveau du noyau, tels que les journaux système, les métadonnées du système de fichiers et les enregistrements de processus. Dans les environnements serveurs, l'analyse judiciaire inclut fréquemment l'examen des journaux d'application et des journaux réseau, qui enregistrent les activités du système et des utilisateurs pendant l'exécution. Bien que ces sources aient été largement étudiées dans les recherches précédentes en criminalistique numérique, les données au niveau des applications sur les systèmes de bureau ont, quant à elles, reçu beaucoup moins d'attention.

À l'inverse, la criminalistique mobile s'est développée autour des données issues des applications. Étant donné que l'acquisition physique complète des appareils mobiles est souvent difficile ou restreinte, les outils d'analyse judiciaire se concentrent principalement sur l'extraction des données générées et stockées par les applications. Cette couche de données inclut généralement les messages, les journaux d'appels, les fichiers multimédias et les enregistrements synchronisés dans le cloud. En conséquence, les données applicatives sont devenues le point central de l'analyse en criminalistique mobile, et la compréhension de la complétude et de la fiabilité de l'acquisition de preuves dépend fortement de la cohérence de ces données au niveau des applications.

Les applications constituent l'interface principale entre les utilisateurs et les systèmes, générant en continu des traces numériques qui décrivent à la fois le comportement des utilisateurs et les processus internes du système. Cependant, les données qu'elles enregistrent, l'endroit où elles les stockent et la durée de leur conservation dépendent des choix de conception effectués lors du développement et de la maintenance. Ces décisions influencent directement la disponibilité, la cohérence et la complétude des preuves

numériques potentielles, affectant ainsi la capacité des enquêteurs à reconstituer les activités ou à détecter des incidents au fil du temps. Par exemple, une entrée de journal essentielle pour retracer une intrusion au sein d'une application peut ne jamais être créée, n'être enregistrée que dans des conditions particulières, ou encore être modifiée ou supprimée ultérieurement. Lorsque de telles données de journalisation – notamment celles des applications de bureau, encore peu étudiées – sont manquantes ou incohérentes, la disponibilité, la cohérence et la complétude des données pour les tâches d'investigation deviennent d'autant plus incertaines. De plus, les bases de données applicatives contenant des identifiants clés peuvent être restructurées ou supprimées dans le cadre de mises à jour fonctionnelles, d'améliorations de sécurité ou de révisions liées à la confidentialité. Ce phénomène est particulièrement visible dans les applications mobiles, où les structures de bases de données évoluent souvent à mesure que les développeurs ajoutent, modifient ou retirent des champs de données. Puisque la criminalistique mobile repose fortement sur les données au niveau des applications, de tels changements peuvent directement influencer la cohérence et la fiabilité de l'analyse judiciaire.

Cela reflète la nature imprévisible des données produites par les applications, où les informations pertinentes pour la preuve peuvent apparaître de manière incohérente dans les journaux, changer d'emplacement entre les tables d'une base de données, voire disparaître complètement de celle-ci. Une telle variabilité soulève des incertitudes fondamentales quant à la capacité des preuves numériques produites par une application à demeurer suffisantes et fiables d'un point de vue judiciaire au fil du temps.

Cette thèse aborde ces incertitudes à travers le prisme de la fiabilité des données probantes générées par les applications. Ici, la fiabilité désigne la capacité d'une application à fournir des données numériques stables et valides sur le plan judiciaire, malgré les mises à jour logicielles ou les conditions incertaines (comme des journaux destinés uniquement au débogage). Cette fiabilité est évaluée à travers trois facteurs critiques qui traduisent directement ces incertitudes : (1) la présence (disponibilité) et les conditions des journaux applicatifs, qui déterminent si les événements liés à la sécurité sont enregistrés et s'ils contiennent suffisamment de détails pour une tâche d'investigation ; (2) la cohérence de la couverture des journaux autour de l'exécution des actions critiques, définissant le meilleur emplacement des journaux dans le code source de l'application pour les données de journalisation pertinentes à la sécurité ; (3) la persistance et la stabilité structurelle des preuves stockées dans les bases de données, qui déterminent si les requêtes judiciaires peuvent continuer à extraire (de manière complète) les données de preuve à mesure que l'application évolue.

Ensemble, ces facteurs définissent la fiabilité fondamentale des données produites par les applications pour la sécurité et mettent en évidence les principales sources de variabilité qui l'affectent, constituant ainsi la base des questions de recherche explorées dans cette thèse.

La suite de la thèse se concentre sur ces deux sources majeures de preuves – les journaux et les bases de données – afin d'expliquer comment les données probantes peuvent représenter et préserver les informations liées à la sécurité, et d'évaluer puis renforcer la fiabilité des applications dans la production de ces données pour les enquêtes judiciaires.

.1.1 Journaux comme preuves pertinentes pour la sécurité

Les journaux fournissent une vue chronologique du comportement des logiciels et des services, en capturant des événements internes et externes qui révèlent le fonctionnement d'un système. Bien qu'ils aient été introduits principalement pour le débogage et la surveillance des performances, les mécanismes de journalisation sont devenus tout aussi essentiels pour la sécurité, car ces mêmes enregistrements d'événements peuvent révéler des attaques, des violations de politiques ou des comportements anormaux. Ces enregistrements apportent les données contextuelles nécessaires pour détecter des menaces, prévenir des compromissions et reconstituer des activités malveillantes. Par exemple, les journaux du système d'exploitation décrivent les activités du noyau et la création de processus, les journaux réseau exposent les schémas de communication et les flux anormaux, et les journaux applicatifs documentent les actions des utilisateurs ainsi que les changements d'état spécifiques à l'application. Lorsque ces différentes sources de données sont corrélées, les enquêteurs peuvent les utiliser comme de véritables « témoins numériques » afin de reconstituer le qui, quoi, quand, où, pourquoi et comment des événements numériques. Par exemple, ils permettent de relier une connexion suspecte au niveau du système d'exploitation à une exfiltration ultérieure de données via les appels réseau d'une application. Pour que ces données de journal puissent constituer une preuve numérique fiable, leur intégrité, leur provenance ainsi que leur collecte et leur conservation correctes doivent être garanties afin d'assurer leur recevabilité juridique et une reconstitution efficace des incidents [Ken04]. Les pratiques établies, telles que le hachage cryptographique et les signatures numériques, l'horodatage sécurisé, l'agrégation centralisée des journaux via des canaux chiffrés et le stockage en mode append-only, ainsi que les recommandations internationales telles que NIST SP 800-92 et ISO/IEC 27037, fournissent les bases techniques et procédurales pour collecter et préserver les journaux de manière juridiquement défendable. Tout aussi essentielles pour leur valeur médico-légale sont la disponibilité et la cohérence de ces journaux. La disponibilité signifie que les événements importants pour la sécurité sont effectivement enregistrés et restent accessibles pour l'analyse, tandis que la cohérence garantit que ces enregistrements sont complets et fiables dans le temps. Toute absence ou incohérence compromet directement la capacité des journaux à servir de preuve pertinente pour la sécurité. Ce défi est particulièrement marqué au niveau applicatif. La journalisation au sein des applications est fréquemment ad hoc, les développeurs privilégiant le débogage et les performances plutôt que la sécurité. En conséquence, les opérations sensibles à la sécurité peuvent rester partiellement ou totalement non journalisées, en particulier lors de défaillances de l'application ou d'attaques informatiques. La compréhension et l'amélioration de l'adéquation des journaux applicatifs en tant que preuves pertinentes pour la sécurité constituent donc un point central de cette thèse. Elle examine comment les pratiques actuelles de journalisation influencent l'acquisition et la complétude des preuves dans les enquêtes numériques.

