

http://researchcommons.waikato.ac.nz/

Research Commons at the University of Waikato

Copyright Statement:

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

The thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of the thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from the thesis.

Redesigning Files to Embed Concise Provenance Metadata

A thesis

submitted in fulfilment

of the requirements for the Degree

of

Doctor of Philosophy

at

The University of Waikato

by

Thye Way Phua



THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

2026

Abstract

Data provenance is essential for validating the origin, history, and trustworthiness of information. However, despite extensive research, provenance systems have seen limited mainstream adoption. This thesis argues that this gap stems from two fundamental deficiencies: (i) the lack of temporal awareness in existing models and (ii) the detachment of provenance metadata from the primary data. Current systems often treat versioning as symbolic markers rather than capturing the content of change, and they store metadata in centralised repositories that are prone to desynchronisation.

To address these challenges, this research proposes a framework that integrates temporal data evolution with embedded provenance. The thesis posits that provenance must be integral to the data artefact itself to ensure portability and integrity. A prototype system was developed using a FUSE-based filesystem, employing delta encoding to capture fine-grained changes and a custom block-based format to embed provenance metadata and version histories directly within files. This architecture ensures that lineage information remains inseparable from the data it describes.

The system was evaluated through microbenchmarks and a macrobenchmark involving a full Linux kernel compilation. The evaluation confirms that the delta-based storage model is efficient, achieving a sixty-fold reduction in space compared to full snapshot retention. Historical state reconstruction incurs linear overhead relative to version depth, remaining manageable for typical file lifespans. Although the prototype introduces a performance overhead of approximately 7.4% during intensive workloads, it successfully validates the feasibility of the approach. The findings demonstrate that integrating temporal awareness with embedded metadata significantly improves the accuracy and completeness of provenance tracking, offering a robust foundation for accountability in dynamic environments.

Acknowledgements

This PhD has tested endurance and resolve in ways few things have, yet nothing compares to the difficulty of being far from family. To my parents: thank you for the life you built for me through your quiet sacrifices and support. The values you instilled and the love you gave have carried me further than any academic training ever could. Not a day passes without thinking of you and all you have done to make this possible.

Daniel Roodt, during the low points I've often questioned this whole path. The doubts, the frustration, all the complicated emotions. Sometimes it's hard not to wish life were simpler. What has made it bearable is knowing I've found a true friend. This PhD, and life itself, would have felt very different without you. I know I haven't always been the friend you deserved. I'm proud of what you have achieved, and of what we have managed to achieve together.

To my CROW lab mates (Atthapan, Chris, Joshua, Abigail). Thank you for all the laughter and for enduring the hard times together. For the pranks, the invisible magic cup, and all the small acts that somehow made things feel lighter. Good memories, and ones that I will carry with me forever.

Dr. Alvin Yeo, you have no idea how much impact those simple words you uttered to me each time we passed each other had: "nearly there, 加油!". They landed when I needed them the most. Thank you.

To my supervisors — Dr. Vimal Kumar, Professor David Bainbridge and Dr. Panos Patros. Thank you for your input, guidance, and wisdom throughout this process. Your patience, critical insights, and willingness to challenge my thinking have shaped this work in ways I couldn't have managed alone.

And finally, all praise be to God.

To my family.

List of Figures

2.1 Layered architecture of the Open Provenance Model [1]. 24
2.2 Example: System call interception of *read()* in VFS. 31
2.3 Example: LSM hooks in provenance capture during file read operation. 34
2.4 High-level architecture of eBPF based tracing and monitoring 35
2.5 Position of a stackable file system in the kernel. 37

3.1 PASSv2 architecture overview [2]. 52
3.2 StoryBook architecture overview [3]. 55
3.3 SPADE architecture overview [4]. 62
3.4 S2Logger architecture overview [5]. 69
3.5 Progger architecture overview [6]. 73
3.6 LPM architecture overview [7]. 76
3.7 CamFlow architecture overview [8]. 81
3.8 Example: Progger 3 JSON logs [9]. 84

5.1 Comparison of provenance representation with and without temporal awareness. 127

6.1 Example: Metadata extraction from a digital image using *exiftool*. 144
6.2 TIFF image format [10]. 145
6.3 Hexadecimal representation of a GeoTIFF file. 150
6.4 ExifTool interpretation of a GeoTIFF file. 151
6.5 Examination of ADS metadata and Zone.Identifier. 161

7.1 Prototype architecture overview. 177
7.2 Illustration of version blocks. 183
7.3 Example of bitmap representation of a file. 184
7.4 Logical structure of provenance file. 185
7.5 Example physical allocation. 185
7.6 Class Diagram of the Provenance Access Interface. 188
7.7 Graph view reconstructed by the provenance explorer from embedded provenance records. 213

7.8 Low-level inspection view of a file with embedded provenance at version 40. 214
7.9 Low-level inspection view of a file with embedded provenance at version 41. 214
7.10 End-to-end prototype workflow. 217

8.1 File size growth (log scale). 235
8.2 Internal fragmentation over workload. 238
8.3 External fragmentation proxy over workload. 239
8.4 Version chain locality over workload. 241
8.5 Reconstruction overhead vs ideal contiguous. 242

List of Tables

1.1 Mapping of Design Science Research Methodology (DSRM) activities to their application, thesis chapters, and research questions. 17

2.1 Variations in the definition of data provenance across domains. 19

7.1 *fuse_operation* to PROV-DM mapping. 196

8.1 Performance gain (buffered vs. direct I/O throughput) for selected operations. 227
8.2 Quantified FUSE overhead relative to *ext4* baseline across record sizes (KiB/s and %). 229
8.3 Average throughput (KiB/s). 230
8.4 Version data table. 233
8.5 Reconstruction and verification results under adversarial delta modification. 257

Contents

1 Introduction **1**

1.1 Typologies of Data Provenance 4
1.2 Gaps in Existing Provenance Systems 5
1.3 Scope 7
1.4 Research Goals 8
1.5 Hypothesis and Research Questions 9
1.6 Research Contributions 11
1.7 Thesis Structure 12
1.8 Research Methodology 14

2 Background **18**

2.1 Data Lineage 21
2.2 Thesis Definition 22
2.3 Provenance Models 23
2.3.1 Open Provenance Model (OPM) 24
2.3.2 W3C PROV 25
2.4 Provenance Capture 27
2.4.1 Kernel Space Instrumentation 28
2.4.2 User Space Monitoring and Other Methods 38
2.5 Provenance Storage 39
2.6 Provenance Sharing and Interoperability 40
2.7 Provenance Reconstruction 40
2.8 Provenance Lifecycle 42
2.9 Isn't This... (Related Concepts) 42
2.9.1 Logging 43
2.9.2 Version Control 44
2.10 Conclusion 45

3 Related Work **47**

3.1 PASS 47
3.2 PASSv2 51
3.3 Story Book 54

3.4 Flogger . . . 58
3.5 SPADE . . . 60
3.6 Hi-Fi . . . 65
3.7 S2Logger . . . 68
3.8 Progger . . . 72
3.9 Linux Provenance Modules (LPM) . . . 75
3.10 CamFlow . . . 79
3.11 Progger 3 . . . 83
3.12 A Word About eBPF . . . 86
3.13 Discussion & Conclusion . . . 87

4 Mainstream Adoption of Provenance Systems 90
4.1 Methodology . . . 91
4.1.1 Literature Search . . . 91
4.1.2 Literature Selection . . . 93
4.1.3 Quality Assessment of Grey Literature . . . 94
4.2 Compliance, Security and Data Provenance . . . 95
4.2.1 General Data Protection Regulation . . . 95
4.2.2 Health Insurance Portability and Accountability Act . . . 99
4.2.3 Sarbanes Oxley Act . . . 101
4.2.4 New Zealand Information Security Manual . . . 103
4.3 Software Supply Chain . . . 105
4.3.1 Google Cloud Build Provenance . . . 108
4.3.2 DevOps in Amazon Web Services . . . 109
4.4 Coalition for Content Provenance and Authenticity . . . 111
4.5 Cybersecurity . . . 114
4.6 Conclusion . . . 116

5 Temporal Data Evolution 118
5.1 Motivating Example . . . 118
5.2 Versioning . . . 122
5.2.1 Semantic Versioning (SemVer) . . . 123
5.2.2 The Need for Versioning . . . 126
5.2.3 Versioning Semantics: Snapshot vs. Derivation . . . 129
5.2.4 Binary Equivalence and Artefact Identity . . . 130
5.2.5 Related Work: Elephant File System . . . 131
5.3 Delta Encoding . . . 133
5.4 Conclusion . . . 138

6 Embedding Provenance **140**
6.1 Metadata . . . 141
6.1.1 Example: Exchangeable Image File Format . . . 144
6.1.2 Example: Flexible Image Transport System . . . 147
6.1.3 Example: GeoTIFF . . . 149
6.1.4 Example: Web Metadata . . . 152
6.2 The Case for Embedding Provenance . . . 154
6.3 Techniques for Embedding Provenance . . . 156
6.3.1 Custom File Structure . . . 157
6.3.2 Extended File Attributes . . . 158
6.3.3 Alternate Data Streams . . . 161
6.3.4 i-node Modification . . . 163
6.4 Conclusion . . . 165

7 Design & Implementation **168**
7.1 Requirements and Design Goals . . . 169
7.1.1 General Requirements for Provenance System . . . 169
7.1.2 Requirements for Temporal Awareness . . . 171
7.1.3 Requirements for Embedding Provenance . . . 172
7.2 Design Principles . . . 173
7.2.1 Principle 1: Everything is File . . . 173
7.2.2 Principle 2: Atomic unit of Provenance . . . 174
7.2.3 Principle 3: Lazy Evaluation . . . 175
7.3 Architecture Overview . . . 176
7.3.1 Delta Extraction Component . . . 177
7.3.2 Provenance Model Component . . . 178
7.3.3 Provenance Embedding Component . . . 181
7.3.4 Provenance Access Interface . . . 186
7.3.5 Provenance Explorer Component . . . 189
7.4 Prototype Implementation . . . 192
7.4.1 Filesystem in Userspace (FUSE) . . . 192
7.4.2 fuse_operation . . . 193
7.4.3 Write Handler . . . 197
7.4.4 libdelta . . . 199
7.4.5 Provenance Model . . . 201
7.4.6 Block-based embedding . . . 203
7.4.7 Provenance Access Interface . . . 210
7.4.8 Provenance Explorer . . . 210
7.4.9 Demonstration of Provenance Explorer . . . 213
7.5 End-to-End Prototype Workflow . . . 216

7.5.1 Provenance Capture Workflow 218
7.5.2 Provenance Exploration and Reconstruction Workflow 219
7.5.3 Integration of the Components 220
7.6 Conclusion 220

8 Evaluation 222
8.1 Setup & Methodology 223
8.1.1 Controlling Cache Effects 223
8.2 Micro-benchmark 224
8.2.1 IOzone 225
8.2.2 Cache Effects 226
8.2.3 FUSE Overhead 228
8.2.4 Throughput Analysis 230
8.3 Simulating File Ageing 230
8.3.1 Replay through the capture layer 232
8.3.2 File Growth 232
8.3.3 Fragmentation 236
8.4 Provenance Query Cost Analysis 244
8.4.1 Model and Assumptions 244
8.4.2 Cost of Reconstructing State at Depth n 245
8.4.3 Cost of Querying Provenance for a Single Entity 246
8.4.4 Sensitivity to Delta Chain Length 246
8.5 Impact of Global Indexing 247
8.5.1 Indexing Model 247
8.5.2 Entity-Scoped Queries 248
8.5.3 Temporal Queries 249
8.5.4 Reconstruction Cost with Indexing 249
8.5.5 Sensitivity to Delta Chain Length 249
8.5.6 Cost of Maintaining the Index 250
8.6 Cross-File Provenance Evaluation 250
8.6.1 Motivation and Scope 250
8.6.2 Workload Description 251
8.6.3 Evaluation Methodology 251
8.6.4 Results 252
8.7 Adversarial Modelling 253
8.7.1 Adversarial Modification Scenario 255
8.7.2 Results 256
8.8 Failure Oriented Evaluation 256
8.8.1 Provenance Invariants Revisited 257
8.8.2 Interrupted Writes During Provenance Append 258

8.8.3 Partial Provenance Corruption 259
8.8.4 Out-of-Order Replay Attempts 259
8.9 Threats to Validity 260
8.9.1 Internal Validity 260
8.9.2 External Validity 262
8.10 Conclusion 265

9 Conclusion & Future Directions 267
9.1 Reflection on Research Questions 269
9.2 Limitations 271
9.3 Future Directions 272
9.3.1 LLM-Assisted Provenance Analysis and Security 274
9.4 Final Remarks 275

A Supplementary Results: Retention vs. Delta Versioning 276

B Implementation of Git-Based File Reconstruction Utility 284

Chapter 1

Introduction

Every object, every idea, and every piece of information has a story, a path that traces its origins, transformations, and journey through time. Whether it is a priceless painting with a well-documented chain of ownership, an historical manuscript annotated by generations of scholars, or even the ingredients in a meal traced back to their sources, provenance gives meaning, trust, and context to what we encounter. It is how we distinguish the authentic from the counterfeit, the reliable from the dubious, and the original from the derivative. In a world increasingly driven by data, understanding where information comes from, how it changes, and who influences it is more critical than ever.

Just as art collectors and historians rely on provenance to verify authenticity, the growing reliance on information systems is driving demand for data provenance, to ensure integrity, reliability, and accountability [11, 12, 13]. Data is constantly being created, shared, modified, and analysed often by multiple systems and stakeholders. Without a clear record of its origins and transformations, data can be misinterpreted, misused, or even manipulated. The field of data provenance seeks to address this challenge by capturing the history of data: where it comes from, how it evolves, and how it is used. From scientific research [14, 15] to cybersecurity [16], financial transactions [17] to artificial intelligence [18, 19], the ability to trace and verify data is fundamental to trust and decision-making in the digital age.

The absence of provenance has consequences that extend beyond inconvenience. Without reliable records of where data originated, how it has been transformed, and which systems or users influenced it, we lose the ability to explain or verify decisions [20, 21]. In cybersecurity, incomplete provenance makes it significantly more difficult to reconstruct attack paths, identify compromised components, or determine the scope of an intrusion [22, 23, 24]. In scientific research [25, 26] and artificial intelligence [27, 28], the inability to trace datasets, preprocessing steps, or model inputs undermines reproducibility and confidence in results. Across these domains, the absence of provenance increases the risk that errors, malicious modifications, or misinformation remain undetected until they have already influenced critical decisions.

Consider this scenario. Sonia, a civil servant in the Ministry for the Environment, receives a terse message from her director:

> *"Urgent. Why are emissions from the Waikato district 8% higher than forecast in Q2? Minister wants an answer by 12pm."*

She glances at the clock. It's 10:17am.

She opens the dashboard. The numbers are there, a rise in particulate matter and CO_2, just enough to raise a red flag. The underlying model was trained on regional data pulled from sensors, factory logs, and shipping manifests merged, cleaned, and aggregated across multiple government databases. All of it fed into a predictive model to guide environmental interventions. But now, that number is wrong. Or at least, unexpected. What possible reason could there be?

She starts poking around. The model version hasn't changed. The ingest scripts haven't changed. The dashboard software hasn't changed. Yet somehow, the inputs are skewed. A colleague suggests one of the factory sensors might have gone rogue. Another thinks it might be a recent software patch. Someone floats the idea that a data cleaner script might have dropped a few thousand records due to a library update. But it's all speculation.

There's no audit trail. There are timestamps, but they don't explain causal-

ity. There are commit logs, but they lack semantic intent. No one can tell which piece of software transformed the input, when, or why. Every trace of what happened to that data, where it came from, how it changed hands, what it became, is either implicit, lost in translation, or never recorded at all. With the digital tools at her disposal, Sonia is unable to provide an adequate explanation for the minister in the time available.

Now imagine a system where that answer is not speculative. Imagine Sonia clicks on the Q2 emissions value and drills into its provenance, a causal trace not just of the final figure, but of every intermediate transformation. She sees that the shipping manifest parser was patched in early April to comply with a new API schema. She sees that this patch reclassified certain types of diesel shipments as non-emitting, because of a type mismatch in a lookup table. She sees that the faulty classification slipped into the regional aggregates, and consequently into the forecast.

More importantly, she sees the patch ID, the author, the exact line of code. She reverts it, retrains the forecast model with corrected inputs, and reruns the dashboard. The emission spike disappears. By 11:47am, her reply is in the Minister's inbox.