.1.2 Applications mobiles comme sources de preuves

Une étude portant sur 12,763 jugements en appel au Royaume-Uni (2006–2011) a montré que les poursuites impliquant des preuves numériques issues de téléphones mobiles sont passées de 51 à 157 cas [MGB13]. De même, une recherche menée en Chine a révélé que 3,3,% de 66,552 affaires pénales (2013–2018) impliquaient des preuves provenant de télé-

phones mobiles, les relevés d'appels étant majoritaires et les données issues de WeChat passant de 1% à 25%, en particulier dans les affaires liées au trafic de drogue et au terrorisme [ZBMN22]. Ces résultats soulignent que les appareils mobiles sont devenus des sources essentielles de preuves numériques, offrant des données telles que les journaux d'appels, les messages texte, les historiques de conversations, les traces de localisation et les fichiers multimédias. Les artefacts mobiles peuvent être acquis à plusieurs niveaux, allant des systèmes de fichiers accessibles à l'utilisateur et des stockages privés d'applications jusqu'aux extractions bit à bit permettant de retrouver des fichiers supprimés, des données de connexion ou des mots de passe. De nombreux outils et méthodes, qu'ils soient commerciaux ou libres, soutiennent l'acquisition médico-légale de données mobiles. Parmi eux, on peut citer Magnet AXIOM, Cellebrite UFED, Belkasoft et Autopsy, qui permettent aux enquêteurs de récupérer divers artefacts à différents niveaux d'acquisition, notamment l'extraction physique, l'extraction du système de fichiers et l'extraction logique. L'extraction physique crée une image complète bit à bit de l'appareil, l'extraction du système de fichiers récupère la structure complète des répertoires et des fichiers, et l'extraction logique acquiert les données actives des applications et des utilisateurs via les interfaces du système. Dans les appareils Android, grâce au support natif du système d'exploitation, les applications gèrent leurs données dans des répertoires isolés (sandbox). Au fil du temps, ces données sont généralement stockées dans des bases structurées comme SQLite. Ces bases de données enregistrent les interactions entre l'utilisateur et l'application, y compris les messages, les contacts, les historiques d'appels et de localisation, les fichiers multimédias, les jetons d'authentification et d'autres journaux d'activité. L'acquisition logique [ABJ14], qui consiste à extraire des fichiers et des bases de données via les interfaces standard du système d'exploitation sans modifier l'appareil, est devenue une technique essentielle pour la récupération des données internes d'une application. Les résultats de cette acquisition peuvent servir de preuves clés, en particulier lorsque l'application stocke encore ses données dans SQLite. Cependant, les preuves stockées dans ces bases de données présentent des défis importants pour l'acquisition médico-légale, notamment lorsque les mises à jour de l'application modifient la structure ou le mode de stockage des données. Les développeurs peuvent modifier les informations enregistrées, les tables existantes, les colonnes incluses et la durée de conservation des données. Ainsi, l'opération d'acquisition peut changer à chaque nouvelle version publiée. Une compréhension claire de ces processus de stockage interne et de leur évolution est donc indispensable pour garantir la fiabilité de l'acquisition, car la complétude des preuves extraites constitue un facteur déterminant de la fiabilité globale. Cette thèse répond à ce besoin en étudiant comment l'évolution des schémas de bases de données des applications influence la complétude et la reproductibilité des extractions médico-légales des applications mobiles et de leurs preuves.

La solidité de ces preuves en tant que « témoin numérique » dépend en définitive de la disponibilité, de la cohérence et de la complétude des données produites par l'application. Dans le cas des journaux applicatifs, leur placement ad hoc et l'absence de directives claires pour la journalisation orientée sécurité soulèvent des doutes quant à leur adéquation pour un usage médico-légal et quant à la pertinence de leur emplacement pour la capture d'événements liés à la sécurité. À ce jour, aucun travail n'a défini ce que doivent être des journaux spécifiquement destinés à la sécurité ni établi de critères pour déterminer où ces journaux doivent être placés afin de couvrir les événements de sécurité. Un autre facteur qui peut compromettre la fiabilité des preuves est l'évolution de la plate-forme sous-jacente. Des mises à jour fréquentes d'applications peuvent modifier la structure des

données où les preuves sont enregistrées ou introduire de nouvelles données, perturbant ainsi les flux d'acquisition médico-légale établis et mettant en péril la complétude et la fiabilité des preuves numériques.

.2 Questions de recherche

Les applications génèrent des données numériques qui peuvent servir de source de preuves pour les enquêtes médico-légales. Parmi les sources les plus importantes figurent les journaux (logs), qui enregistrent des traces détaillant le quoi, le comment et le pourquoi des événements, ainsi que les bases de données, qui conservent des informations sur les utilisateurs et l'activité de l'application. Pour que ces sources possèdent une réelle valeur probante, les données qu'elles contiennent doivent rester disponibles, cohérentes et complètes dans le temps.