This imagined scenario is not merely a speculative exercise or a personal vision. It reflects the broader aspirations of the data provenance research community. The ambition to embed provenance deeply into our data lifecycle, decision-making processes, and accountability mechanisms is shared across the field. Indeed, more concrete and attainable versions of this vision have been articulated, such as Pasquier et al.'s *Provtopia* [29] which situates provenance at the heart of distributed systems to enable compliance, security, and transparency in the management of sensitive and regulated data.

1.1 Typologies of Data Provenance

The concept of data provenance has been fractured across research communities, with each adopting its own perspective on what constitutes provenance and how it should be classified. Consequently, there is no single, universally accepted taxonomy. Instead, a set of overlapping typologies has emerged, each grounded in the priorities of its host domain. These can be organised along four principal axes: (i) abstraction, (ii) capture mechanism, (iii) semantics, and (iv) intent.

Abstraction level. At the highest layer lies application-level provenance, in which applications or workflow engines explicitly record the steps taken to derive a dataset. This form is prevalent in scientific workflow management systems [30] and database engines [31, 32], where operators and transformation stages can be directly annotated. At the opposite end is whole-system provenance [8, 33], which captures fine-grained causal relationships at the operating system level, often through system call tracing or kernel instrumentation. Between these extremes are domain-specific and middleware approaches which expose provenance at intermediate levels of abstraction, for example, workflow provenance in computational biology pipelines, which balance generality against granularity.

Capture mechanism. Provenance systems can be further distinguished by how they acquire information. Annotation-based systems [34, 35] depend upon applications or users to emit provenance metadata deliberately, whereas observational systems infer provenance by monitoring execution [36], typically by tracing system events or database queries. Hybrids of these two approaches exist, combining explicit semantic detail with low-level observational completeness.

Semantics. Database research has long categorised provenance according to explanatory role [37]. Where-provenance denotes the source locations from which a data item originates; Why-provenance captures the subset of inputs that justify the presence of that item in a result; and how-provenance expresses

the full derivation process, including operators and transformation functions. These distinctions, although originating in relational theory, have informed broader work on provenance representation, influencing the design of general-purpose models such as W3C PROV [38].

Intent and use case. Finally, provenance is often differentiated by its functional objective. In scientific reproducibility, provenance ensures that experiments can be validated and repeated [39, 40]. In security and forensics, whole-system provenance is deployed to provide tamper-evident audit trails and support intrusion analysis [41]. In compliance and governance, provenance demonstrates adherence to regulatory requirements such as General Data Protection Regulation (GDPR) [42, 43] or Health Insurance Portability and Accountability Act (HIPAA) [44]. Although technically anchored by the same representational frameworks, these use cases drive varying requirements for scope, scale, and trust guarantees.

Together, these typologies demonstrate that "types of provenance" is not a singular classification but a multi-dimensional design space. Database researchers may count three types, workflow researchers emphasise the distinction between prospective (planned) and retrospective (executed) provenance, and systems researchers contrast whole-system with application-level approaches.

1.2 Gaps in Existing Provenance Systems

As will be shown in the literature review, data provenance has been studied extensively and numerous systems have been proposed, real-world adoption however remains limited [45, 46]. The obstacles are not purely technical in terms of "can provenance be captured?" but practical in terms of "can provenance be used effectively." Two challenges stand out:

The first is accessibility. In most systems, provenance is recorded centrally as part of a global, intertwined record of all activity. While this makes for a

complete history, it also makes it unwieldy. A user wishing to ask a seemingly simple question "why did this data value change?" cannot easily isolate just the relevant record. Instead, using the query tools currently available, a large amount of data provenance records are typically returned, requiring significant filtering and interpretation.

This is not merely a usability nuisance. Further complicating matters, provenance often contains information about other processes or datasets, especially in a multi-tenanted environment, some of which may be more sensitive than the data under investigation [47]. As a result, providing provenance to end users risks revealing details that cannot, or should not, be disclosed, leading to a cautious withholding of provenance data altogether.

The second is the gap between what systems claim to reveal and the fidelity of what they actually record. Provenance is often presented as an authoritative record of what happened to a piece of data, but in practice it usually stops short of describing the precise nature of the transformation. Consider a process that processes a file via a dynamically linked library. If that library is temporarily replaced with a malicious variant, used to manipulate the file, and then restored, many provenance systems can at best record that the library changed at some point and may have influenced the file. They cannot show the exact modifications made while the malicious version was in use. The issue here is that such systems do not track the semantic differences between those versions, meaning the true nature of the change is lost.

These are not incidental shortcomings; they are manifestations of deeper conceptual constraints in how provenance is modelled, recorded, and disseminated. This thesis addresses these constraints directly, not by producing yet another provenance system for its own sake, but by introducing a concept designed to overcome these barriers. This thesis addresses them through a temporally aware, embedded approach to provenance. Recording version specific changes preserves the precise nature of data transformations, while associating provenance with the data object enables users to access a focused and

portable record without retrieving an extensive global provenance graph. Together, these mechanisms reduce the loss of fidelity, complexity, and disclosure risks that limit the practical use of existing provenance systems. The proposed approach is designed for incorporation into existing systems, providing a more accessible and practical foundation for provenance-based use cases.

1.3 Scope

The central contribution of this thesis is the introduction of a concept that directly addresses the barriers to the adoption and usability of provenance. The aim is to provide a framework that can be embedded into the design of existing provenance systems.

To this end, the thesis deliberately centres on system-level provenance. This focus is driven by two considerations. First, system provenance offers the most application-agnostic foundation, capable of retrofitting into diverse domains and workflows without requiring assumptions about their semantics. Second, the past decade has seen substantial advances in whole-system provenance research [8, 33, 4], creating a rich body of work against which the proposed contributions can be situated and evaluated.

The focus here is on recording low-level interactions, including system calls, process executions, file modifications and data dependencies. This research addresses the core challenge of capturing how data is generated, transformed, and utilised within a system, independent of its application context. Such tracking enables users to reconstruct execution histories, audit processes for compliance or error diagnosis, and understand data origins without relying on domain-specific descriptive information. This foundational capability serves as a neutral platform upon which specialised provenance systems could build their higher-level annotations or application-specific interpretations.

As to scope, the thesis proposes and evaluates a system architecture that, while inherently capable of interoperating with domain-specific extensions via

well-defined APIs, does not itself explore their implementation. The emphasis remains on creating a robust, scalable infrastructure that reliably captures system-wide and file-level provenance traces, leaving domain-specific integrations to future work or complementary studies. This deliberate boundary ensures the research addresses the underlying technical challenges of whole-system provenance without being constrained by the complexities of any single application area.

1.4 Research Goals

The primary goal of this research is to develop a unified provenance framework that integrates temporal awareness with embedded metadata to enhance the accuracy, completeness, and scalability of lineage tracking in dynamic enterprise environments. This objective directly addresses the hypothesis by synthesising technical innovations from prior work into a cohesive system capable of reconstructing workflows, tracing dependencies across time, and enabling audits at any historical point.

The secondary goals focus on foundational investigations: first, identifying real-world drivers, barriers, and technological enablers influencing provenance adoption to bridge theory and practice; second, addressing critical limitations in existing static systems by analysing how temporal data evolution compromises lineage accuracy and exploring storage-efficient methodologies for versioned records; third, evaluating the feasibility of embedding provenance metadata directly within data artifacts as an alternative to centralised repositories. These secondary objectives collectively inform and validate the design choices of the primary framework, ensuring its practicality, scalability, and alignment with enterprise needs.

1.5 Hypothesis and Research Questions

The central hypothesis of this thesis is as follows:

> A unified provenance framework integrating temporal awareness with embedded metadata enhances the accuracy and completeness of provenance tracking.

To investigate this hypothesis, the thesis is structured around the following research questions (RQ):

RQ1: What are the drivers, barriers, and technological enablers influencing data provenance adoption in enterprise environments?

Despite decades of research on data provenance, its real-world application remains limited. In Chapter 4, we (i) quantify the current extent of provenance adoption across industries, (ii) identify theoretical advantages versus practical limitations of existing systems, and (iii) model factors that hinder or promote adoption in enterprise settings. This foundational RQ bridges theory and practice by uncovering gaps between research potential and real-world implementation, informing future design priorities.

RQ2: How do temporal data evolution and static provenance systems compromise lineage accuracy in dynamic environments?

Current provenance approaches assume static relationships between data objects. In Chapter 5, we demonstrate how neglecting temporal changes (e.g., versioning, updates) obscures critical differences in lineage, leading to incomplete or misleading provenance. We propose a framework for dynamic provenance that accounts for time-based dependencies and evolutionary data states. This question critiques existing methodologies and justifies the need for temporally aware systems, setting the stage for technical innovations in RQ 3-5.

RQ3: What storage and versioning strategies balance storage efficiency and temporal accuracy in provenance records?

Capturing temporal data evolution demands scalable solutions. Chapter 5 investigates delta encoding techniques to optimise storage while preserving precise temporal resolution. Evaluate trade-offs between compactness, query performance, and historical fidelity through experimental validation. This question addresses a core technical challenge identified in RQ2, advancing provenance systems toward practical scalability.

RQ4: What are the feasibility and implications of embedding provenance metadata directly within data artifacts versus centralised repositories?

Most systems decouple provenance from data, centralising it in databases, a design with privacy, accessibility, and fragmentation risks. In Chapter 6, we compare a decentralised approach, which is to embed provenance as metadata to centralised solutions across dimensions like scalability, security, interoperability, and usability. A prototype was built to demonstrate advantages/disadvantages of embedded provenance. This question challenges conventional architectures, proposing a novel design paradigm for provenance storage.

RQ5: How do temporally aware, embedded provenance systems enable scalable lineage tracing while preserving auditability from arbitrary historical points?

Integration of RQ2's temporal awareness and RQ4's metadata embedding creates opportunities for holistic solutions. In Chapter 7, we synthesise findings from prior chapters to design a unified framework that combines delta-encoded versioning with embedded provenance. We demonstrate its ability to reconstruct workflows, trace dependencies across time, and support audits at any

point in an artifact's lifecycle. This final RQ unites technical contributions (RQ2-4) into a cohesive system, validating the feasibility of scalable, audit-ready provenance for enterprise use.

1.6 Research Contributions

The main contribution of this research is the introduction of a data provenance framework that embeds provenance metadata directly within files themselves by treating provenance as an integral component of every file rather than maintaining it externally or in siloed systems. The proposed approach ensures seamless integration with existing workflows and reduces fragmentation inherent in current solutions. This design enables users to access, audit, and manage provenance without requiring specialised tools or additional infrastructure, fostering broader adoption across diverse applications.

Building on the foundational principle of Linux's "everything is a file" paradigm, this work introduces separation of concerns into provenance management. Provenance metadata is decoupled from application-specific logic while remaining inherently tied to each object (e.g., files, directories, or system entities). This architecture ensures modularity and scalability whereby provenance can be queried, modified, or audited independently without interfering with the object's primary function. Such a design contrasts sharply with existing systems that often conflate provenance tracking with application layers, leading to inefficiencies and inflexibility.

A third key contribution is the integration of temporal awareness into provenance tracking. Current systems typically record static derivation relationships (e.g., object A was derived from object B) but lack mechanisms to capture which version or state of B existed at the time of derivation. This work addresses this gap by embedding timestamps and version identifiers within provenance records, enabling precise temporal queries. For instance, users can now determine not only that a dataset originated from a specific source

but also isolate the exact state of that source at the moment of derivation, a critical capability for reproducibility, accountability, or auditing in dynamic environments.

To validate these concepts, this research includes a proof-of-concept system built atop a Linux-based file system. The implementation demonstrates feasibility by embedding provenance as extended attributes within files, leveraging existing kernel mechanisms to ensure low overhead and compatibility with standard workflows. Performance evaluations confirm that the approach incurs minimal runtime penalties while enabling efficient querying of lineage and temporal metadata. This prototype serves as both a validation tool for theoretical contributions and a blueprint for future scalable deployments.

1.7 Thesis Structure

This thesis is structured to systematically build the case for rethinking how data provenance should be managed, specifically arguing for the importance of versioning and the embedding of provenance within the data itself. The document is organised into nine chapters, each contributing progressively towards this argument and the demonstration of a practical prototype embodying these principles.

Having established the motivation and scope for the work in this chapter, the latter sections define the hypotheses, articulate the goals, and highlight the key contributions. Following this, Chapter 2 presents the necessary background to situate the work within the broader research landscape. It introduces fundamental concepts surrounding data provenance, including definitions, methods for capture, storage, reconstruction, sharing, interoperability, and lifecycle management. It also examines closely related concepts and prevailing models in the field, equipping the reader with the knowledge required to engage with the discussions in later chapters.

Chapter 3 surveys related work, presenting a series of general-purpose

provenance systems and offering a historical perspective through a timeline of developments. This chapter frames the progress and limitations of existing approaches, setting the stage for the argument that current methods are insufficient.

Chapter 4 investigates the real-world adoption of provenance systems, or rather the noticeable lack thereof. Despite significant research output, the chapter demonstrates that mainstream uptake across regulatory frameworks, software supply chains, creative industries, and DevOps practices remains minimal. It speculates that this limited adoption stems from fundamental shortcomings in existing systems, particularly the absence of versioning capabilities and the persistent disassociation between provenance and the data it describes.

Chapters 5 and 6 form the core of the thesis's conceptual argument. Chapter 5 focuses on the need for versioning in data provenance, hereafter referred to as temporal data provenance. The chapter defines the problem, illustrates the practical implications, and discusses potential techniques to enable versioned provenance information. Chapter 6 addresses the second major theme: the necessity of a stronger, more intrinsic bond between provenance metadata and the data object itself. Drawing parallels with other well-accepted metadata practices. It argues that embedding provenance directly into the data is both logical and advantageous, and explores how this can be achieved through practical techniques.

Chapter 7 introduces the prototype system developed to embody these two ideas. It details the design principles of the system and the specifics of its implementation, demonstrating the feasibility of the proposed approach.

Chapter 8 evaluates the prototype through a combination of microbenchmarking and macrobenchmarking experiments. This evaluation assesses both the technical performance and the broader practical viability of the system in real-world contexts.

Finally, Chapter 9 concludes the thesis by summarising the contributions, reflecting on the findings, and discussing avenues for future research. In do-

ing so, it closes the loop from the initial motivations to practical realisation and outlines the broader implications of this research for the future of data provenance systems.

1.8 Research Methodology

This research adopts the Design Science Research Methodology (DSRM) proposed by Peffers et al. [48]. DSRM provides a structured approach to identifying research problems, developing artefacts to address them, and evaluating their effectiveness. The research follows the six activities of DSRM: problem identification and motivation, definition of objectives, design and development, demonstration, evaluation, and communication.

The research began with a literature review to establish the theoretical foundations of data provenance and examine existing approaches. Chapter 2 reviews provenance concepts, models, capture mechanisms, storage techniques, and related technologies. Chapter 3 examines existing provenance systems, focusing on their architectures, capabilities, and limitations. The investigation considers how these systems capture, represent, store, and provide access to provenance information. Chapter 4 extends this analysis by examining the adoption of provenance systems across different domains, identifying practical barriers and factors influencing their use. These investigations establish the research problem and motivate the need for an alternative approach, particularly in relation to temporal awareness and the separation of provenance metadata from the data it describes.

The findings from these investigations informed the objectives and requirements of the proposed solution. Chapter 5 examines the limitations of static provenance representations and investigates versioning strategies, including delta encoding, to represent changes in data over time. Chapter 6 explores the feasibility of embedding provenance metadata directly within data artefacts rather than maintaining it exclusively in centralised repositories. These

investigations provide the conceptual basis for the unified provenance framework and inform the system requirements and architectural decisions presented in Chapter 7.

The design and development stage involved implementing a proof-of-concept system to investigate the feasibility of the proposed approach. A Linux file system environment was selected as the experimental setting because it provides an application independent context in which file operations and data transformations can be observed. The prototype was implemented using Filesystem in Userspace (FUSE), allowing provenance capture to be investigated at the file system level without modifying the Linux kernel. As detailed in Chapter 7, the implementation incorporates mechanisms for capturing provenance relationships, recording content-level changes through delta encoding, maintaining historical versions, embedding provenance information, and accessing and reconstructing provenance histories. The prototype serves as an experimental artefact through which the proposed framework can be demonstrated and evaluated.

The demonstration stage establishes the functional feasibility of the proposed approach. Chapter 7 illustrates how the prototype captures provenance during file operations, associates provenance information with successive data versions, and supports the reconstruction and exploration of historical provenance records. These demonstrations provide evidence that the proposed mechanisms can be integrated within a working system and exercised through file system operations and workflows.

The evaluation stage, presented in Chapter 8, uses controlled experiments and analytical methods to investigate the performance, feasibility, and limitations of the prototype. Experimental data are collected through file system benchmarks, file-ageing experiments, provenance queries, and larger workloads involving cross-file dependencies. Additional experiments examine the integrity of historical reconstruction and the behaviour of the system under adversarial and failure conditions. The resulting measurements and obser-

vations are analysed to characterise performance overhead, storage growth, reconstruction costs, query behaviour, and provenance integrity. Where applicable, results are compared against baseline behaviour and interpreted in relation to the design objectives and requirements established earlier in the research. The evaluation focuses on the feasibility and implications of the proposed provenance architecture rather than establishing the prototype as a production ready system.

Finally, the research findings are communicated through the thesis, with Chapter 9 synthesising the contributions, discussing the limitations of the proposed approach, and identifying directions for future research. Table 1.1 summarises how the six DSRM activities are applied throughout the research and maps them to the corresponding thesis chapters and research questions.

DSRM Activity	Application	Chapter	RQ
Problem identification and motivation	Establish the state of provenance research, examine existing systems and their adoption, and identify limitations associated with temporal representation, accessibility, and the separation of provenance from data.	2–4	RQ1, RQ2
Define objectives for a solution	Establish requirements for temporal provenance, efficient historical representation, embedded provenance, portability, reconstructability, integrity, and practical system operation.	5–7	RQ2, RQ3, RQ4
Design and development	Design the unified provenance framework and implement a FUSE-based prototype in a Linux file system environment, selected to investigate provenance capture independently of application-specific requirements. Implement delta representation, provenance modelling, embedded storage, and provenance access and exploration.	7	RQ3, RQ4, RQ5
Demonstration	Exercise the prototype through representative file operations and workflows to demonstrate provenance capture, embedding, historical reconstruction, validation, and exploration.	7	RQ4, RQ5
Evaluation	Conduct experiments to measure performance, storage growth, reconstruction costs, query behaviour, cross-file provenance, integrity, and failure handling. Analyse the results against the framework's design objectives and identify its limitations.	8	RQ3, RQ4, RQ5
Communication	Present the research problem, proposed artefact, experimental evaluation, findings, limitations, and resulting design knowledge through the thesis and its conclusions.	1–9	RQ1–RQ5

Table 1.1: Mapping of Design Science Research Methodology (DSRM) activities to their application, thesis chapters, and research questions.

Chapter 2

Background

Provenance, as a concept, is inherently intuitive. It refers to the origin, history, and lineage of an entity, whether it be data, an artifact, or a process. However, across different fields, domains, and use cases, the interpretation of provenance often varies, leading to differences in its definition. In provenance research, diverging research perspectives and goals not only drive variation but can also shape interpretation, at times blurring clarity or reframing representation to align with specific narratives.

To illustrate this ambiguity, Table 2.1 presents a comparative selection of definitions drawn from a range of domains including biomedical research, computer science, healthcare, cybersecurity, and environmental science. Each definition captures a facet of provenance, whether it be transformation, ownership, or process history, but there is little consensus on what constitutes the core of the concept. Some disciplines treat provenance primarily as a type of *metadata*, that is, data about data; while others as a historical record, while others embed it within broader notions of quality assessment, trust, or policy tracking.

The similarities are easy to spot. Across all domains, provenance is defined as a record or metadata that describes "where data come from" and "what has happened to them." Terms such as *origin*, *history*, *lineage*, *derivation*, and *source* appear consistently. Furthermore, there is an acknowledgement

Definition	Domain
Data provenance, a record that describes the origins and processing of data. [18]	Artificial Intelligence
Data provenance refers to the origin, processing, and movement of data. [15]	Biomedical
Provenance information tracks and documents the origin, owner ship, processing, and custody of data throughout their life cycle. [49]	Clinical Research
Data Provenance is the record of the parts, inputs, systems and processes that affect the data collected and offer historical context. [12]	Cloud Forensics
Data provenance, one kind of metadata, pertains to the derivation history of a data product starting from its original sources. [14]	Computer Sci.
Data Provenance refers to the "origin," "lineage," and "source" of data. [50]	Computer Sci.
Provenance is information about entities, activities, and people involved in producing a piece of data or thing, which can be used to form assessments about its quality, reliability or trustworthiness. [38]	Computer Sci.
Data provenance refers to a record trail representing an event's origin and explaining how and why it got to the current state. [24]	Cybersecurity
Provenance information corresponds to essential metadata that describes the entities, users, and processes involved in the history and evolution of a data object. [11]	Cybersecurity
Data provenance is information about where data come from (provenance data) and how they transform (provenance transformation). [32]	Database
Provenance is a record that describes the people, organisations, entities, and activities involved in producing, influencing, or delivering a policy, objective, obligation, action, or any outcome in an environmental management system. [51]	Environmental Sci.
Data provenance means recording data origins and the history of data generation and processing. [52]	Healthcare
Provenance refers to any amount of information, comprising all the elements and their relationships, which contribute to the existence of given data. [53]	Healthcare
Data provenance is a concept that covers the origin of the data, the changes made, and the processes it has gone through. [54]	Internet of Things
Provenance refers to the origin of an object or captures information about the creation or evolution process of an object. [55]	Scientific Workflow

Table 2.1: Variations in the definition of data provenance across domains.

that provenance serves not only as an archival trace but also as a mechanism to assess trustworthiness, reliability, or quality. In both technical (computer science, cybersecurity) and applied (healthcare, environmental science) contexts, provenance is framed as essential for validating the integrity of data.

The differences, however, reveal how provenance is conceptualised to meet domain-specific needs. In technical and infrastructural domains such as computer science, databases, and cloud forensics, the definitions lean towards precision in describing systems, processes, inputs, and transformations. This framing treats provenance as a structural metadata record, used for debugging, reproducibility, and audit trails. In biomedical and healthcare contexts, the emphasis shifts towards ownership, custody, and lifecycle tracking, reflecting regulatory compliance requirements, patient data governance, and ethical considerations. Cybersecurity definitions adopt a forensic tone, emphasising event trails, entities, and the reconstruction of *how* and *why* an event occurred, aligning provenance with intrusion detection, accountability, and incident response. Environmental science extends the concept beyond data in a strict sense, using provenance to trace the origin and influence of policies, objectives, and outcomes, demonstrating its applicability to decision-making and governance processes.

Another point of divergence lies in granularity. Some definitions, for example, in database or scientific workflow contexts, are concise and focus narrowly on data products and transformations, whereas others such as healthcare or environmental science are more expansive, incorporating relationships between entities, organisations, and people. This suggests that the more interdisciplinary or policy-driven a field, the broader and more socially contextualised its notion of provenance becomes.

The abundance of variants reflects healthy interdisciplinarity, but it also introduces conceptual noise that can stifle standardisation, reduce interoperability, and slow the evolution of provenance frameworks. For the work reported in this thesis, a research challenge is to distil the shared core elements while still

accommodating domain-specific extensions, essentially, to architect a layered or modular definition that supports both universality and specialisation.

This chapter establishes the foundation for the thesis by first setting out the formalisation of definition that will be used throughout. Having clarified what is meant by provenance in this context, the discussion turns to the practical question of how it is captured, stored, and queried, since the choice of technique directly shapes both the fidelity of the recorded information and the kinds of questions that can later be asked of it. In practice, provenance is not always readily available, which leads naturally to alternative strategies such as reconstructing provenance from traces or system logs. Once the means of obtaining provenance have been considered, attention shifts to its longer-term management, addressing how provenance evolves, how it should be retained, and what challenges arise across its lifecycle. Finally, the chapter broadens the view to consider concepts such as versioning and logging, which share certain features with provenance but differ in scope and purpose, allowing the boundaries of the thesis to be drawn more clearly.

2.1 Data Lineage

A fundamental distinction, yet often blurred in literature, is between data provenance and data lineage. Though closely related, they are not synonymous.

Data lineage typically refers to a coarse-grained trace of how data moves and transforms across systems. It is usually concerned with high-level operations, such as the sequence of databases, pipelines, or modules that data flows through. Lineage can be thought of as a graph of dependencies showing which inputs produced which outputs, commonly represented in data flow diagrams or Extract, Transform, Load (ETL) pipelines [56, 57].

Data provenance, on the other hand, is a broader and more nuanced concept. It encompasses not only the lineage but also the contextual metadata

that explains the who, what, when, where, and why of data transformations. Provenance captures details such as systems and users involved in producing or modifying a data, functions applied, temporal aspects of changes and the environmental conditions under which data was processed [58].

Thus, lineage can be seen as a structural subset of provenance, necessary but not sufficient for full provenance tracking. In practical systems, the two are often used together [59, 8]. Lineage provides the skeleton of data flow, while provenance adds semantic flesh.

2.2 Thesis Definition

For the purpose of discussing how the work reported here relates to prior work, it becomes necessary to established a standardised, domain appropriate understanding of data provenance. One that is sufficiently general to accommodate varied use cases, yet precise enough to support technical rigour. To that end, we adopt the following generalised definition and supporting terms, which will serve as the foundation for discussion throughout the remainder of this thesis.

Definition 1. *Data Provenance* is the systematic documentation of records[†] to establish relationships[‡] between data entities and their contexts, enabling data consumers to discern[§] the history and derivation of data, with the highest detail allowing data recreation.

Definition 2. *Records*[†] are metadata, logs, and other documented information that provide evidence of data creation, modification, and interactions within a data lifecycle.

Definition 3. *Relationships*[‡] define the connections between records, data entities, and their contexts, illustrating data flow and transformations across a data lifecycle.

Definition 4. *Discernment*[§] is the capacity of data consumers to comprehend provenance information in a manner that allows them to understand the origin, transformations, and context of data.

2.3 Provenance Models

When discussing data provenance, the instinct is often to focus on provenance capture, i.e., what mechanisms can be deployed to record the necessary traces of computation or workflow execution. Yet capture, storage, and query all presuppose something more fundamental, a shared understanding of what provenance is and how it can be represented. Provenance models provide this conceptual foundation. They define the abstractions such as entities, activities, agents, and their interrelations that give structure to provenance information. Without a model, provenance remains a collection of system specific metadata, lacking the semantic coherence required for reuse, interoperability, and reasoning.

Early provenance efforts were typically confined to the systems in which they originated. Workflow engines are systems that coordinate the execution of computational tasks and the flow of data between them. Systems such as Chimera [60], Kepler [61], and Taverna [62] developed their own schemas to record execution histories. For example, a schema might represent each task by its name, inputs, outputs, and execution time, linking output files to the tasks that generated them. Although such records supported workflow monitoring and facilitated reproducibility, differences in provenance representations complicated the exchange of provenance information across systems. The Open Provenance Model (OPM) [63] marked the first coordinated attempt at a community framework.

2.3.1 Open Provenance Model (OPM)

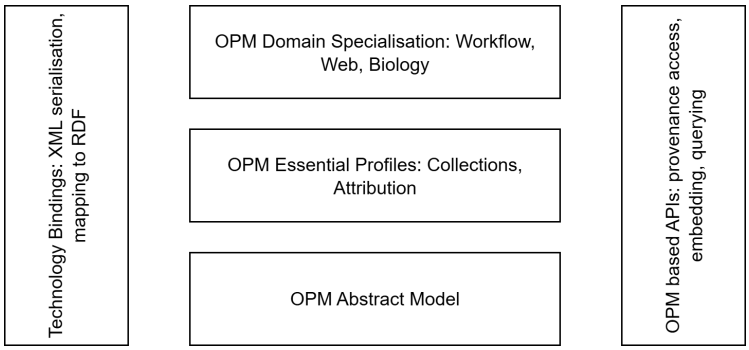


Figure 2.1: Layered architecture of the Open Provenance Model [1].

OPM was designed with the recognition that provenance information must serve a wide range of use cases, from abstract modelling through to practical system integration and domain-specific specialisation. Its architecture is therefore layered, and Figure 2.1 provides a visual representation of this stratification.

The foundation of OPM is the **Abstract Model**, which specifies the core conceptual entities and relationships. This layer captures the essential causal structure of provenance graphs, defining artifacts, processes, and agents, along with the fundamental relations between them. It is deliberately abstract, ensuring that the model remains domain agnostic and extensible across scientific and computational contexts.

Built upon this base are the **OPM Essential Profiles**, which extend the abstract model to provide commonly required features such as collections (groupings of artifacts or processes) and attribution (explicitly linking agents with responsibilities). These profiles enable richer expressiveness without compromising the simplicity of the core.

The next layer encompasses **OPM Domain Specialisation**, where the model is tailored to specific application domains such as scientific workflows, the Web, and biology. In these contexts, domain ontologies and vocabularies align with OPM constructs, supporting interoperability across tools and communities while retaining the semantic rigour of the abstract model.

At the edge of the stack are the mechanisms that allow OPM to be deployed in practice. On the left, **Technology Bindings** translate the abstract concepts into concrete representations, for example via eXtensible Markup Language (XML) serialisations or mappings to Resource Description Framework (RDF). RDF is a standard for describing information and relationships between things on the Web in a structured, machine-readable format. It represents information as simple statements consisting of three parts: a subject, a predicate (relationship), and an object. These bindings are critical for interoperability, ensuring that provenance graphs can be exchanged between heterogeneous systems. On the right, **OPM-based APIs** expose functionality for embedding provenance capture within applications, accessing stored provenance, and executing queries over provenance graphs. Together, bindings and APIs bridge the gap between the theoretical model and practical use in real-world infrastructures.

This layered view illustrates OPM's dual design goals, which is to provide an abstract foundation for reasoning about provenance, while also supporting the engineering realities of implementation, integration, and domain adaptation. Although OPM captured core provenance concepts, its informal semantics often led to inconsistent interpretations across systems [64, 65]. Addressing the ambiguities and limited expressiveness of OPM, the W3C PROV model was established as a formal, extensible standard for representing provenance.

2.3.2 W3C PROV

Building on OPM, the W3C PROV [66] establishes itself as an international standard for capturing and exchanging provenance information across diverse computational ecosystems. Developed through collaborative efforts within the World Wide Web Consortium (W3C), PROV addresses interoperability challenges by providing a unified vocabulary to describe the origins, evolution, and context of digital artifacts. Central to its design are three core conceptual entities. **Entities** (any thing that exists), **Activities** (processes that generate

or modify entities) and **Agents** (entities responsible for initiating activities). These foundational constructs form the basis of PROV's data model (PROV-DM) [67], which systematically defines relationships between these elements through a set of well-defined semantics. For instance, an entity may be generated by an activity (*wasGeneratedBy*), while that activity might have used another entity as input or relied on an agent for execution (*wasAssociatedWith*). The relationship *wasDerivedFrom* , meanwhile, establishes lineage between entities through transformations or modifications, enabling the reconstruction of derivation chains critical to understanding data evolution.

PROV's expressiveness is further augmented by its layered architecture. Beyond PROV-DM, additional components support practical implementation across different systems. PROV defines multiple serialisations, such as PROV-N [68], a human-readable notation, PROV-XML [69], and PROV-JSON [70], that encode provenance graphs in machine-readable forms. PROV-O [71], an Web Ontology Language (OWL2) ontology, translates the abstract model into RDF, enabling integration with semantic web technologies for querying and reasoning over provenance data. This layered design allows flexibility. Users can opt for lightweight serialisations for ease of exchange or leverage the ontology for complex semantic analysis. However, this structure also introduces trade-offs; while PROV's abstraction promotes interoperability, its generic nature may necessitate domain specific extensions to capture nuanced scenarios beyond the core model.

A critical aspect of PROV lies in its ability to balance generality with specificity. By defining a minimal yet comprehensive set of primitives, it avoids over-constraining applications but occasionally risks ambiguity when modelling complex provenance contexts. For example, while *wasDerivedFrom* captures lineage relationships, it does not inherently encode the precise transformations that occurred between entities, requiring additional metadata or external models for richer descriptions. Similarly, accountability and trustworthiness, key concerns in many domains are addressed only indirectly through

agents and attribution mechanisms like *wasAttributedTo*, prompting efforts such as PROV-AQ [72] (Accountability and Quality) to extend the model's scope. Such extensions highlight both PROV's adaptability and its limitations when faced with domain specific requirements, necessitating alignment between provenance capture and application needs.