En pratique, il est cependant difficile de garantir cette stabilité. Les journaux sont fréquemment conçus pour le débogage ou la détection d'erreurs plutôt que pour l'investigation médico-légale, et les bases de données des applications changent souvent de structure lorsque des mises à jour ajoutent de nouvelles données ou suppriment des informations existantes. Cette variabilité introduit une incertitude fondamentale quant à la capacité des données générées par les applications à soutenir de manière fiable la reconstitution d'événements et les investigations de sécurité. Ces incertitudes créent des lacunes critiques dans la définition et l'assurance de la fiabilité des données produites par les applications comme source stable et cohérente de preuves numériques.

Pour combler ces lacunes, cette thèse vise à répondre aux trois questions de recherche suivantes:

Question #1 : Quelles sont les caractéristiques des journaux applicatifs pertinents pour l'investigation numérique et dans quelle mesure les pratiques actuelles de journalisation sont-elles préparées à soutenir les enquêtes de sécurité et d'investigation forensique ? Cette question examine la fiabilité et les incertitudes des journaux applicatifs, en se concentrant sur le premier facteur critique : la présence et les conditions des journaux. Elle analyse ce qui rend un journal applicatif pertinent pour une investigation numérique et évalue si sa présence et sa structure répondent aux exigences d'une tâche médico-légale. En s'appuyant sur des travaux préliminaires tels que [SD24], cette étude identifie et formalise les exigences essentielles concernant les informations à consigner et la manière de structurer les entrées de journal afin d'assurer leur valeur probante. Elle examine ensuite dans quelle mesure les développeurs d'applications respectent ces conditions en pratique, notamment dans les environnements de bureau où les enregistrements applicatifs ont reçu peu d'attention, afin d'évaluer l'adéquation des pratiques de journalisation actuelles aux besoins des enquêtes de sécurité et de criminalistique numérique.

Question #2 : Comment la présence et l'emplacement des instructions de journalisation peuvent-ils être définis afin de garantir une couverture complète des événements de sécurité pertinents pour l'investigation numérique ? Cette question porte sur la cohérence de la couverture des journaux autour des actions critiques, afin de s'assurer que chaque opération importante est capturée sur l'ensemble des chemins d'exécution possibles, garantissant ainsi l'exhaustivité et la cohérence des données probantes. En s'appuyant sur des travaux antérieurs centrés principalement sur

la journalisation pour le débogage [YPH+12a, ZHF+15a, FZH+14a, ZRL+17b], cette question déplace l'analyse vers une perspective de sécurité, en étudiant et en présentant le placement et la cohérence des journaux le long des chemins critiques pour l'investigation des événements de sécurité. Dans ce contexte, il est également important de considérer que de mauvaises décisions de conception en matière de placement des journaux (décrites dans la littérature comme *log smells* [SSB24]) peuvent créer des lacunes ou des incohérences compromettant la complétude des données de journalisation à valeur probante. L'objectif est d'identifier les couvertures manquantes ou incohérentes et d'évaluer dans quelle mesure les applications réelles répondent aux exigences d'une journalisation fiable et orientée sécurité.

Question #3 : Dans quelle mesure l'acquisition forensique demeure-t-elle complète et exacte lorsque les bases de données applicatives évoluent au fil des versions ? Les applications mobiles sont fréquemment mises à jour pour corriger des vulnérabilités, introduire de nouvelles fonctionnalités et améliorer la protection de la vie privée. Ces mises à jour modifient souvent la structure des bases de données internes en ajoutant, supprimant ou réorganisant des tables et des colonnes, ce qui peut déplacer l'emplacement des preuves et rendre incomplètes ou invalides les requêtes forensiques existantes. Cette question se concentre sur la persistance et la stabilité structurelle des preuves stockées dans les bases de données applicatives, en évaluant si les requêtes forensiques permettent de récupérer de manière fiable l'ensemble des données probantes lorsque les applications ajoutent, suppriment ou réorganisent ces données. Elle vise à mesurer l'impact des évolutions de schéma sur l'exhaustivité des preuves et à déterminer les conditions dans lesquelles l'acquisition forensique demeure fiable malgré les mises à jour des applications.

.3 Contributions

Cette thèse apporte plusieurs contributions majeures au domaine, présentées ci-dessous.

Contribution I (Chapitre 3)

La première contribution de cette thèse est une investigation systématique du rôle des journaux applicatifs dans le soutien à l'analyse forensique. Elle répond directement à la **Question de recherche 1 (RQ1)**, qui s'interroge sur *ce qui constitue des journaux applicatifs pertinents pour l'investigation numérique et dans quelle mesure les pratiques actuelles de journalisation soutiennent les enquêtes de sécurité et de criminalistique numérique.* Cette étude commence par définir les tâches clés que requièrent typiquement les investigations forensiques, notamment la reconstitution de chronologies, la corrélation d'événements, le partitionnement d'exécutions, la détection d'abus et la détection d'attaques. Sur cette base, elle identifie les types d'informations spécifiques que les journaux doivent contenir pour soutenir efficacement ces tâches, tels que les horodatages, les actions de l'utilisateur, les identifiants d'événements et les avertissements liés à des données anormales ou malformées. Pour évaluer dans quelle mesure ces exigences sont satisfaites en pratique, une analyse détaillée de 60 applications de bureau open source a été menée. Dans le cadre de cette évaluation, les journaux déjà implémentés dans chaque application ont été collectés puis examinés afin de déterminer s'ils incluaient les paramètres nécessaires à l'accomplissement

des tâches forensiques. Les résultats révèlent des limitations importantes dans les pratiques actuelles de journalisation à des fins de sécurité, un grand nombre d'applications ne consignant pas de manière systématique des éléments essentiels tels que les horodatages, les actions utilisateur et les conditions d'erreur. En formalisant les exigences des journaux pertinents pour l'investigation et en quantifiant empiriquement les pratiques existantes, cette contribution apporte une réponse concrète à la RQ1 et établit une base pour la conception de systèmes de journalisation applicative orientés sécurité.