The adoption of PROV has been notable across scientific computing [73] and data publishing platforms [74], where it provides a common vocabulary for exchanging provenance metadata. In research domains such as computational biology and climate modelling, PROV has been used to document workflows and dataset transformations, supporting reproducibility [75].

Despite these successes, adoption is inconsistent, and many systems use PROV only partially or alongside proprietary extensions. Scaling PROV to meet the demands of modern data intensive pipelines remains challenging [76]. Provenance graphs can grow to massive size, making efficient storage and querying difficult. Reconciling provenance collected at different levels of granularity across heterogeneous systems is an ongoing problem [77]. At the same time, balancing transparency with privacy, exposing enough lineage for accountability while protecting sensitive information, remains an active research area.

2.4 Provenance Capture

To make these models operational, mechanisms are needed to acquire provenance as processes unfold. This shifts the focus from modelling to provenance capture, the capture of relevant and reliable information about the flow, transformation, and usage of data. The manner in which provenance is collected, whether at the kernel, system, or application level, fundamentally shapes its fidelity, performance overhead, and trustworthiness.

Capturing provenance involves observing system-level events such as file reads and writes, inter-process communication and network activity. These

events must be recorded in a manner that retains causal relationships, preserves temporal ordering, and includes sufficient metadata to support provenance use cases. Provenance capture mechanisms therefore face a range of design trade-offs, i.e., completeness versus performance, trust versus ease of deployment, and generality versus specificity.

This section examines established approaches to provenance capture, focusing first on kernel space mechanisms which offer the highest level of visibility and trust, followed by user space techniques which provide greater flexibility and portability but often at the cost of reduced guarantees. These methods are not mutually exclusive and may be used in concert in layered or hybrid architectures.

2.4.1 Kernel Space Instrumentation

Provenance capture in kernel space provides the most comprehensive and authoritative view of system activity. Operating below the user space abstraction boundary, the kernel has full visibility over process execution, file system activity, network communication, and hardware interaction. Provenance captured at this level is typically resistant to tampering by user space adversaries, assuming a trusted kernel.

The primary challenge with kernel space capture is the intrusiveness of the instrumentation. Modifying or extending kernel behaviour for provenance purposes introduces complexity, risks to system stability, and potential performance degradation. They often require administrative privileges for installation and maintenance, which may limit their applicability in production environments or user-facing systems. Nevertheless, the kernel remains the ideal vantage point for whole-system provenance collection, especially in adversarial or high-assurance environments.

When focusing on provenance capture through kernel space instrumentation, there are several distinct ways this can be implemented. The following sections will consider main points of entry for integrating provenance capture

within the kernel, each carrying different implications for performance, security, maintainability, and scope. The purpose here is to set out these options as the foundation for a more detailed discussion of their relative merits.

System Call Interception

System calls form the primary interface through which user space programs interact with the kernel. Intercepting these calls allows for the capture of data flow and control flow events with minimal assumptions about program internals. Provenance capture via system call interception targets key primitives such as *open*, *read*, *write*, *execve*, and *fork*, which together cover most data access and process control semantics.

There are two principal ways to implement system call interception in the Linux kernel: (i) directly modifying the kernel source, or (ii) loading a kernel module or similar mechanism that dynamically hooks into system call paths.

Directly Modifying the Linux Kernel. One approach is to integrate provenance capture logic directly into the Linux kernel by modifying its source code. This yields a high level of control and can ensure tight coupling between provenance logic and the kernel's native mechanisms for event dispatch, scheduling, and memory management.

However, direct kernel modification is brittle and unsustainable outside of controlled research environments. It is generally regarded as an unwise strategy unless the intention is to contribute upstream and secure eventual inclusion within the mainline source tree. The reasons for this are structural, practical, and cultural, and each imposes costs on long-term sustainability.

At the structural level, the Linux kernel is an evolving system in which internal refactoring, Application Programming Interface (API) and Application Binary Interface (ABI) changes, and architectural clean-ups occur continuously across release cycles. Any modification maintained outside of the mainline tree must therefore be manually forward-ported against these changes. This results

in an ongoing maintenance burden, effectively committing the maintainer to rework with every new release. The more invasive the modification, the greater the likelihood of conflicts and regressions.

From a practical standpoint, a private kernel fork complicates deployment and support. Distributions do not integrate such patches, meaning the patched kernel becomes isolated from the tested, security-patched kernels maintained by the community. Security and stability fixes must be backported independently, often without access to the same level of validation infrastructure available upstream. Debugging is also rendered more difficult. Problems encountered on a private fork are unlikely to receive assistance from the wider development community, since the first step in any diagnostic process is to reproduce the issue on an unmodified mainline kernel.

Linux kernel development adopts an "upstream first" ethos, in which long-term sustainability of features depends on their acceptance into the mainline. Patches that are not accepted upstream typically degrade over time as internal interfaces evolve, forcing maintainers into costly rewrites. By bypassing the upstream review and integration process, developers also forgo the benefits of peer review, design critique, and alignment with the kernel's broader architectural direction.

For these reasons, extension mechanisms such as Loadable Kernel Modules (LKM), Linux Security Module (LSM), or increasingly, extended Berkeley Packet Filter (eBPF) have become the preferred method for introducing new functionality, which will be introduced in the subsequent section. These approaches provide insulation from kernel internals, ensure continued compatibility with supported releases, and reduce maintenance overheads. When kernel modification is unavoidable, the only viable path is to design the feature with upstream acceptance in mind and to engage with the established contribution and review process. Any alternative approach represents not only technical debt but a form of debt whose cost compounds as the kernel evolves.

Loadable Kernel Module (LKM). An LKM provides a mechanism for extending the kernel at runtime without requiring a full recompilation or reboot. They are dynamically loadable pieces of code, typically compiled as *.ko* files, that can be inserted into and removed from the kernel as needed. This modularity makes them attractive for research contexts where fine-grained instrumentation of system behaviour is necessary but must be done without destabilising the entire operating system.

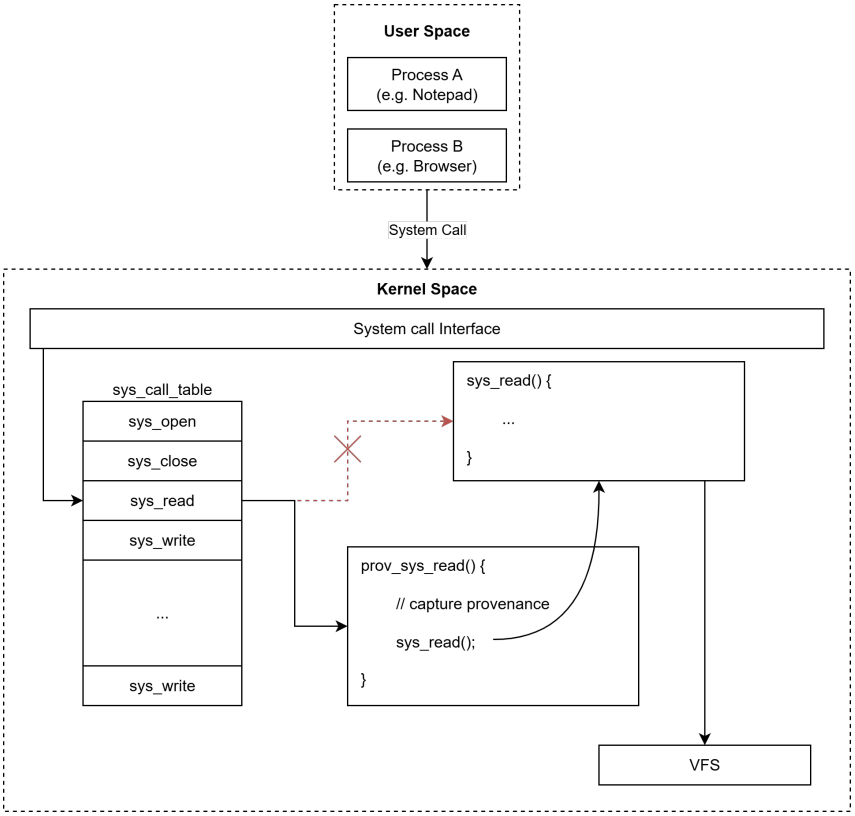


Figure 2.2: Example: System call interception of *read()* in VFS.

LKM have many use cases, ranging from device drivers to system features. They allow the Linux kernel to stay minimal while still offering a broad feature set on demand. For example, hypervisors such as Kernel-based Virtual Machine (KVM) exist entirely as LKMs (*kvm.ko*), combined with vendor-specific modules, so that Linux can transition into a virtualisation host without unnecessary bloat. In provenance capture systems, LKMs are positioned to instrument directly within the kernel's execution path. LKMs interact with the kernel through pre-defined registration hooks. Once loaded, a module can

register callback functions against specific kernel subsystems (e.g. file operations, networking, inter-process communication). Existing kernel hooks can be utilised to record the semantics of an operation at the moment it occurs. The provenance system then serialises this contextual information into its internal graph or log representation.

An example of a LKM-based provenance capture technique is to alter pointers in system call table (*sys_call_table*). The system call table is a kernel data structure that maps *syscall* numbers to the corresponding kernel function pointers. For example, when a process invokes *open()*, the kernel consults the table to determine the address of *sys_open* and dispatches execution accordingly. By altering the pointers in this table, one can redirect execution to alternative implementations, wrappers that record provenance before calling the original function.

Figure 2.2 illustrates an example where a user processes such as a text editor or browser invoke system calls (e.g. *read()*) which are resolved via the system call interface and dispatched to the Virtual File System (VFS). By altering the system call table, the kernel module redirects these calls to a custom handler function, *prov_sys_read()* in this example. The custom function can perform additional tasks such as logging, provenance capture, or access control checks. After completing its own logic, the handler typically invokes the original system call implementation to preserve normal system behaviour.

While effective, system call table modification from an LKM has drawbacks. First, the system call table is no longer exported as a symbol in modern kernels, so modules must rely on heuristic methods to locate it, for example, scanning kernel memory, which introduces fragility across kernel versions. Second, write-protecting mechanisms, such as the *CR0* write-protect bit on x86, must be temporarily disabled to modify entries, which is invasive and increases the risk of destabilising the kernel. Third, since many security tools and rootkits historically abused system call table hooking for stealth [78], modern kernels actively discourage and defend against it.

Because of these issues, provenance systems using LKMs have increasingly shifted towards other in-kernel interception mechanisms such as LSM hooks and eBPF programs. These provide similar interception capabilities but in a way that is officially supported and more stable across kernel versions.

Linux Security Module (LSM). The LSM framework is a standardised interface for enforcing security policies at key points within the kernel. Unlike ad-hoc system call interception, LSM introduces a set of hooks throughout the kernel at critical enforcement points, such as system call handling, process creation, inter-process communication, networking, and VFS operations. These hooks allow external modules to register their own decision-making logic without the need to modify the kernel core.

The significance of LSM is in its separation of mechanism from policy. The kernel provides a consistent interface for mediation, but the choice of policy is deferred to loadable modules. This decoupling has been essential for both academic research into new security models and for adoption across heterogeneous deployment environments. Since LSMs are triggered during authorisation checks, they provide precise visibility into who is accessing what and under what context, allowing provenance graphs to be constructed with strong semantic guarantees.

Figure 2.3 illustrates the flow of a file read operation in a system with LSM hooks integrated into the kernel. When a user process such as a notepad or a browser attempts to access a file, it issues a system call that transitions control from user space to kernel space through the system call interface. The call is directed to the relevant handler, in this case *sys_read()*. Before the kernel proceeds with the read operation, the LSM framework invokes a hook for the read call. This hook acts as an interception point where a series of security policies are evaluated. Each policy check determines whether the operation is permissible according to access control rules such as role-based restrictions, multi-level security, or application-specific constraints. Provenance capture

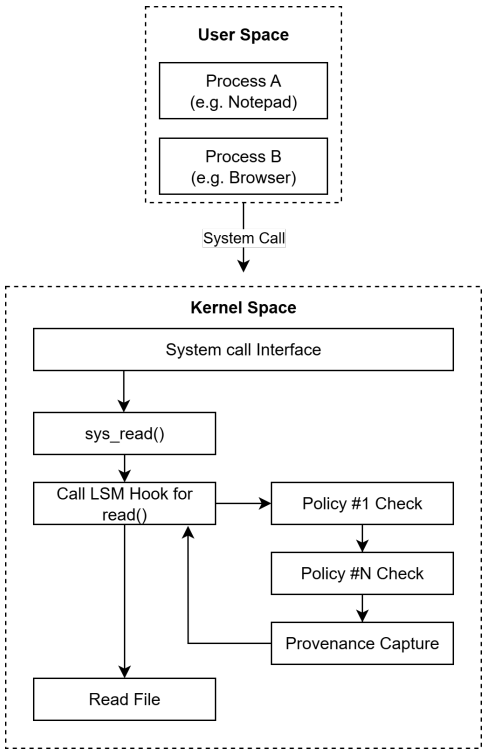


Figure 2.3: Example: LSM hooks in provenance capture during file read operation.

can be implemented as a LSM module to record contextual metadata about the access, providing a traceable history of which process accessed which resource, when, and under what authorisation conditions. Only if all checks succeed does the kernel proceed to complete the read operation and return the file contents to the requesting process. This flow demonstrates how LSM integrates fine-grained policy enforcement and provenance tracking directly into the kernel's system call path, ensuring that security controls are applied before resource access is granted.

Recent advances in the LSM stack, such as support for stacking multiple modules, make this a viable long-term path for integrating provenance capture in production systems. However, care must be taken to avoid performance bottlenecks and to ensure compatibility with other security modules like SELinux [79] and AppArmor [80].

Extended Berkeley Packet Filter (eBPF). eBPF is a kernel-level execution framework that enables programmable execution of custom code directly within the operating system kernel. Originally developed for efficient packet filtering, eBPF has since evolved into a general-purpose instrumentation framework, allowing user defined programs to be attached to diverse kernel hooks, including system call entry and exit points, tracepoints, and performance events. Unlike direct system call interception or static LSM hooks, eBPF allows a provenance system to dynamically define what should be monitored, without requiring kernel recompilation or intrusive kernel module loading.

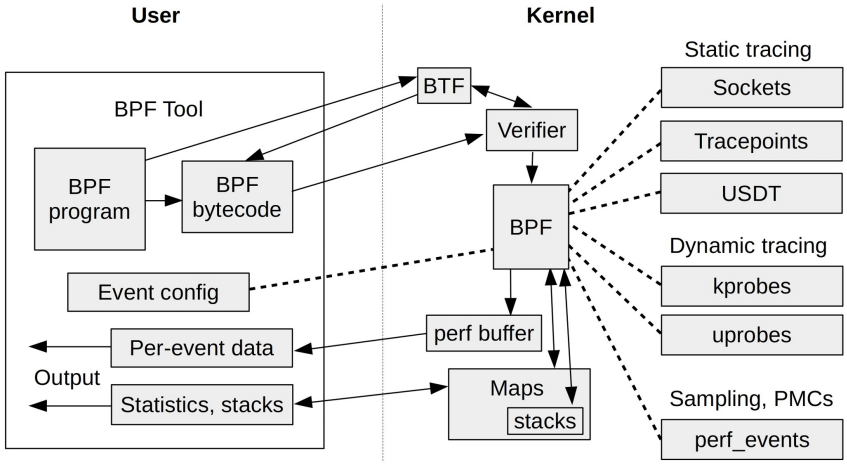


Figure 2.4: High-level architecture of eBPF based tracing and monitoring [81].

From the perspective of provenance, eBPF can be used to trace system calls, network packets, block Input/Output (I/O) events, and scheduler activity, all with fine-grained control over filtering and data extraction. Provenance collection can therefore be tailored to capture only those events that are causally relevant, reducing overhead compared with blanket interception. Furthermore, eBPF programs can maintain state maps in kernel memory, enabling correlation of events before they are exported to user space.