Cette étude et ses résultats sont publiés dans :

- Azahari Afiqah et Davide Balzarotti
  "On the Inadequacy of Open-Source Application Logs for Digital Forensics."
  Forensic Science International: Digital Investigation, Volume 49, juin 2024. [AB24]

Contribution II (Chapitre 4)

La deuxième contribution de cette thèse est l'analyse du positionnement des instructions de journalisation par rapport aux opérations critiques. Elle répond à la **Question de recherche 2 (RQ2)**, qui étudie *comment la présence et l'emplacement des instructions de log peuvent être définis afin de garantir une couverture complète pour les investigations forensiques à finalité de sécurité.* Une stratégie de journalisation cohérente est indispensable autour des opérations sensibles à la sécurité ou sujettes à des défaillances, car ces opérations peuvent compromettre l'intégrité du système, provoquer des plantages inattendus ou ouvrir des vecteurs d'attaque. Cette étude examine si les messages de log sont émis avant ou après l'exécution d'actions critiques et si des branches conditionnelles peuvent en empêcher l'exécution. En reconstruisant les chemins d'exécution complets, elle met en évidence l'importance de l'absence de journaux le long de chaque chemin allant de l'entrée utilisateur jusqu'à l'appel critique et au-delà. L'absence de journaux à tout moment réduit leur valeur probante, car elle empêche de capturer ce qui s'est produit durant une exécution normale ou, pire, pendant un crash ou un incident de sécurité. Pour réaliser cette analyse à grande échelle, un outil nommé **BlindSpot** a été développé. Construit au-dessus de Joern, BlindSpot combine l'analyse de flux de données pour tracer les entrées non fiables, la traversée des flux de contrôle pour reconstruire les chemins d'exécution inter-procéduraux, et les vérifications de dominance pour évaluer si les instructions de log sont exécutées de manière fiable. Cela permet de déterminer automatiquement si les journaux offrent une couverture fiable des opérations critiques tout au long de l'exécution de l'application. BlindSpot a été appliqué à dix applications de bureau open source basées sur GTK. Les résultats ont révélé de nombreuses incohérences : des opérations influencées par des entrées externes étaient souvent non journalisées ou uniquement enregistrées sous certaines conditions. En définissant les exigences relatives à la présence et à l'emplacement corrects des logs, et en identifiant un nouveau critère de *log smells* liés à la présence et au positionnement ayant un impact sur la sécurité, cette contribution établit un cadre pour l'évaluation de la journalisation orientée sécurité. Elle démontre ensuite ces lacunes sur des applications réelles, apportant ainsi une réponse complète à la RQ2.

Cette étude et ses résultats sont actuellement en cours d'évaluation pour publication dans la revue *Digital Threats: Research and Practice.*

Contribution III (Chapitre 5)

La troisième contribution de cette thèse porte sur l'impact de l'évolution des bases de données sur la criminalistique mobile, en particulier sur la manière dont la disponibilité des preuves dans les bases de données des applications Android a évolué. Cette contribution répond directement à la **Question de recherche 3 (RQ3)**, qui s'interroge *dans quelle mesure l'acquisition forensique demeure complète et exacte lorsque les bases de données des applications évoluent au fil des versions.* Pour répondre à cette question, une analyse longitudinale a été menée sur 20 applications Android populaires couvrant 320 versions d'APK. Elle fournit l'une des premières vues systématiques de la façon dont les mises à jour des applications modifient la structure des bases de données et de l'impact de ces changements sur le processus d'acquisition forensique. L'étude introduit une classification des dérives de schéma, regroupant les applications en catégories de dérive forte, modérée et faible. Les applications de communication et de réseaux sociaux présentent les niveaux de dérive les plus élevés, introduisant souvent des modifications qui perturbent les méthodes forensiques existantes. Même de petits ajustements du schéma suffisent à rompre des requêtes, à modifier les résultats obtenus ou à réduire la quantité de preuves pouvant être collectées. Des études de cas détaillées montrent en outre que les changements de schéma peuvent affaiblir les flux de travail forensiques établis, mais aussi ajouter de nouvelles sources de preuves qui n'existaient pas dans les versions antérieures. Ces effets mettent en lumière à la fois les risques et les opportunités liés aux mises à jour continues des applications. Cette contribution renforce le domaine en offrant une perspective longitudinale sur la dérive de schéma dans les applications mobiles et en détaillant ses conséquences pour les investigations numériques. En quantifiant la dérive de schéma et en démontrant son impact sur la fiabilité de l'extraction des preuves, elle apporte une réponse directe à la RQ3.

Cette étude et ses résultats sont actuellement en cours d'évaluation pour une présentation et une publication à la conférence *DFRWS Europe 2026 (Digital Forensics Research Conference).*

Travaux connexes

extraire et analyser les traces numériques laissées par les applications afin de soutenir les investigations forensiques. Ces traces se retrouvent principalement à deux niveaux : d'une part dans les journaux applicatifs (logs) qui consignent les événements système et utilisateur, et d'autre part dans les bases de données internes qui conservent les données persistantes de l'application. Ces deux sources ont été étudiées en profondeur, mais selon des approches et des priorités différentes. Les travaux consacrés aux journaux se sont concentrés sur l'amélioration de la qualité et du positionnement des instructions de journalisation, initialement pour le débogage et la surveillance des performances, puis plus récemment pour la conformité en matière de sécurité et la détection d'attaques. En parallèle, les recherches portant sur les applications mobiles ont documenté l'évolution rapide des versions et ses effets sur la fonctionnalité, les permissions et les vulnérabilités. Malgré ces avancées, des lacunes importantes subsistent : la définition précise du contenu de journalisation pertinent pour la sécurité ainsi que les critères de son positionnement optimal restent mal établis, et l'impact à long terme de la dérive des schémas de bases

de données sur la complétude et la fiabilité des preuves numériques demeure largement inexploré.

.3.1 Journaux applicatifs : de l'outil de débogage à la preuve numérique pour la sécurité

Les journaux (logs) constituent depuis longtemps une source fondamentale d'analyse, permettant la reconstruction d'événements, la détection d'anomalies et l'investigation d'incidents. Ces analyses s'appuient sur un large éventail de données issues, par exemple, des journaux d'événements systèmes, des journaux d'objets connectés ou des journaux d'audit réseau. De nombreux travaux ont mis en évidence l'importance de ces données. Par exemple, He et al. [HHC+20] présentent des techniques automatisées pour la détection d'anomalies, la prédiction de pannes et le diagnostic système, tandis que Studiawan et al. [SSP19] proposent une revue complète sur l'utilisation des journaux d'événements en investigation forensique. De plus, Kara et al. [Kar21] introduisent le concept de *forensic log records*, désignant les entrées de journal pertinentes pour l'analyse d'incidents, soulignant ainsi le rôle des journaux comme source de preuve essentielle pour la reconstruction d'événements et l'investigation de la sécurité.

Plusieurs travaux se sont intéressés aux pratiques de journalisation du point de vue du développement logiciel, en cherchant à déterminer quelles informations consigner, quelles variables sont les plus pertinentes et où insérer les instructions de log dans le code. Li et al. [LHZ+24] ont proposé SCLogger, une approche contextualisée d'insertion automatique de logs basée sur l'analyse statique inter-méthodes et des modèles de langage. Dans le même esprit, Liu et al. [LXL+21] identifient les variables les plus importantes à consigner, et Li et al. [LSH18] emploient la régression ordinale pour recommander automatiquement un niveau de log approprié. Concernant le *où*, Fu et al. [FZH+14b] ont mené une étude empirique sur les pratiques industrielles de placement des logs, suivis par Zhao et al. [ZRL+17a] qui utilisent la théorie de l'information pour identifier les points de programme les plus révélateurs. Zhu et al. [ZHF+15b] ont développé LogAdvisor, un outil d'aide au placement de logs, et Zhao et al. [ZRL+17c] ont proposé d'optimiser le placement afin de minimiser la surcharge tout en maximisant la compréhension des défaillances. Saarimäki et al. [SSB24] introduisent enfin la notion de *log smells*, une taxonomie des mauvaises pratiques de conception qui dégradent la fiabilité des journaux. La majorité de ces études adoptent une perspective centrée sur le développeur, visant l'amélioration des journaux pour le débogage, la surveillance des performances et la compréhension du comportement système, tout en négligeant souvent leur rôle comme preuve numérique pour la sécurité ou l'investigation forensique. Parallèlement, plusieurs travaux se sont orientés vers la journalisation pour la conformité et la préparation forensique. Shen et al. [SSZ23] utilisent l'analyse statique pour identifier les logs manquants liés aux refus d'accès, essentiels à l'audit de sécurité. Rivera-Ortiz et Pasquale [ROP19, ROP20] proposent une approche proactive intégrant dès la conception les exigences de journalisation et automatisant le placement selon les scénarios d'attaque. Olegård et al. [OAL25] introduisent une méthode de traçabilité causale pour suivre le mouvement d'un attaquant à travers plusieurs composants. Dans le domaine de la conformité, King et Williams [KW13, KPW15, KSRW17] définissent des catégories critiques pour les dossiers médicaux électroniques, tandis que Sahin et Daniele [SD24] analysent les écarts de journalisation par rapport aux recommandations de sécurité. Malgré ces avancées, il manque encore une définition claire du

contenu précis à consigner (par exemple variables, entrées, données contextuelles) et des règles systématiques de placement assurant la complétude et la cohérence des journaux à des fins forensiques. Cette thèse comble ces lacunes en proposant un modèle de journalisation orienté forensique, définissant le contenu pertinent des logs et établissant des recommandations de placement dans le code source afin de garantir une couverture complète et fiable des actions critiques pour la sécurité.

.3.2 Application mobile : évolution des bases de données et fiabilité forensique

La criminalistique numérique appliquée aux applications mobiles a été largement étudiée sous différents angles, notamment l'extraction des preuves, l'évolution des applications au fil du temps et l'impact de ces évolutions sur la fiabilité des outils d'investigation. De nombreux travaux se sont concentrés sur le développement de techniques visant à extraire des preuves numériques à partir des bases de données des applications mobiles et à évaluer leur récupérabilité face aux changements de schéma des bases. Un premier travail en ce sens a été mené en 2015 par Sgaras et al. [SKLK15], qui ont montré comment acquérir des preuves issues d'applications de communication populaires comme WhatsApp et Viber, sans toutefois analyser l'évolution structurelle des bases au cours du temps. En parallèle, plusieurs outils open source ont été proposés pour extraire des traces numériques depuis les applications, tels que ALEAPP [Bri23], qui s'appuie sur des greffons pour analyser des copies du système de fichiers Android et produire des rapports sur l'activité des utilisateurs et des applications, Chatistics [SkS23], qui parse des journaux de conversations exportés, et Whapa [B1623], qui déchiffre et analyse les bases de données de WhatsApp. Un premier effort pour traiter spécifiquement l'évolution des schémas de bases de données applicatives a été proposé par Shimmi et al. [SDKA20], qui ont étudié l'évolution des schémas dans les sauvegardes iOS et développé une comparaison automatisée des structures de tables et de colonnes afin de maintenir la capacité d'extraction des preuves. Bien qu'ils aient démontré que la dérive de schéma (schema drift) compromet la fiabilité des requêtes, leur analyse se limite aux applications iOS natives. Dans une autre approche, Lee et al. [LCL23] ont présenté une méthode prédictive permettant d'identifier les schémas de bases de données dans les applications de messagerie, sans toutefois analyser leur évolution au fil des versions. Plus récemment, AnForA [ACG20] et Argus [BDJH25] ont été proposés pour automatiser la détection d'artefacts générés par les applications via la capture d'instantanés et l'exécution de scénarios automatisés. Ces deux outils s'appuient sur *SQLDiff* pour mettre en évidence les modifications de bases de données, mais sans évaluer l'impact de ces changements de schéma sur l'extraction de preuves forensiques. Au-delà du domaine forensique, plusieurs recherches ont examiné l'évolution des applications Android sous d'autres perspectives. Des travaux antérieurs ont étudié la complexité et la maintenabilité du code [GLBK19], l'évolution des fonctionnalités d'accessibilité [DSODJ+23, OdSAE24], ainsi que la modélisation thématique des changements de version [TAHB14]. Hecht et al. [HBR+15] ont montré comment de mauvais choix de conception lors des mises à jour peuvent dégrader les performances et la qualité. D'autres études axées sur la sécurité et la confidentialité ont révélé que les autorisations demandées par les applications évoluent parfois de manière risquée [CG17, TM17], tandis que des analyses à grande échelle de millions d'applications ont mis en évidence des problèmes persistants tels que le pistage par des tiers et des comportements malveil-