Figure 2.4 illustrates a high-level overview of eBPF architecture. Provenance is captured by instrumenting kernel events through a structured toolchain that bridges user space and kernel space. A BPF program is written and compiled into bytecode, which is then submitted alongside BPF Type Format

(BTF) metadata to the kernel. The verifier ensures safety before the bytecode is loaded into the BPF subsystem. Once resident, the program can attach to a wide range of kernel hooks, including static tracepoints, dynamic probes (*kprobes* and *uprobes*), sockets, User space Static Defined Tracing (USDT) probes, and performance counters. This allows it to observe and record interactions between processes, system calls, and resources at runtime. The output from these executions is stored in maps and *perf* buffers, which are returned to user space as per-event data, statistics, and stack traces.

eBPF is safe by design, unlike raw *kprobes* or system call table patching, eBPF programs are verified by the kernel before execution to guarantee termination and prevent illegal memory access. This makes them deployable in production environments where traditional kernel instrumentation would be considered too risky. However, the expressiveness of eBPF is limited by the verifier, which disallows unbounded loops and enforces strict constraints on program complexity. This means that heavy provenance reasoning cannot be done in kernel space; instead, eBPF should be seen as a highly efficient filter and event extraction mechanism, with causal reconstruction deferred to user space analysis. Additionally, because eBPF is still evolving rapidly, portability across kernel versions can be challenging [82].

Stackable File System.

Stackable File System takes advantage of the modular nature of the file system interface to introduce new layers of functionality without altering the underlying kernel or core file system implementation. A stackable file system is implemented between the application and the lower-level file system, intercepting file operations, logging them, and in some cases enriching them with contextual information before forwarding them to the next layer. This approach offers a relatively clean separation of concerns, allowing provenance capture logic to be developed and maintained independently of the complexities of a file system.

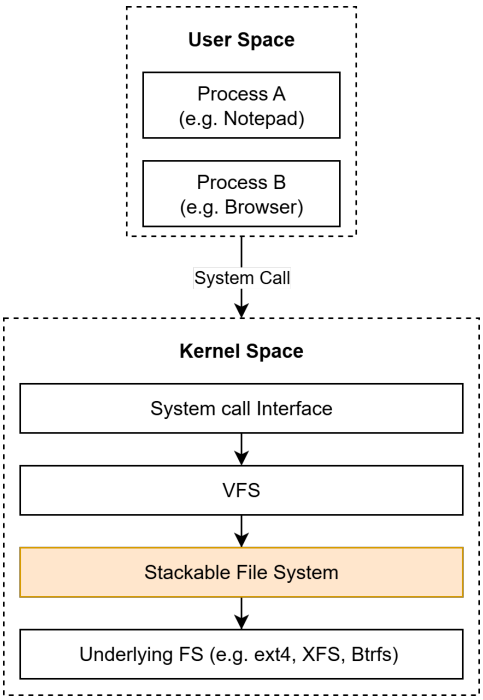


Figure 2.5: Position of a stackable file system in the kernel.

Figure 2.5 illustrates how a stackable file system fits within the operating system's I/O stack and provides a point of interception for provenance capture. User space applications issue file operations through system calls, which are routed into the kernel via the system call interface. From there, the VFS layer provides a generic abstraction that allows different file system implementations to coexist transparently.

The stackable file system sits between the VFS and the underlying file system (such as *ext4*, *XFS*, or *Btrfs*). At this point in the stack, it is able to intercept and process all file-related operations before they reach the physical storage layer. The provenance recorded by a stackable file system is naturally tied to file level activity. Every *open()*, *read()*, *write()*, and *close()* operation can be observed and associated with metadata such as process identifiers, user identifiers, and timing information. Because this layer sits close to the user–kernel boundary, it can capture a wide range of application behaviour while remaining agnostic to the details of the applications themselves. It does not require rewriting or instrumenting applications.

While this technique can reliably track provenance related to file access

and manipulation, it has limited visibility into system activity beyond the file system layer. Inter-process communication, network events, and in-memory data flows are not naturally exposed to a stackable file system, which constrains the completeness of provenance when viewed from a holistic perspective.

2.4.2 User Space Monitoring and Other Methods

Some provenance collection systems deliberately operate in user space, accepting a reduction in fidelity in exchange for ease of deployment. By intercepting system calls at the user level, such approaches can still capture a broad range of relevant events. The SPADE provenance framework [4] implements such design choice. In its initial implementation, SPADE avoided kernel modifications by using a Filesystem in Userspace (FUSE)-based file system to intercept file operations and by extracting process information from the */proc* filesystem. Provenance collection was structured around modular reporters, each responsible for tapping into a specific source of events. These reporters transmitted events to a central provenance service, which unified them into a consistent provenance representation. This architecture decoupled provenance collection from the operating system, thereby simplifying deployment while maintaining flexibility in the integration of diverse event sources.

Another noteworthy direction is capturing provenance at the virtual machine or hypervisor level, extending coverage into cloud and distributed environments. Macko et al. modified the Xen hypervisor to collect provenance from guest VMs [83], thereby reducing reliance on the guest operating system for trustworthy reporting, though this approach still faces the semantic gap between hypervisor-visible events and application-level activity. Beyond hypervisor modification, researchers have also investigated taint tracking and binary instrumentation. For example, ProTracer [84] demonstrates an alternating strategy that combines system call logging with targeted taint analysis, enabling it to record data flows that are invisible at the system call boundary while keeping overhead lower than continuous tainting. These techniques yield

fine-grained provenance but remain resource intensive compared with coarse event logging. Over time, general-purpose provenance systems have tended to converge on designs that strike a balance between fidelity and efficiency, typically by integrating with existing operating system frameworks such as the LSM and, where possible, exploiting hardware or virtualisation support. Such designs make provenance collection more transparent and reduce performance costs, though overhead and data management remain significant considerations in practice.

2.5 Provenance Storage

Once provenance information has been captured, the challenge shifts to how it is stored in a manner that ensures scalability and accessibility.

Early work on provenance storage was typically relational, leveraging well-understood database technologies to persist provenance records as tuples [37, 85]. While such approaches benefited from established query optimisations and transactional guarantees, they proved inadequate when confronted with the highly interconnected and recursive nature of provenance [37]. Provenance graphs, which naturally capture relationships between entities, activities, and agents, pushed the field towards graph databases and dedicated provenance stores [14, 63]. Systems such as RDF triple stores or graph databases like Neo4j [86] became increasingly attractive for storing provenance [87], compliant with standards such as W3C PROV. Hybrid approaches that combines relational backends for structured attributes with graph representations for lineage is also explored, attempting to balance query efficiency with representational fidelity [25].

Provenance by its very nature is verbose, even simple workloads can generate large volumes of events describing system calls, data flows, and execution contexts. In [88], Ko et al. observed over 29,000 provenance events within just 30 minutes of running Microsoft Word, despite minimal or no user ac-

tivity. Though not strictly a provenance system, GrAALF [89] operates on provenance-like data in the form of detailed system audit logs that capture process interactions and information flows. In their evaluation, the authors report that a single host generates more than 2.5 million system events in an hour, and other studies cited note that a single device can produce over 3 GB of audit logs daily.

2.6 Provenance Sharing and Interoperability

Provenance sharing mechanisms allow data to carry its history across systems. A foundational development in this area has been the creation of standard provenance models and formats. Many general-purpose systems align their internal records with widely accepted standards, ensuring interoperability. Some implementations use centralised provenance servers where multiple systems report their provenance to a single repository, while others employ federated models where peer nodes exchange provenance information. Another method for provenance sharing is embedding provenance metadata directly with the data as it is transmitted. Some systems implement this by inserting provenance into network packets, ensuring continuity of provenance across distributed systems. These network-level provenance techniques require coordination and security measures but enable provenance-aware data exchange in cloud environments. Provenance sharing also raises issues of policy and access, as not all provenance can be freely shared due to privacy or security concerns.

2.7 Provenance Reconstruction

Provenance is not only captured during data processing but can also be reconstructed retrospectively. The process of reconstructing provenance relies on the ability to trace the captured metadata back to its origins. Reconstructing provenance from logs [90], system traces, or stored metadata can be done through reverse queries or by utilising specific algorithms that piece together

the fragmented records. This can be performed either as a post-hoc analysis or, in some systems, incrementally as data evolves. Approaches range from reverse queries over system logs to algorithmic methods that piece together fragmented records. This becomes particularly necessary in heterogeneous environments, where data flows across multiple systems or undergoes transformations that are not consistently captured.

A core challenge in provenance reconstruction is establishing the completeness and correctness of the inferred lineage. In practice, captured information is often partial, inconsistent, or missing altogether. As a result, reconstruction typically involves resolving conflicts, filling gaps, and dealing with uncertainty. Graph-based algorithms, probabilistic models, or heuristics can be employed to infer plausible data flows where explicit records are absent. These methods provide estimates or hypotheses rather than guarantees, and their accuracy depends heavily on the richness and reliability of the underlying metadata.

Provenance reconstruction is valuable in several contexts. In regulatory and compliance settings such as finance, healthcare, or scientific research, it supports validation and auditing by making it possible to verify data integrity and trace origins, even when records are incomplete. In engineering contexts, it aids debugging and error analysis by highlighting how erroneous or unexpected outputs might have arisen from earlier transformations. In both cases, reconstructed provenance provides additional assurance, but only relative to the quality of the captured traces.

In some scenarios, reconstruction goes beyond explicit logs. Techniques drawing on semantic reasoning, domain knowledge, or machine learning can infer indirect or implicit relationships between data elements. Such methods are especially useful when operations are abstract or high-level, and not explicitly recorded. However, this remains an emerging area: while these approaches can improve the apparent completeness of reconstructed provenance, they introduce uncertainty and cannot ensure exact fidelity to the original processes.

2.8 Provenance Lifecycle

Provenance information is a dataset in its own right. The management of this data is non-trivial because provenance exhibits distinctive characteristics. It is graph-structured, highly interconnected, and grows in proportion to the scale and complexity of the underlying data and processes. The volume and velocity of provenance generation mean that it must be managed with the same attention as primary data, if not more, given its role in reproducibility and accountability.

One of the most immediate technical challenges is scale. Consider workflow systems such as Kepler [61] or Taverna [91], where each task execution, intermediate artefact, and dependency produces provenance events. For large-scale scientific workflows, the resulting provenance graph can contain millions of nodes and edges. Storing such graphs naïvely leads to prohibitive storage requirements, prompting the adoption of pruning and summarisation strategies. Pruning typically removes intermediate nodes that are deemed low value, such as transient files or temporary process steps, while retaining nodes representing key datasets or decisions. Summarisation, by contrast, aggregates provenance at higher levels of abstraction, collapsing sequences of fine-grained steps into a single composite node. Buneman et al. (2006) describe provenance compression techniques that identify repeated substructures in provenance graphs and encode them compactly, while Chapman and Jagadish (2009) propose indexing mechanisms that store only differences between successive provenance versions to reduce redundancy.

2.9 Isn't This... (Related Concepts)

In this section, we explore the intersections and distinctions between Provenance and other similar concepts that often overlaps in their functionalities or objectives, such as logging and version control. While these domains share a foundational interest in tracking changes, history, and activities within their

respective systems, they are distinguished by their primary focuses. Each area emphasises different aspects of tracking based on its unique requirements and objectives.

2.9.1 Logging

Logging is the process of recording events, actions, or state changes in a system. Logs are typically stored in files, databases, or monitoring systems and provide a chronological record of system behavior. Logging is widely used in software development, system administration, and cybersecurity to track system activity, diagnose issues, and monitor security incidents. A log entry generally consists of a timestamp indicating when the event occurred, a severity level categorising the importance of the event, a message describing the event, and optional metadata such as user identifiers, system component details, or error codes.

Logging and data provenance are often linked because both involve recording information about activity. However, they serve different purposes and are not interchangeable. While both logging and data provenance track events and changes over time, aiding in system behavior analysis, debugging, and transparency, they differ in scope, granularity, and use cases. Logging captures a broad range of events, including operational activities, errors, and security incidents, whereas data provenance focuses specifically on tracking the origins, transformations, and history of data. Data provenance provides fine-grained details on data lineage, including how data is modified, by whom, and for what purpose, while logs typically capture more general system events. Logs are primarily used for debugging, auditing, and monitoring, whereas data provenance ensures reproducibility, accountability, and trust in scientific workflows, machine learning models, and data-intensive applications.

2.9.2 Version Control

Version control is a system that manages changes to documents, programs, and other collections of information over time. It enables multiple users to collaborate on the same project without interfering with each other's work. Predominantly used in software development, Version Control Systems (VCS) like Git, Subversion, and Mercurial track changes to source code or project files, allowing developers to revert to previous states and explore the evolution of software over time.

Version control and data provenance share the core principles of traceability and auditability [92]. Both systems allow for the tracking of history and origin, with version control focusing on code and documents, and data provenance encompassing the broader lineage of data. In this sense, version control can be understood as a specialised form of provenance, primarily concerned with recording the evolution of versioned artefacts rather than capturing the complete lineage of data. This shared emphasis on historical tracking facilitates collaboration among multiple stakeholders and promotes accountability by maintaining a record of changes over time.

Despite their similarities, version control and data provenance cater to different needs and function within different contexts. Version control is tailored for managing software and documents, emphasising the evolution and versioning of files. Data provenance, however, focuses on the lineage of data, capturing detailed records of who has made changes, when, and under what circumstances. Furthermore, data provenance often incorporates contextual information about the reasons for data alterations, which is generally outside the scope of version control systems.

A further distinction lies in capture mechanisms. Version control relies on explicit user actions, where changes are only recorded when collaborators choose to commit. This model suits software development, where changes are deliberate and consolidated at specific points. Provenance systems, however, aim for unbroken chains of data history. They capture operations au-

tomatically and at finer granularity to preserve integrity and transparency, requirements that are critical in domains demanding strict audit trails, such as scientific workflows [26], finance [17], and healthcare [52].

The two approaches are not interchangeable, but they can be complementary. Provenance can be layered on top of version control. De Nies et al. [92] introduced Git2PROV, a system that exposes the provenance embedded in Git version control repositories in the W3C PROV standard. The tool maps Git concepts such as commits, authors, committers, and file versions to PROV notions of Activity, Agent, and Entity, linking them through relations like *wasGeneratedBy*, *used*, and *wasDerivedFrom*. Git2PROV can be run locally or accessed as a REST-like web service for public repositories, and supports multiple PROV serialisations including PROV-N, PROV-O, PROV-JSON, and PROV-XML. Their work demonstrates how version control histories can be converted into structured, interoperable provenance data, thereby enabling their integration into provenance-aware applications, particularly in Linked Data and Semantic Web contexts.

2.10 Conclusion

In closing, this chapter has set the groundwork by defining the scope of provenance and positioning the formalisation used in this thesis against the wider variety of interpretations in the field. It has examined how provenance can be captured, stored and queried, while also recognising that gaps often necessitate reconstruction through indirect means. From there, the discussion has moved to the lifecycle of provenance, emphasising that its value depends not only on how it is produced but also on how it is maintained and evolved over time. By situating provenance alongside related practices, such as versioning and logging, the chapter has clarified both the overlaps and the boundaries that distinguish it. With these conceptual and technical foundations in place, the next chapter turns to related work, examining concrete implementations

that embody the techniques introduced here and assessing how they address the challenges of provenance in practice.

Chapter 3

Related Work

In the preceding chapter, the conceptual foundations of provenance were established through definitions, models, and the technical approaches for capturing and storing provenance information. That discussion set out the abstractions and methods that make provenance both a research discipline and an applied practice.

This chapter shifts the focus from concepts and techniques to their embodiment in concrete provenance systems. Eleven systems are reviewed in chronological order, spanning a range of implementation techniques including loadable kernel modules (LKM), Linux security modules (LSM), and user space mechanisms. The specific details of these implementations are not central to this work; what is of greater value is a critical assessment of how these systems measure against the core abilities that this research seeks to enable. The emphasis therefore falls on design considerations, on whether and how provenance models are adopted or adapted, and on the practical strengths and weaknesses that emerge in their operation.

3.1 PASS

Historically, provenance had often been handled in scientific databases, source control systems, or specialist workflow managers. These approaches required manual annotation or domain-specific instrumentation, and crucially they sep-

arated data from provenance, making consistency difficult to guarantee. Provenance-Aware Storage System (PASS) [93] is motivated by this problem. It proposes that the storage system itself is the natural locus for provenance management, precisely because storage already governs metadata and is in the unique position to capture system-level events consistently.

Provenance Model

PASS does not implement W3C PROV or OPM as both of these work only emerged later in standardisation. Instead, it defines a low-level, attribute–value record model optimised for kernel level capture. Provenance is essentially represented as Directed Acyclic Graphs (DAGs) relating nodes, capturing system level context such as command line arguments, process environments, and library dependencies. The authors note that their representation can be mapped into relational or XML formats, making it adaptable to higher-level standards.