lants [WLG19]. Des recherches sur les vulnérabilités [GLK+19] ont également montré que des dépendances non sécurisées pouvaient persister au travers de multiples mises à jour. Si ces travaux mettent en évidence la rapidité d'évolution des applications mobiles, ils se concentrent principalement sur le code, les autorisations ou les fonctionnalités. À ce jour, aucun travail empirique n'a étudié l'évolution des versions d'applications sous l'angle de la dérive des schémas de bases de données ni son impact direct sur la fiabilité des extractions et acquisitions forensiques.
Cette thèse s'appuie sur ces travaux antérieurs et comble les lacunes de recherche qu'ils ont mises en évidence. Bien que les journaux d'exécution (*logs*) et les bases de données mobiles diffèrent par leur structure et leur finalité, ils fournissent tous deux des traces numériques pouvant constituer des preuves de sécurité essentielles. Pour les journaux applicatifs, cependant, la définition précise d'un contenu pertinent pour la sécurité demeure insuffisamment explorée, rendant incertaine leur adéquation à un usage forensique. De plus, la présence et la localisation des instructions de journalisation, indispensables à la reconstitution des événements dans un contexte judiciaire, restent une question ouverte. Les travaux existants sur le placement des logs [FZH+14b, ZRL+17a, ZHF+15b, ZRL+17c] se concentrent principalement sur l'amélioration du débogage et du suivi de performance. Bien que ces apports soient utiles, ils ne définissent pas de critères spécifiques à la sécurité pour déterminer où les logs doivent être positionnés ni comment garantir leur présence, alors que la disponibilité, la cohérence et l'exhaustivité des données de preuve doivent être rigoureusement assurées pour les événements de sécurité. Dans le domaine des applications mobiles, les études montrent que celles-ci évoluent rapidement, mais l'on sait peu de choses sur la manière dont les schémas de base de données, qui stockent des preuves forensiques, se transforment au fil du temps et sur les conséquences de ces changements pour la fiabilité et l'exhaustivité des preuves. Cette lacune soulève des questions critiques sur la capacité des outils d'investigation à collecter de façon constante l'ensemble des données pertinentes à mesure que les applications et leurs structures internes évoluent.

References

[AB24] Afiqah Azahari and Davide Balzarotti. On the inadequacy of open-source application logs for digital forensics. *Forensic Science International: Digital Investigation*, 49:301750, 2024.

[ABJ14] Rick Ayers, Sam Brothers, and Wayne Jansen. Guidelines on mobile device forensics. NIST Special Publication 800-101 Revision 1, National Institute of Standards and Technology, May 2014.

[abs25] absadiki. whatsapp-msgstore-viewer: Free, open-source and cross-platform tool to decrypt, read, and view whatsapp `msgstore.db` database. https://github.com/absadiki/whatsapp-msgstore-viewer, 2025. Accessed: 15 July 2025.

[ACG17] Cosimo Anglano, Massimo Canonico, and Marco Guazzone. Forensic analysis of telegram messenger on android smartphones. *Digital Investigation*, 23:31–49, 2017.

[ACG20] Cosimo Anglano, Massimo Canonico, and Marco Guazzone. The android forensics automator (anfora): A tool for the automated forensic analysis of android applications. *Computers & Security*, 88:101650, 2020.

[Ano25] Anonymous. Dataset and additional tables, 2025. Accessed: 2025-05-16.

[APK25] APKMirror. Apkmirror – free and safe android apk downloads. https://www.apkmirror.com/, 2025. Accessed: 6 August 2025.

[B1623] B16f00t. Whapa: Whatsapp parser tool. https://github.com/B16f00t/whapa, 2023. Accessed: 2025-05-16.

[BDJH25] Abdul Boztas, Jeroen De Jong, and Christos Hadjigeorghiou. Argus: A new approach for forensic analysis of apps on mobile devices. *Forensic Science International: Digital Investigation*, 53:301938, 2025.

[Bel18] Belkasoft. What's new in bec v.9.0. https://belkasoft.com/whats_new_in_version_9_0, 2018. Accessed: 6 August 2025.

[Bel25] Belkasoft. What's new in belkasoft x v.2.8. https://belkasoft.com/new, 2025. Accessed: 6 August 2025.

[Bes] Logging best practices - dzone performance.

[BM15] Dirk Brouwer and Perry Mertens. Forensic logging requirements, 2015.

[Bri23] Alexis Brignoni. ALEAPP: Android logs events and protobuf parser. https://github.com/abrignoni/ALEAPP, 2023. Accessed: 2025-05-16.

[CBG+17] Eoghan Casey, Sean Barnum, Ryan Griffith, Jonathan Snyder, Harm Van Beek, and Alex Nelson. Advancing coordinated cyber-investigations and tool interoperability using a community developed specification language. *Digital Investigation*, 22:14–45, September 2017.

[CBG+18] Eoghan Casey, Sean Barnum, Ryan Griffith, Jonathan Snyder, Harm Van Beek, and Alex Nelson. The Evolution of Expressing and Exchanging Cyber-Investigation Information in a Standardized Form. In Maria Angela Biasiotti, Jeanne Pia Mifsud Bonnici, Joe Cannataci, and Fabrizio Turchi, editors, *Handling and Exchanging Electronic Evidence Across Europe*, volume 39, pages 43–58. Springer International Publishing, Cham, 2018. Series Title: Law, Governance and Technology Series.

[Cel18] Cellebrite. Ufed ultimate & ufed infield and ufed physical analyzer, ufed logical analyzer & cellebrite reader v7.5 and analytics desktop v7.0 release notes, 2018. May 2018.

[Cel20] Cellebrite. Ufed, ufed infield, ufed physical analyzer, ufed logical analyzer and cellebrite reader v7.32 release notes, 2020. April 2020.

[CG17] Paolo Calciati and Alessandra Gorla. How do apps evolve in their permission requests? a preliminary study. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 37–41. IEEE, 2017.

[CJ17a] Boyuan Chen and Zhen Ming Jiang. Characterizing and detecting anti-patterns in the logging code. 2017.

[CJ17b] Boyuan Chen and Zhen Ming (Jack) Jiang. Characterizing logging practices in java-based open source software projects – a replication study in apache software foundation. *Empirical Software Engineering*, 22, 2017.

[CNH19] Eoghan Casey, Alex Nelson, and Jessica Hyde. Standardization of file recovery classification and authentication. *Digital Investigation*, 31:100873, December 2019.

[Con] Logging guidelines.

[CP10] Anton Chuvakin and Gunnar Peterson. How to do application logging right. *IEEE Security and Privacy*, 8(4):82–85, 2010.

[Dev] Developer's guide - logging - best practices.

[DSODJ+23] Paulo Sérgio Henrique Dos Santos, Alberto Dumont Alves Oliveira, Thais Bonjorni Nobre De Jesus, Wajdi Aljedaani, and Marcelo Medeiros Eler. Evolution may come with a price: analyzing user reviews to understand the impact of updates on mobile apps accessibility. In *Proceedings of the XXII Brazilian Symposium on Human Factors in Computing Systems*, pages 1–11, 2023.

[FZH+14a] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, page 24–33, New York, NY, USA, 2014. Association for Computing Machinery.

[FZH+14b] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, page 24–33, New York, NY, USA, 2014. Association for Computing Machinery.

[FZH+14c] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 24–33, 2014.

[GLBK19] Jun Gao, Li Li, Tegawendé F Bissyandé, and Jacques Klein. On the evolution of mobile app complexity. In *2019 24th international conference on engineering of complex computer systems (ICECCS)*, pages 200–209. IEEE, 2019.

[GLK+19] Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. Understanding the evolution of android app vulnerabilities. *IEEE Transactions on Reliability*, 70(1):212–230, 2019.

[HBR+15] Geoffrey Hecht, Omar Benomar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. Tracking the software quality of android applications along their evolution (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 236–247. IEEE, 2015.

[HGL+19] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage. In *Proceedings 2019 Network and Distributed System Security Symposium*, San Diego, CA, 2019. Internet Society.

[HHC+20] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R. Lyu. A survey on automated log analysis for reliability engineering. *CoRR*, abs/2009.07237, 2020.

[HLS+23] Yintong Huo, Yichen Li, Yuxin Su, Pinjia He, Zifan Xie, and Michael R. Lyu. AutoLog: A Log Sequence Synthesis Framework for Anomaly Detection, August 2023. arXiv:2308.09324 [cs].

[HNDB20] Wajih Ul Hassan, Mohammad A. Noureddine, Pubali Datta, and Adam Bates. Omegalog: High-fidelity attack investigation via transparent multi-layer log analysis. Internet Society, 2 2020.

[HSS20] Md Nahid Hossain, Sanaz Sheikhi, and R. Sekar. Combating Dependence Explosion in Forensic Analysis Using Alternative Tag Propagation Semantics. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1139–1155, San Francisco, CA, USA, May 2020. IEEE.

[ISO] Iso 8601 date and time format. https://www.iso.org/iso-8601-date-and-time-format.html. Accessed: January 7, 2024.

[Joe] Joern Project. Joern: Open-source code analysis platform. https://github.com/joernio/joern. Accessed: 2025-01-01.

[Kar21] Ilker Kara. Read the digital fingerprints: log analysis for digital forensics and security. *Computer Fraud & Security*, 2021(7):11–16, 2021.

[Ken04] Erin Kenneally. Digital logs: Proof matters. In *Proceedings of the 18th Annual FIRST Conference*, Budapest, Hungary, 2004. Forum of Incident Response and Security Teams (FIRST).

[KPW15] Jason King, Rahul Pandita, and Laurie Williams. Enabling forensics by proposing heuristics to identify mandatory log events. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, pages 1–11, Urbana Illinois, April 2015. ACM.

[KSRW17] Jason King, Jon Stallings, Maria Riaz, and Laurie Williams. To log, or not to log: using heuristics to identify mandatory log events – a controlled experiment. *Empirical Software Engineering*, 22(5):2684–2717, 2017.

[KW13] Jason King and Laurie Williams. Cataloging and comparing logging mechanism specifications for electronic health record systems. In *Proceedings of the 2013 USENIX Conference on Safety, Security, Privacy and Interoperability of Health Information Technologies*, HealthTech'13, page 4, USA, 2013. USENIX Association.

[LCHX23] Zhenhao Li, An Ran Chen, Xing Hu, and Xin Xia. Are They All Good? Studying Practitioners' Expectations on the Readability of Log Messages. *38th IEEE/ACM International Conference on Automated Software Engineering (ASE 2023)*, 2023.

[LCL23] Wei-Che Lee, Hong-Yen Chen, and Tsung-Nan Lin. A predictive classification model for identifying conversation-related mobile forensic data in instant messaging applications. In *2023 International Symposium on Digital Forensics and Security (ISDFS)*, pages 1–8. IEEE, 2023.

[LHZ+24] Yichen Li, Yintong Huo, Renyi Zhong, Zhihan Jiang, Jinyang Liu, Junjie Huang, Jiazhen Gu, Pinjia He, and Michael R. Lyu. Go static: Contextualized logging statement generation, 2024.

[LLC+22] Steven Locke, Heng Li, Tse-Hsun Chen, Weiyi Shang, and Wei Liu. LogAssist: Assisting Log Analysis Through Log Summarization. *IEEE Transactions on Software Engineering*, 48(9):3227–3241, September 2022.

[LLC+23] Zhenhao Li, Chuan Luo, Tse-Hsun Chen, Weiyi Shang, Shilin He, Qingwei Lin, and Dongmei Zhang. Did We Miss Something Important? Studying and Exploring Variable-Aware Log Abstraction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 830–842, Melbourne, Australia, May 2023. IEEE.

[Loga] Logging best practices - dev community.

[Logb] Logging best practices: The 13 you should know | dataset.

[Logc] Logging guidelines for developers - diwebsity.

[LSH18] Heng Li, Weiyi Shang, and Ahmed E. Hassan. Which log level should developers choose for a new logging statement? (journal-first abstract). In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 468–468, 2018.

[LXL+21] Zhongxin Liu, Xin Xia, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. Which variables should i log? *IEEE Transactions on Software Engineering*, 47(9):2012–2031, 2021.

[LZL+20] Xuyuan Li, Yongjun Zhang, Xiangzhen Li, Peng Wang, and Shuping Peng. Automated vulnerability detection in source code using minimum intermediate representation learning. *Applied Sciences*, 10(5):1692, 2020.

[MGB13] Jack Euan Ross McMillan, William Bradley Glisson, and Michael Bromby. Investigating the increase in mobile phone evidence in criminal activities. In *Proceedings of the 46th Hawaii International Conference on System Sciences (HICSS)*, pages 4898–4909. IEEE, 2013.