Provenance Capture

PASS's capture mechanism is embedded in the Linux kernel. Provenance is collected by a collector that intercepts system calls such as *execve*, *open*, *read*, and *write*, and translates them into provenance records. These records are associated with kernel structures (process task structures, *inodes*, etc.), forming an in-memory ancestry graph that links files to processes, environment variables, libraries, and even kernel modules. When persisted, each file or process is assigned a unique provenance identifier (*pnode*), and non-persistent objects such as pipes have their provenance copied into dependent objects. Special mechanisms are implemented to handle versioning and cycles in the graph; for example, cycles caused by processes reading and writing the same file are broken using node merging, where interdependent processes are collapsed into a single provenance entity.

Provenance Storage

PASS employs a stackable file system called *PASTA*, layered on the *ext2* filesystem and coupled with an in-kernel Berkeley DB (KBDB). Provenance records are stored in five databases. One for mappings of *inodes* to *pnodes*, one for provenance entries proper, and several optimised for command-line argument indexing. This choice allowed indexed lookups and reasonable query times. Retention was handled append-only in the prototype, though the authors identified pruning and garbage collection of provenance metadata as a major open challenge, since uncontrolled growth could quickly become wasteful.

Provenance Query

PASS exposes provenance databases through standard Berkeley DB interfaces. Tools were built both for interactive browsing, that is the Provenance Explorer, a Java interface and also tool for command line scripting. Queries supported include generating the sequence of commands that produced a file (akin to auto makefile generation), retrieving complete provenance records, looking up files by command line arguments, and listing annotations. These tools enabled both practical debugging, such as discovering hidden system dependencies in build environments and scientific reproducibility.

Evaluation

In their evaluation of the original PASS prototype, Muniswamy-Reddy et al. reported relatively low overhead in a large file microbenchmark, with sequential reads incurring a 2.01% slowdown and sequential writes a 15.47% slowdown [93]. However, their evaluation also showed substantially higher overheads for workloads involving many small files. In a microbenchmark involving 2,500 files of 4 KB each, they measured time overheads of 232.41% for file creation and 172.09% for writing, while the corresponding disk space overheads were 16.89% and 29.33%, respectively. The authors attributed the elevated

time overheads partly to duplicate elimination and the generation of additional provenance records. They also observed an especially large space overhead for deletion, measuring a 14,466% increase, with PASS occupying 3,486 KB compared with just 24 KB for *ext2* after deletion. This result demonstrated that provenance related storage could persist even after the underlying files had been removed. Despite these measured overheads, Muniswamy-Reddy et al. considered the performance impact acceptable, arguing that the absolute execution times remained small and that the small file benchmark represented a particularly challenging workload for PASS [93].

Other Considerations

PASS was not primarily designed with adversarial security in mind; it focused on correct and complete provenance collection rather than making the logs tamper-proof. Provenance data in PASS is stored as part of the filesystem, which means integrity relies on the system's security policies. There were no built-in cryptographic integrity guarantees for the logs in the initial design. In fact, the PASS authors acknowledged that adding tamper-proofing (to prevent malicious modifications of provenance) would complement their work, but was outside PASS's initial scope. Thus, if an attacker gained sufficient privileges (e.g., root access), they could potentially alter or delete provenance records in PASS. On the other hand, because PASS stores provenance tightly coupled with data, it benefits from any security that protects the data itself. For example, if the file system is secure and access-controlled, the provenance is similarly protected by those same controls.

PASS was among the first to demonstrate that provenance can be captured transparently at the system level, paving the way for later systems that integrated provenance into cloud storage, scientific workflow tools, and eventually standardised models such as W3C PROV. Later work introduced PA-S3FS [94], a Filesystem in Userspace (FUSE) module to allow storing provenance in cloud storage, indicating some flexibility in storage back-end.

3.2 PASSv2

PASSv2 [2] builds on the earlier PASS to integrate provenance across multiple layers of abstraction, including operating systems, workflow engines, browsers, and scripting environments. Rather than treating provenance as an isolated feature of one layer, the authors develop a general framework through which provenance flows with data, creating a unified infrastructure that enables cross-layer reasoning.

Muniswamy-reddy et al. showed that layering enables queries that no single layer can support. For example, a workflow engine processes inputs stored on a remote Network File System (NFS) server and produces outputs on a different one. If only workflow provenance is captured, the system misses changes in inputs outside the workflow's scope. If only file system provenance is captured, the logical grouping of workflow stages is lost. When layered together, the system can explain precisely why two workflow runs differ, linking changes in NFS input files to altered final outputs. Similarly, layered provenance in a browser allows tracing malware back both to its web origin and to its subsequent spread across the file system.

Provenance Model

Similar to its original work, PASSv2 predates the W3C PROV and OPM standards. It does not adopt an external model but defines its own lightweight record format centred on objects, attributes, and relationships. Objects are assigned permanent identifiers (*pnode* numbers), and relationships are expressed through Disclosed Provenance API (DPAPI), a universal interface that defines operations like *pass_read*, *pass_write*, *pass_mkobj*, and *pass_sync*. While not formally compatible with OPM or PROV, the system's graph based model anticipates many of the same principles, and later inspired query languages such as Path Query Language (PQL).

Provenance Capture

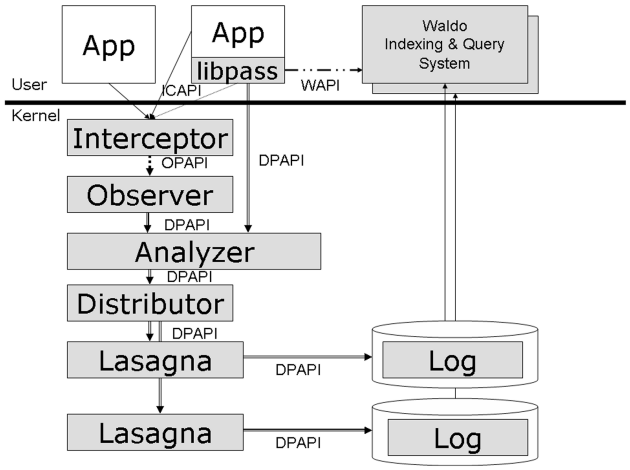


Figure 3.1: PASSv2 architecture overview [2].

Provenance in PASSv2 is captured via a modular architecture as illustrated in Figure 3.1. At the system level, an interceptor hooks into system calls such as *execve*, *fork*, *read*, *write*, *mmap*, and *pipe*. These events are passed to an observer, which translates them into provenance records, linking processes and files with precise versioned identifiers (*pnode* numbers). The observer forwards records to an analyser, which deduplicates events and applies a cycle avoidance algorithm to prevent inconsistent dependency graphs. Application level systems can augment this by calling the DPAPI. Through DPAPI, applications disclose semantic objects (e.g. workflow operators, browser sessions, Python function calls) which PASSv2 then integrates with system level provenance. Thus, provenance is captured both passively (system call interception) and actively (application disclosure).

Provenance Storage

Provenance is stored through a hybrid mechanism. The Lasagna file system, derived from *eCryptfs* [95], logs both data and provenance in a consistent fashion. Provenance records are first written to log files alongside transactional metadata and Message-Digest Algorithm 5 (MD5) checksums, ensur-

ing consistency under crash recovery. A user level daemon, *Waldo*, monitors logs, extracts provenance, and indexes it into a database for efficient querying. Provenance retention is managed by rolling logs. Once logs exceed a configured size or idle period, they are closed and processed. This avoids scalability bottlenecks that was observed in the original PASS implementation's direct database writes, while still ensuring eventual structured storage.

Provenance Query

The system introduces PQL, derived from Lorel [96] but adapted for graph traversal. PQL treats provenance as a graph of objects and relationships, enabling users to query by following paths. Queries can specify regular expressions over edges, support subqueries and aggregation, and treat paths as first class objects. For example, a query can return all ancestors of a given file by traversing provenance edges, or restrict to certain operator types in a workflow. This represents a significant advance over relational or XML-based provenance storage, which struggled with expressing graph queries naturally.

Evaluation

The authors evaluated runtime and storage overheads using five benchmarks: Linux kernel compilation, the Postmark email server simulation [97], Mercurial version control activity, a biological BLAST workload [98], and a Kepler workflow [61]. Runtime overheads varied by workload. BLAST and Kepler incurred minimal slowdowns of approximately 1–2%, whereas Linux kernel compilation and Postmark experienced moderate overheads. Metadata intensive Mercurial patching incurred the highest runtime overhead at 23.1%, primarily due to interference between provenance writes and metadata operations. Storage overheads were comparatively modest, with provenance records accounting for less than 7% of baseline storage across all workloads. Including database indexes, total storage overheads reached 18.4% for Linux compilation and 14.2% for Kepler.

Other Considerations

PASSv2 was designed for distributed settings. With Provenance-Aware NFS (PA-NFS), provenance records are transmitted alongside data over the network using extended NFS operations (*OP_PASSWRITE*, *OP_PASSREAD*, *OP_PASSMKOBJ*, etc.). This ensures that provenance flows across client–server boundaries. Browser and workflow integration further illustrate dissemination to end-users. A file downloaded from the web retains its URL provenance even if renamed or copied locally. Thus provenance is preserved across application, host, and storage boundaries. However, the work focused on integration within cooperating systems rather than standardised external sharing; interoperability with independent provenance systems remained limited.

3.3 Story Book

Story Book [3] represents one of the early attempts to simplify provenance capture by moving the bulk of the functionality into user space. Story Book is a provenance framework that treats events as a generic log, decoupling application specific provenance extensions from the common mechanisms of dependency tracking, query, and storage.

Unlike earlier systems such as PASS, which were tightly coupled to the kernel and application data representations, Story Book makes a deliberate move to user space. This design is to allow the system to rely on off the shelf components such as FUSE for file system interception, MySQL for database logging, and Stasis [99] or Berkeley DB for storage. The motivation behind this work is the pursuit of flexibility and extensibility. Provenance is made a pluggable concept, so that it can be applied not only to file systems but also to databases, text formats, and potentially services like email or web servers. In historical context, this design reflects a transition away from monolithic provenance architectures that were tightly bound to kernel mechanisms and required provenance specific code paths.

An example of what the system enables is file type aware provenance capture. The authors implemented inspectors for plain text and Microsoft Word documents. For *.txt* files, Story Book records patches that reflect the changes made between open and close operations, whereas for *.docx* files, it captures changes to document metadata such as the author field. The experiments show that, although this introduces measurable overheads, the approach works in practice and provides provenance at a granularity previously difficult to obtain outside of application specific instrumentation.

Provenance Model

Story Book does not adopt established standards such as W3C PROV or OPM. Instead, it uses a custom schema tailored to its storage strategy. Provenance records are reduced to minimal, efficient tables and logs, with optional extended records storing richer metadata. This represents both a strength and a limitation. The strength is efficiency and flexibility, but the limitation is that without semantic annotations, Story Book cannot natively express relations such as "derived-from" or "version-of" without custom extensions. This omission reflects its focus on performance and extensibility over interoperability.

Provenance Capture

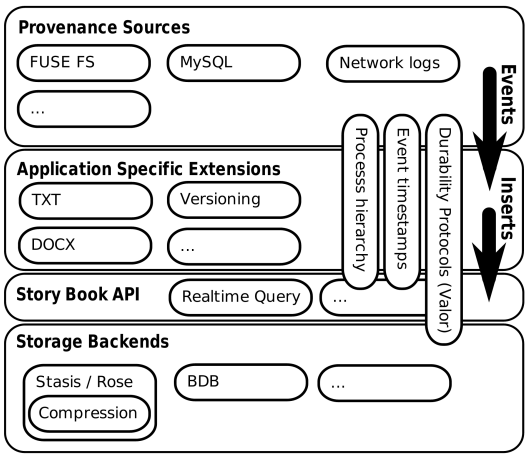


Figure 3.2: StoryBook architecture overview [3].

The capture mechanism in Story Book is layered as in Figure 3.2. Provenance sources (e.g., FUSE for file systems and MySQL logs for databases) intercept events. These are passed to application specific extensions, also called provenance inspectors, which interpret and augment them. Three types of records are generated: (i) basic records for general operations such as read, write, open, and close, (ii) process records that capture parent–child process relationships and (iii) extended records for application specific annotations. The implementation makes use of write-ahead logging and integrates with Valor [100] for enforcing write ordering relative to the kernel page cache. The storage backend, known as *Fable*, employs Stasis [99] for transaction support and Rose indexes [101], which are compressed log structured merge (LSM) trees, to ensure high throughput insertions.

Provenance Storage

Story Book separates provenance metadata from application data, a design decision explicitly made for performance. Basic and process records are stored in hash tables and Rose indexes, while extended records are appended to logs. The system cannot automatically truncate these logs, as they contain unique application specific data. This highlights a conflict between performance and long-term manageability. Retention periods and storage pressure become open issues for deployment. The choice of compressed LSM trees reflects sensitivity to the cost of random versus sequential I/O, with the authors noting that sequential access is becoming exponentially cheaper relative to random access.

Provenance Query

Querying provenance involves a breadth-first traversal over basic and process records, beginning with an *inode* lookup. Transitive closure is performed over processes and files, in reverse chronological order based on log sequence numbers. This design allows queries to reconstruct provenance graphs without maintaining large in-memory structures. However, the lack of semantic aware-

ness means that queries over annotations incur additional cost, as extended records are stored separately.

Performance

The authors benchmarked Story Book against *ext3* filesystem and PASSv2 under file system workloads, as well as under TPC-C database benchmarks [102] with MySQL. Results show that Story Book is slower than *ext3* in worst-case workloads, primarily due to FUSE overhead. When compared with PASSv2, Story Book demonstrated superior throughput, as its bottleneck was in generic instrumentation rather than provenance specific graph manipulation. For MySQL, provenance tracking incurred a small increase in response times and throughput loss. Query performance was mixed. Berkeley DB offered faster queries, while Fable excelled in write throughput. Overall, the evaluation highlights the practicality of Story Book's design but also its limitations in latency-sensitive environments.

Other Considerations

Story Book was not designed with cross-system provenance dissemination in mind. It can, in principle, be extended to distributed settings using established mechanisms such as logical clocks or Paxos [103], and the authors explicitly note this possibility. Yet in its published form, Story Book maintains provenance locally, without providing mechanisms for cross-system exchange. This represents a limitation when compared to more recent provenance models that emphasise interoperability across services and infrastructures.

Story Book must be understood as a counterpoint to in-kernel, monolithic approaches such as PASS. By externalising provenance functionality, it simplified the addition of provenance to new applications and demonstrated the viability of provenance as a modular, extensible layer. Its limitations foreshadow issues that later work in provenance standards and distributed provenance infrastructures sought to address.

3.4 Flogger

File-centric Logger (Flogger) [88] was one of the earliest attempts to design a file-centric logging mechanism specifically aimed at Cloud environments. It is a kernel-level logger deployed across both virtual machines (VMs) and physical machines (PMs). Its purpose is to capture fine-grained records of file accesses and file transfers, thereby providing a consolidated provenance view of the file life cycle across distributed cloud infrastructures.

What sets Flogger apart from other contemporary systems is its explicit departure from system-centric monitoring towards data-centric transparency. Traditional mechanisms such as TripWire [104] or system activity monitors either focus on integrity checking or on performance statistics, but they do not expose the history of file events in a way that satisfies forensic, regulatory or accountability needs. The motivation behind Flogger is rooted in the gap where cloud providers have concentrated on preventive controls, Ko et al. recognised the absence of detective controls that would allow auditors, users, and regulators to interrogate *who*, *what*, *where*, *when*, *how*" in relation to files in the cloud.

To illustrate what the system enables, the authors present a scenario where a user creates and modifies a confidential file, which is later accessed by another user without permission. Flogger captures and consolidates these events, exposing the unauthorised access in the joined VM/PM logs. A second scenario shows how Flogger traces file transfers across VMs through socket activity, proving the feasibility of tracking movement as well as access.

Provenance Model

Flogger does not directly adopt W3C PROV or OPM. Rather, it generates raw, attributed logs from which provenance graphs could later be derived. The authors situate Flogger within the "System Layer" of the TrustCloud framework [105], viewing the file-centric logs as the building blocks upon which

higher level provenance models and regulatory compliance processes might later be constructed.

Provenance Capture

Flogger was implemented for both Linux and Windows. On Linux VMs, a LKM was used to intercept system calls related to file and network operations; while on Windows, a device driver performs equivalent interception. These events are recorded as *flogs*. Each log entry contains attributes such as full path, user and process identifiers, MAC and IP addresses, precise timestamps, and the nature of the operation (create, read, write, delete, socket send/receive). The VM flogs are sent securely to the host PM over a hypervisor supported serial like communication channel, avoiding network transfer and reducing the attack surface. On the PM side, additional contextual data is appended, such as host identifiers, before consolidation into a database.

Provenance Storage

Storage is managed through database loaders that transfer the consolidated logs into relational and columnar stores. PostgreSQL [106] was used for baseline testing, while MonetDB [107] was explored to meet anticipated analytical needs. Considerations of retention and scale are discussed but not resolved. Ko et al. notes that file-centric logs grow extremely quickly, potentially outpacing the size of the files themselves. Early experiments showed tens of thousands of events generated by a single document under automatic backup. The paper notes this as a limitation and sketches possible mitigations through tiered storage, archival, de-duplication, and summarisation.

Provenance Query

Querying is achieved through conventional database tools, spreadsheets, or front-end applications. At this stage the work is less about providing novel

query languages and more about demonstrating that the joined VM/PM logs can be analysed with off-the-shelf software. The provenance is therefore explorable but not yet expressed in a formalised query interface.

Evaluation

Ko et al. evaluated Flogger by deploying it across several Linux distributions (CentOS 5.3, Fedora 15 and Ubuntu 11.04) and Windows systems (Windows XP Professional SP3 and Windows Server 2008 R2). Through these experiments, the authors demonstrated the feasibility of capturing file related activities across different operating systems, consolidating logs in database systems and retrospectively examining records of unauthorised file access. However, their evaluation also revealed significant limitations, particularly the rapid growth of log data and the challenges associated with scalable, long-term storage. The authors have also further identified the need for tamper-proof, immutable logs and secure transmission between physical machines and database storage.

Other Considerations

Provenance captured is preserved across systems by virtue of consolidation. The VM originating records and the PM data are joined into a single auditable view. However, the logs are not yet shared across administrative domains or integrated into cross-provider workflows. The visibility is primarily for providers, auditors, and regulators within the same cloud domain, but the authors speculate on future services that would expose tailored views to end-users and regulators with appropriate privileges.

3.5 SPADE

Support for Provenance Auditing in Distributed Environments (SPADE) [4], is an infrastructure for the collection, management, and querying of data prove-

nance in distributed computing contexts. It was motivated by the need to provide security and trustworthiness guarantees in large scale Grid environments, where heterogeneous users and organisations operate collaboratively. Unlike earlier provenance systems that were either too tightly coupled to applications or reliant on centralised logging, SPADE was designed as a distributed service with a dedicated provenance kernel that decouples collection, storage, and querying.

SPADEv1 was never released as a single, self-contained systems paper in its own right. Instead, its design and deployment details were scattered across a series of earlier publications that dealt with specific challenges such as lineage authentication, distributed querying, and provenance certification. The term SPADEv1 was only retrospectively applied when Gehani and Tariq introduced SPADEv2 in [4]. SPADEv1 is primarily discussed as a precursor whose limitations such as fine-grained certification, high overheads for adding new attributes, and rigidity in storage models, motivated the complete redesign. As a result, much of what is known about SPADEv1 survives through the SPADEv2 narrative rather than from a dedicated systems publication.

What distinguishes SPADE from earlier approaches is its focus on distributed provenance management across administrative domains while remaining agnostic to the underlying applications. Previous systems such as PASS were either centralised or application specific, whereas SPADE is situated as middleware, capable of collecting provenance unobtrusively from the operating system, applications, and network layers. Its redesign from SPADEv1 to SPADEv2 reflects this shift. The first version focused on selective replication and certification but imposed heavy overheads, while the second generation introduces a modular kernel that mediates between multiple reporters, storage back ends, and query clients. This modularity is the key motivation, intended to allow extensibility, support diverse provenance sources, and reduce the barriers to adoption in real distributed environments.

The practical capability of SPADE is demonstrated in distributed work-

flows such as those in the NIGHTINGALE [108] project, where it was used to identify bottlenecks and track dependencies when releasing new versions of large-scale toolchains. SPADE was deployed to capture provenance metadata across multiple institutions and platforms. For instance, the ability to locate bottlenecks in distributed workflows illustrates how provenance, when collected in detail, can optimise coordination in scientific collaborations.

Provenance Model

SPADE adopts the OPM model. Vertices in its graph-based data model represent Agents, Processes, and Artifacts, while directed edges encode relationships such as *WasGeneratedBy*, *WasControlledBy*, and *Used*. Each node and edge can carry arbitrary annotations, thus permitting domain specific semantics to be layered over the OPM base. This compliance situates SPADE in the broader ecosystem of interoperable provenance systems and highlights a historical shift from ad hoc representations towards standards based descriptions.

Provenance Capture

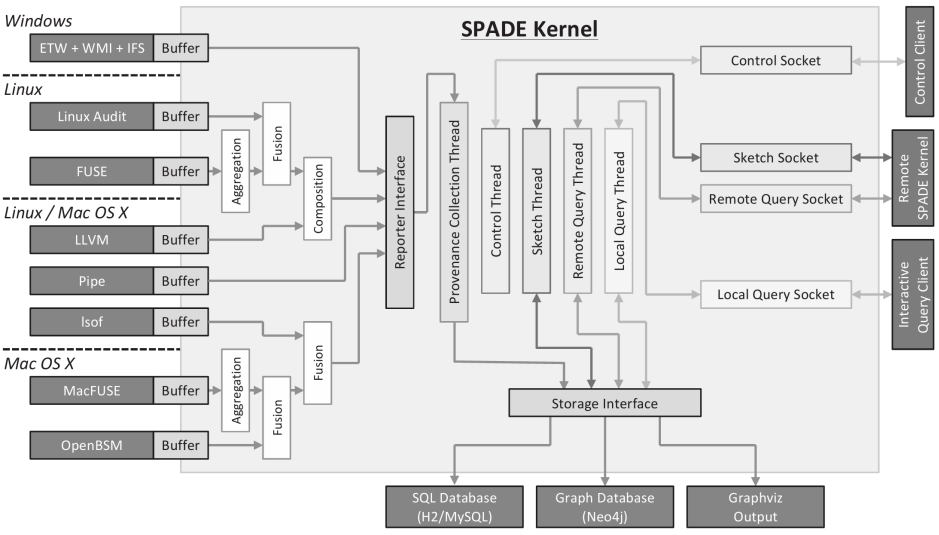


Figure 3.3: SPADE architecture overview [4].

As illustrated in Figure 3.3, SPADE employs *reporters* that abstract the

translation of system or application activity into provenance elements. At the operating system level, reporters exploit audit frameworks such as Linux Audit, OpenBSM on macOS, and Event Tracing for Windows (ETW) or Windows Management Instrumentation (WMI) on Windows, to collect process-level and file-level events. This ensures coverage across system calls like *exec()*, *open()*, *read()*, and *write()*. For finer control, SPADE provides FUSE based reporters on Linux and macOS to interpose on filesystem activity in specific directories, thus reducing overhead and providing granular context. On Android, SPADE extends the Linux kernel with audit patches to capture process and Binder IPC events. At the application level, provenance can be recorded either through a domain specific language with an OPM inspired grammar communicated over named pipes, or through compiler assisted LLVM [109] instrumentation that traces intraprocess function calls. These mechanisms allow both high level and fine grained provenance capture, though with trade-offs. For example, system-wide audit trails can produce excessive extraneous data, while compiler-based tracing requires careful filtering to avoid overwhelming volumes.

Provenance Storage

SPADEv1 relied on relational databases, but this proved inadequate for scaling provenance graphs due to expensive join operations. SPADEv2 introduces a storage abstraction layer that supports multiple back ends. Neo4j [86] is the default choice, enabling graph-native queries augmented with Apache Lucene [110] indexing to support boolean, fuzzy, proximity, and range queries. SQL backends remain supported through Java Database Connectivity (JDBC) adapters, though they are limited to vertex–edge storage without path queries. Considerations of storage overhead and schema evolution are addressed through these flexible options, though the trade-off is evident in balancing query richness against collection throughput.

Provenance Query

Querying is provided locally and across distributed systems. Locally, users can issue queries using operators such as wildcard, fuzzy, or boolean, taking advantage of the indexing back ends. Across systems, SPADE supports ancestor and descendant queries that transparently propagate until termination, and accelerates distributed path queries with provenance sketches, an optimisation to reduce the number of remote lookups. This distributed querying capability sets SPADE apart from earlier systems, which often assumed centralised provenance repositories. For visualisation and interoperability, provenance can also be output in Graphviz [111] DOT format.

Evaluation

Gehani & Tariq tested overheads on three platforms: Linux, macOS, and Windows, using representative workloads, such as building the Apache web server and running BLAST genome sequence alignments [98]. The results showed platform dependent costs, overhead on Windows during Apache builds reached over 50%, while Linux and macOS reporters added less than 10%. When running services like Apache under provenance collection, throughput degradation was below 15%. These results highlight the variability of system-level auditing frameworks and demonstrate SPADE's practical viability for scientific computing, although performance costs on Windows sees a limitation.

Other Considerations

Preservation across systems is enabled through the explicit modelling of network connections as provenance artifacts. By encoding endpoints deterministically, subgraphs collected independently on different hosts can be recombined into coherent distributed provenance chains. This decentralised approach facilitates sharing and dissemination across administrative boundaries, although the granularity of certification and the overhead of incorporating new at-

tributes proved limiting in SPADEv1. SPADEv2 alleviates this by supporting modular filters for aggregation, fusion, and cross-layer composition, ensuring greater interoperability while retaining scalability.

SPADE emerges as a response to the inadequacies of both lightweight lineage tracking and heavyweight, centralised provenance services. By adopting OPM, supporting distributed queries, and modularising collection and storage, it positioned itself as an extensible platform for scientific collaborations. Yet its limitations such as lossy collection under high load, incomplete capture of inter-process communication, and platform-specific deficiencies (such as missing IP addresses on macOS) are reminders that provenance capture in practice remains a compromise between completeness, efficiency, and interoperability.

3.6 Hi-Fi

High-Fidelity Whole-System Provenance (Hi-Fi) [33] is a kernel level provenance collection framework for Linux designed to capture whole-system provenance with high fidelity. Its objective is to provide a complete and accurate record of data origin and transformations in a system, allowing for forensic-level analysis of security events, from kernel boot through shutdown.

Earlier provenance systems had important shortcomings. Provenance collection was usually limited to file system activity or user space system calls, leaving kernel initiated actions unrecorded and creating gaps in the provenance chain. Hi-Fi distinguishes itself by rooting its design in the reference monitor concept, which insists on complete mediation, tamper-proof, and verifiability. By using the LSM framework instead of system call interception, Hi-Fi ensures kernel level fidelity that earlier systems such as PASS, or Story Book could not achieve. Its motivation is thus to eliminate fidelity gaps and provide a forensically trustworthy record that can withstand user space compromise.

Provenance Model

Hi-Fi does not enforce a fixed provenance model. Instead, it produces kernel level logs that can be consumed and transformed by handlers. The system supports flexibility whereby administrators can export raw logs or format them according to existing standards such as OPM. In their experiments, they explicitly mention using OPM to encode provenance when desired. Hence, Hi-Fi is not locked into W3C PROV or OPM but is designed to be compatible.

Provenance Capture

Hi-Fi's capture mechanism is centred on LSM hooks placed at points of mediation over kernel objects. These hooks intercept reads and writes to files, pipes, and sockets, program executions with full arguments and environments, inter-process communications, credential changes, and user transitions. Each event generates structured provenance entries describing the subject, object, action, and context. For network provenance, Hi-Fi modifies socket behaviour by assigning identifiers to queues and packets, places identifiers in IP options, and introduces new hooks for datagram level mediation. This design ensures that even ephemeral actions, such as a process forking or a socket transmitting a packet are captured as provenance events.

Provenance Storage

Captured provenance is written into a relay buffer, a kernel ring buffer optimised for efficient transfer of large amounts of data to user space. This avoids the limitations of *printk* buffers or on-disk writes used in earlier systems. The provenance handler, decoupled from the collector, then moves the logs to disk, remote servers, or alternative storage systems, depending on administrator needs. This separation is crucial because it allows integration with WORM (write-once read-many) storage or distributed collection for tamper resistance. The buffer is sized at boot to avoid overwriting, and temporary in-

memory buffers are used to preserve fidelity during early boot before the relay is initialised. The authors explicitly note retention considerations, pointing to append-only or WORM systems as natural complements.

Provenance Query

Querying is not a core function embedded in Hi-Fi itself. Rather, it is delegated to provenance handlers. These handlers can apply post-processing, graph reconstruction, or transformation into queryable representations such as databases or OPM-encoded XML. The modular design avoids constraining users to one approach. However, this means querying efficiency depends on downstream tools rather than the collector, representing both strength in flexibility and weakness in immediate usability.

Evaluation

The authors evaluated Hi-Fi in terms of both functionality and performance overhead. For the functional evaluation, they demonstrated that Hi-Fi could capture various stages of simulated malware activity, including persistence, remote control, data exfiltration, and propagation. The collected provenance logs recorded trojan binaries, cron modifications, the creation of privileged accounts, remote shells, and worm propagation, demonstrating Hi-Fi's ability to provide detailed forensic evidence. In terms of performance, the authors reported overheads below 1% for most system calls, with read operations experiencing the highest overhead of 6.2%. They also observed an average overhead of less than 3% for an I/O-bound workload. A Linux kernel build benchmark showed a slowdown of only 2.8%, while the disk intensive PostMark [97] benchmark showed no measurable performance degradation. These results suggest that Hi-Fi can collect high-fidelity provenance with limited performance overhead under the evaluated workloads, although managing the volume of generated provenance data remains a challenge for practical deployment.

Other Considerations

One of Hi-Fi's most significant contributions is socket provenance, which enables provenance to span multiple hosts. By embedding identifiers in IP options, Hi-Fi supports reconstruction of end-to-end flows across provenanced systems. When a packet is sent from host A to host B, both logs include compatible identifiers, allowing forensic investigators to reconstruct inter-host causality. This makes Hi-Fi one of the earliest systems to explicitly preserve provenance across system boundaries.

While earlier systems like PASS advanced the practice of logging system calls, their inability to capture kernel activity left forensic gaps. Hi-Fi bridged this gap using LSM, laying groundwork for later provenance systems that further integrated with security modules and network monitoring. However, limitations remain. The burden of downstream storage and querying was intentionally offloaded, which made Hi-Fi a powerful collector but less a full fledged provenance management system. Its reliance on kernel modifications, although necessary for fidelity, limited its ease of deployment compared with user space tools such as eBPF that emerged later. Nonetheless, its demonstration of cross host provenance and malware detection was novel.

3.7 S2Logger

S2Logger [5] is a direct successor and extension of Flogger, designed specifically for cloud environments. Unlike traditional system-centric monitoring, S2Logger adopts a data-centric perspective, tracking data events at both file and block level across virtual and physical machines. By linking atomic events, it produces an end-to-end provenance chain that allows administrators to visualise and analyse data flows throughout the entire lifecycle. S2Logger's objective is to detect data leakage, malicious insider actions, and policy violations in distributed virtualised infrastructures.

Provenance Model

S2Logger does not adopt existing standards such as W3C PROV or OPM. Instead, it defines its own provenance representation internally, centred around graph structures of processes, files, sockets, and their interconnections. Each node carries attributes such as host identifier, process ID, or *inode*, and edges describe causal or communicative links. While this custom model increases system efficiency, it also limits direct interoperability with other frameworks.

Provenance Capture

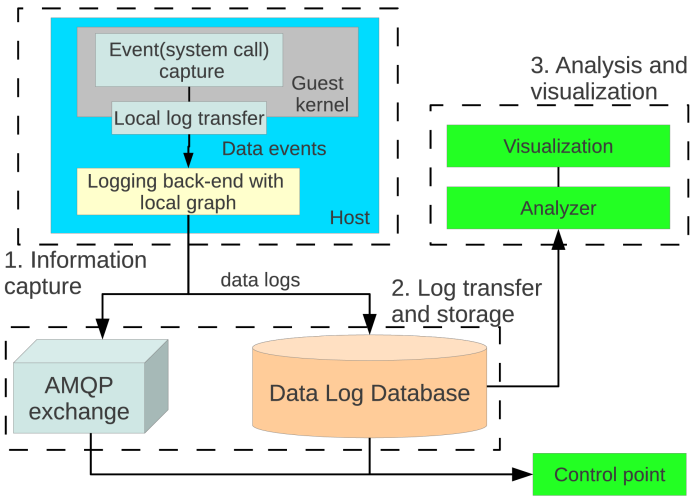


Figure 3.4: S2Logger architecture overview [5].