[MSA14] MSAB. Xry offers support for new encrypted version of whatsapp. https://www.msab.com/updates/xry-offers-support-for-new-encrypted-version-of-whatsapp, 2014. Accessed: 6 August 2025.

[MSA22] MSAB. Recover even more digital evidence from apps with xry 10.0.1. https://www.msab.com/updates/released-today-recover-even-more-digital-evidence-from-apps-with-xry-10-0-1, 2022. Accessed: 6 August 2025.

[MZW+17] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. {MPI}: Multiple perspective attack investigation with semantic aware execution partitioning. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1111–1128, 2017.

[OAL25] Johannes Olegård, Stefan Axelsson, and Yuhong Li. When is logging sufficient? — tracking event causality for improved forensic analysis and correlation. *Forensic Science International: Digital Investigation*, 52:301877, 2025. Selected Papers from DFRWS EU 2025.

[OdSAE24] Alberto Dumont Alves Oliveira, Paulo Sergio Henrique dos Santos, Wajdi Aljedaani, and Marcelo Medeiros Eler. Exploring the influence of software evolution on mobile app accessibility: Insights from user reviews. *Journal of the Brazilian Computer Society*, 30(1):584–607, 2024.

[OWA] OWASP. Logging - owasp cheat sheet series.

[PCI18] Payment card industry (pci) data security standard requirements and security assessment procedures version 3.2.1, 2018.

[Rob] Five guidelines for robust logging | hackernoon.

[ROP19] Fanny Rivera-Ortiz and Liliana Pasquale. Towards automated logging for forensic-ready software systems. In *2019 IEEE 27th International Requirements Engineering Conference Workshops (REW)*, pages 157–163. IEEE, 2019.

[ROP20] Fanny Rivera-Ortiz and Liliana Pasquale. Automated modelling of security incidents to represent logging requirements in software systems. In *Proceedings of the 15th International Conference on Availability, Reliability and Security (ARES 2020)*, pages 1–8, Virtual Event, Ireland, 2020. ACM.

[SD24] Merve Sahin and Noemi Daniele. Towards Understanding and Improving Security-Relevant Web Application Logging. 2024.

[SDKA20] Samiha S. Shimmi, Gokila Dorai, Umit Karabiyik, and Sudhir Aggarwal. Analysis of ios sqlite schema evolution for updating forensic data extraction tools. In *2020 IEEE International Conference on Intelligence and Security Informatics (ISI)*, pages 1–8. IEEE, 2020.

[sep25] sepinf-inc. IPED: Digital forensic tool to process and analyze digital evidence. https://github.com/sepinf-inc/IPED, 2025. Accessed: 15 July 2025.

[SKLK15] Christos Sgaras, M-Tahar Kechadi, and Nhien-An Le-Khac. Forensics acquisition and analysis of instant messaging and voip applications. In *Proceedings of the 13th ADFSL Conference on Digital Forensics, Security and Law (CDFSL)*. ADFSL, 2015.

[SkS23] SkSumit. Chatistics: Parse and visualize chat logs from messenger, hangouts, whatsapp, and telegram. https://github.com/SkSumit/Chatistics, 2023. Accessed: 2025-05-16.

[SSB24] Nyyti Saarimäki, Donghwan Shin, and Domenico Bianculli. Taxonomy of software log smells, 2024.

[SSP19] Hudan Studiawan, Ferdous Sohel, and Christian Payne. A survey on forensic investigation of operating system logs. *Digital Investigation*, 29:1–20, 2019.

[SSZ23] Bingyu Shen, Tianyi Shan, and Yuanyuan Zhou. Improving logging to reduce permission over-granting mistakes. In *Proceedings of the 32nd USENIX Security Symposium*, 2023.

[TAHB14] Stephen W. Thomas, Bram Adams, Ahmed E. Hassan, and Dorothea Blostein. Studying software evolution using topic models. *Science of Computer Programming*, 80:457–479, 2014.

[TM17] Vincent F. Taylor and Ivan Martinovic. To update or not to update: Insights from a two-year study of android app evolution. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, page 45–57, New York, NY, USA, 2017. Association for Computing Machinery.

[Whe04] Dennis Wheeler. Understanding seventeenth-century ships' logbooks: An exercise in historical climatology. *Journal for Maritime Research*, 6(1):21–36, December 2004.

[WLG19] Haoyu Wang, Hao Li, and Yao Guo. Understanding the evolution of mobile app ecosystems: A longitudinal measurement study of google play. In *The World Wide Web Conference*, WWW '19, page 1988–1999, New York, NY, USA, 2019. Association for Computing Machinery.

[YMZ+21] Le Yu, Shiqing Ma, Zhuo Zhang, Guanhong Tao, Xiangyu Zhang, Dongyan Xu, Vincent E Urias, Han Wei Lin, Gabriela Ciocarlie, Vinod Yegneswaran, and Ashish Gehani. Alchemist: Fusing application and audit logs for precise attack provenance without instrumentation. 2021.

[YPH+12a] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. *10th USENIX Symposium on Operating Systems Design and Implementation*, 2012.

[YPH+12b] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012*, pages 293–306, 2012.

[YPZ12] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. *Proceedings - International Conference on Software Engineering*, pages 102–112, 2012.

[ZBMN22] Aolan Zhang, Ben Bradford, Ruth M. Morgan, and Sherry Nakhaeizadeh. Investigating the uses of mobile phone evidence in china criminal proceedings. *Science & Justice*, 62(4):385–398, 2022.

[ZHF+15a] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. Learning to Log: Helping Developers Make Informed

Logging Decisions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 415–425, Florence, Italy, May 2015. IEEE.

[ZHF+15b] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. Learning to log: Helping developers make informed logging decisions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 415–425, 2015.

[ZLP+21] Yongjun Zhang, Xiangzhen Li, Shuping Peng, Peng Wang, and Xuyuan Li. Automated software vulnerability detection based on hybrid neural network. *Applied Sciences*, 11(7):3201, 2021.

[ZRL+17a] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. The game of twenty questions: Do you know where to log? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 125–131, New York, NY, USA, 2017. Association for Computing Machinery.

[ZRL+17b] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 565–581, Shanghai China, October 2017. ACM.

[ZRL+17c] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 565–581, New York, NY, USA, 2017. Association for Computing Machinery.