S2Logger captures data provenance using two complementary approaches, as illustrated in Figure 3.4. First, a LKM hooks into system call tables to intercept operations such as file opens or process creation. This technique provides broad compatibility across Linux versions but introduces complexity and risk due to the need to manipulate protected kernel structures. Second, S2Logger can alternatively leverage LSM. This cleaner approach hooks into selected kernel functions and captures parameters such as process IDs, file descriptors, socket endpoints, and user credentials. Both techniques generate structured log entries, including timestamps, system call names, and contextual metadata, ensuring fine-grained provenance capture. These logs are relayed to the

host via a shared memory channel with I/O port signalling, minimising virtualisation overhead.

Provenance Storage

The system offers both synchronous and asynchronous alternatives. In the synchronous mode, MySQL provides Atomicity, Consistency, Isolation, and Durability (ACID) guarantees, ensuring no event loss at the expense of verbosity and bandwidth. The asynchronous option uses the Advanced Message Queuing Protocol (AMQP), specifically RabbitMQ. This supports reliable message delivery, durable queues, and topic-based routing, making it suitable for real-time enforcement and notifications. Evaluations demonstrate that AMQP consumes significantly less bandwidth and CPU overhead compared to SQL based storage, largely because its serialised object messages are more compact than textual SQL commands.

Provenance Query

Provenance is queried and analysed via graph based methods. Logs are preprocessed into a graph database where nodes represent entities such as processes or files, and edges capture interactions. To aid interpretation, pruning and simplification techniques are applied, cycles are removed, irrelevant subtrees collapsed, and data leaks highlighted visually. Administrators can explore provenance through force directed graphs or tree views, making it feasible to detect data leaks, policy violations, or insider threats in large distributed environments.

Evaluation

Suen et al. evaluated S2Logger in an OpenStack-based environment comprising 100 virtual machines distributed across 10 physical hosts [112]. Performance benchmarks revealed substantial overhead from kernel level event cap-

ture, primarily attributable to string processing during log generation, which the authors identified as an area for further optimisation. In contrast, the local communication channel between guest and host introduced relatively little additional overhead. Network bandwidth consumption remained manageable under the evaluated workloads, with AMQP logging generating substantially less network traffic and lower peak CPU utilisation than SQL based logging, although the volume of transferred data still exceeded the amount ultimately stored. The evaluation also highlighted differences between the ACID-backed storage guarantees of SQL and the communication efficiency and event-notification capabilities of AMQP.

Other Considerations

S2Logger enables provenance preservation across systems. The asynchronous AMQP backend allows security tags and subgraphs to be synchronised between hosts. For example, when a sensitive file leaves one machine, its provenance state is transmitted to other hosts or control points, ensuring that security policies can be enforced consistently at network boundaries. This makes provenance dissemination dynamic rather than retrospective, and positions S2Logger as a tool for both forensics and real-time enforcement.

S2Logger addressed shortcomings of previous provenance systems by ensuring visibility across the guest–host boundary and maintaining continuity of provenance at the data, rather than system level. The absence of standards integration is a notable limitation, as is the dependence on kernel modifications, which complicates deployment in production. Nonetheless, the work marks a significant shift towards operationalising provenance for cloud accountability, blending forensic reconstruction with real-time enforcement in ways that earlier research had only theorised.

3.8 Progger

Similar to Flogger, Progger (Provenance Logger) [6] is a kernel space logger, designed to record data-centric events at the lowest possible level of the operating system. Unlike system-centric loggers that simply track usage patterns or performance metrics, Progger was created to capture the provenance of individual data items, answering the perennial cloud accountability questions such as "what has happened to my data in your cloud?"

The primary motivation behind Progger is to address the observed limitations of earlier provenance systems such as Flogger and PASS. These earlier tools were constrained by their inability to ensure tamper-evidence, causal timestamp integrity across distributed machines, or efficient space management. Progger was explicitly designed to address these weaknesses, situating itself not just as another logger, but as a step toward a high assurance provenance infrastructure that enhances accountability and trust between cloud providers and their users.

Will et al. demonstrate that Progger was used to log the full lifecycle of a file created and edited by a user with the vim editor, revealing how each save operation involves backup creation and rewriting of the file. In another example, it captures the transfer of a file into the cloud via Secure Copy Protocol (*SCP*), logging the precise system calls and socket activity involved.

Provenance Model

Progger does not adopt provenance models such as W3C PROV or OPM. Rather, it introduces its own lightweight, system call oriented log schema. Each entry encodes user identifiers, process identifiers, session identifiers, file descriptors, paths, and data in hexadecimal representation. This bespoke approach privileges efficiency and tamper-evident chaining over semantic expressiveness. While it enhances direct auditability of raw system activities, it also limits interoperability with broader provenance frameworks.

Provenance Capture

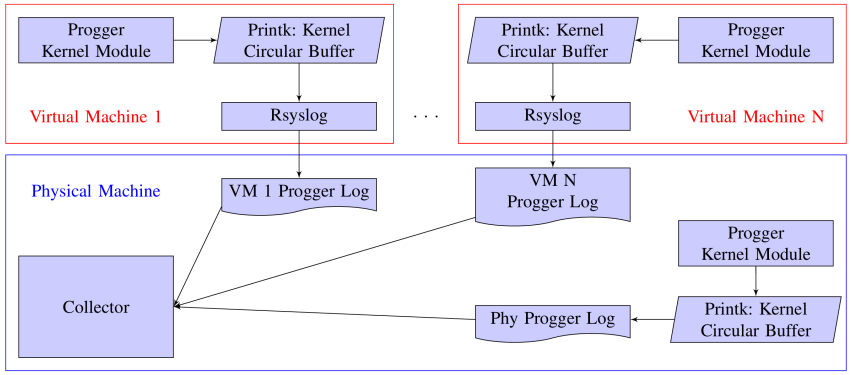


Figure 3.5: Progger architecture overview [6].

Progger's capture mechanism relies on a kernel module that dynamically locates and modifies entries in the system call table. By inserting hooks into calls such as *open*, *close*, *read*, *write*, *rename* and *chmod*. Progger intercepts all core data related activities. To log them, the system avoids direct kernel file access, instead adopting *printk* to copy information into the kernel's circular buffer, as illustrated in Figure 3.5. This buffer is subsequently processed by *rsyslog*, which filters Progger messages into dedicated log files. However, this approach is not without risks. Disabling kernel page protection to insert hooks introduces a small window of vulnerability [113], and the reliance on *printk* incurs performance costs [114].

Provenance Storage

Provenance logs are written via *rsyslog* to files, which may reside either on virtual disks or be redirected to physical hosts. The latter is preferred to mitigate the rapid growth of log files and to simplify collection. To prevent tampering, Progger restricts log file access to the *rsyslog* process, rejecting even root level attempts to open them directly. Further protection is achieved through hash chaining of log entries, which makes undetected modification extremely difficult. Nevertheless, the authors acknowledge that once root is fully compromised, absolute tamper-proofing becomes impossible. Retention is

treated cautiously, the system keeps logs concise by minimising entry size and avoiding unnecessary duplication, but it does not attempt advanced archival or tiered storage strategies.

Provenance Query

Queries over provenance in Progger are not handled natively. Instead, the logs are designed for later ingestion into analysis tools or databases. The examples presented in the research show how analysts can reconstruct user actions, trace socket activity, or follow file transfers by correlating entries. Thus, querying is largely an external concern, relying on downstream software to provide visualisations, audit dashboards, or investigative interfaces.

Evaluation

Progger was evaluated using controlled benchmarks measuring the overhead introduced when logging system calls such as *open*, *read*, *write*, and *close*. The results identified *printk* as a major source of latency, with particularly high absolute execution times for write operations due to the virtual machine's limited disk I/O performance. Although the logging overhead exhibited relatively consistent behaviour with respect to log size, its magnitude depended substantially on the underlying storage environment. The authors therefore recommended storing log files separately from the primary disk or forwarding them to the physical host to reduce I/O contention. The case studies demonstrated Progger's ability to capture detailed provenance information, including file operations and application level data activity observable through system calls. These findings support its potential as a foundation for cloud data accountability tools, while highlighting the need to address logging overhead in practical deployments.

Other Considerations

Progger's design inherently supports cross machine correlation. Because it runs within every virtual and physical machine, the logs can be joined together. The time synchronisation issue, often the Achilles' heel of distributed provenance, is addressed not by native correction within Progger, but by enabling post-hoc alignment through analysis of correlated events such as socket creation. This allows distributed logs to be reconciled into coherent causal sequences, although the burden lies with external tools rather than the logger itself.

3.9 Linux Provenance Modules (LPM)

Bates et al. introduces LPM [7], a framework for trustworthy whole-system provenance collection in Linux. The system embeds provenance capture within the kernel itself, in a manner invisible to user applications, and extends support into user space and across networks through secure, attested channels.

The central motivation for this work is trustworthiness. Whereas earlier provenance-aware systems like PASS or SPADE either operated in user space or lacked adversarial resilience, LPM explicitly addresses malicious environments. The authors recognise provenance as an attractive attack vector. An adversary may wish to forge, delete, or tamper with lineage to evade forensic tracing or to manipulate access control decisions. Unlike predecessors, LPM aims not only for completeness in capture but also tamper-resistance and verifiability, supported through TPM based attestation and SELinux based runtime enforcement. In this sense, it is the first generalised kernel level framework that treats provenance as a security critical artefact.

Bates et al. demonstrates LPM's capabilities through its implementation of Provenance-Based Data Loss Prevention (PB-DLP). A file transfer service that prevents exfiltration of sensitive data by checking provenance graphs before allowing transmission. For instance, if a file is derived from a database containing payment card details, PB-DLP blocks its release to external hosts,

even if the data has been laundered through intermediate files. The authors show empirically that such checks can be executed in tens of milliseconds, which makes PB-DLP feasible at line speed.

Provenance Model

LPM adopts the W3C PROV-DM standard, encoding provenance as directed acyclic graphs of entities, activities, and agents. This ensures interoperability with existing systems and facilitates exchange between deployments. The authors emphasise that, unlike some prior systems that employed bespoke or minimal lineage formats, adopting PROV-DM allows both standardisation and expressive semantics.

Provenance Capture

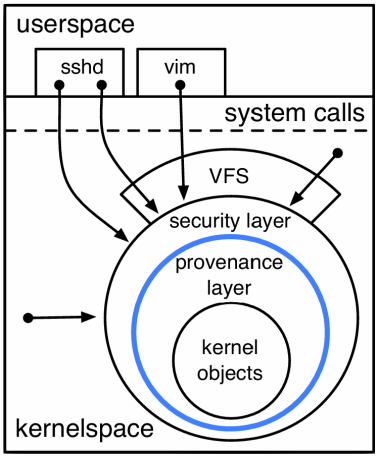


Figure 3.6: LPM architecture overview [7].

Figure 3.6 illustrates the architectural placement of the provenance capture mechanism within the Linux kernel. Userspace applications such as *sshd* or *vim* interact with the kernel exclusively through system calls. These invocations traverse the VFS and associated kernel subsystems, which are themselves mediated by the security layer. The provenance layer is positioned directly beneath the security layer, ensuring that every authorised operation, such as file access, process execution, or inter-process communication is accompanied by a

provenance event. This ordering guarantees that provenance is only recorded for actions that successfully pass access control checks, thereby aligning captured lineage with effective system semantics.

A set of provenance hooks added to the kernel, each hook is placed just beneath the corresponding LSM authorisation hook, ensuring that provenance is recorded only for authorised system operations. More than 170 hooks cover system calls and controlled data types such as files, *inodes*, sockets, IPC messages, and shared memory. For network traffic, LPM integrates Netfilter hooks and cryptographic message commitment. Digital Signature Algorithm (DSA) signatures are embedded into IP options fields, signing over immutable header fields and payload, and verified on receipt. This guarantees continuity and authenticity of distributed provenance logs. Applications may also disclose higher level workflow provenance, with integrity checked via the Integrity Measurement Architecture (IMA), which hashes binaries at load time and binds them to Trusted Platform Module (TPM) registers. Thus LPM integrates kernel events, network flows, and attested application-level disclosures into a unified, trustworthy stream.

Provenance Storage

LPM includes recorders for *Gzip* compressed logs, PostgreSQL databases, Neo4j [86] graph stores, and most importantly SNAP, an in-memory graph structure compliant with PROV-DM. Each recorder is flagged as provenance-opaque to prevent infinite loops. *Gzip* provides fast but offline storage. SNAP supports high speed graph queries, with observed insertion rates exceeding 150,000 events per second. Storage considerations are explicit, raw capture during a kernel compilation produced 3.7 GB of provenance, which compressed to 450 MB with Gzip or 1.6 GB in SNAP due to deduplication. This highlights scalability concerns, but also demonstrates that storage overheads are competitive with prior systems such as PASS.

Provenance Query

Querying is performed primarily through graph traversal. Functions like *FindAncestors* and *FindSuccessors* allow ancestry and successor analysis, which forms the foundation of PB-DLP enforcement and monitoring phases. In practice, ancestry queries on graphs with millions of nodes and edges returned in less than $21ms$, demonstrating that interactive or even inline policy enforcement is viable.

Evaluation

The authors' microbenchmark experiments showed that the LPM framework alone introduced negligible overhead for most operations, although enabling provenance collection resulted in noticeable performance costs for certain file system operations. In their application level evaluations, the authors reported slowdowns of 2.7% for kernel compilation, 7.5% for Postmark [97], and 4.8% for BLAST [98]. The authors also evaluated the efficiency of provenance ancestry queries on a graph containing approximately 6.5 million nodes, reporting a maximum query execution time of 21 ms. However, their experiments did not establish the scalability of the system to larger, long-lived provenance stores. The most significant performance overhead was observed in their evaluation of Provmon's cryptographic network message commitment protocol, where per-packet DSA signing reduced TCP throughput from 4,490 Mbps to 482 Mbps in local benchmarks. By batching signatures across ten packets, the authors improved throughput to approximately 1,600 Mbps, although this remained substantially below the baseline. In their comparison with earlier provenance systems, the authors demonstrated that LPM's Provmon module incurred lower collection overhead than PASS during kernel compilation and Postmark benchmarks, while exhibiting performance comparable to Hi-Fi and offering stronger protections against provenance tampering.

Other Considerations

Provenance is not only preserved locally but also secured across systems. Through its cryptographic Netfilter protocol, LPM ensures that provenance-aware hosts exchange tamper-evident lineage metadata. Remote attestation using TPM provides proof of kernel integrity, allowing federated trust across distributed deployments. However, the authors acknowledge limitations whereby provenance is still primarily local, and availability in distributed systems remains unsolved. They suggest replication at neighbouring hosts as a future avenue.

LPM is unable to capture all workflow layer abstractions, such as clipboard operations in *Xorg*, leading to conservative over approximations in provenance graphs. It also does not address provenance confidentiality, a recognised concern in sensitive domains. Yet in historical context, it represents the first secure, adversary resilient kernel framework for provenance, bridging the gap between high performance lineage systems and security models rooted in trusted computing.

3.10 CamFlow

CamFlow [8] is a Linux based whole-system provenance capture mechanism. It is conceived as a kernel module that can be integrated into Platform-as-a-Service (PaaS) environments, providing tenants with *Provenance-as-a-Service*. The system intercepts interactions between processes and kernel objects, along with network exchanges, in order to generate comprehensive provenance streams.

Pasquier et al. recognised that provenance systems such as PASS and LPM sought to deliver complete tracking of information flows but failed to achieve long term adoption. They either produced unmanageable overheads, became incompatible with evolving Linux kernels, or generated impractical volumes of data. CamFlow distinguishes itself by being architected with maintainability

and integration in mind. It leverages modern kernel interfaces, specifically the LSM and NetFilter frameworks, and deliberately maintains a self contained structure. Unlike its predecessors, it allows administrators to tailor provenance capture policies to specific application needs, reducing overhead and enabling practical deployment.

CamFlow demonstrates its potential by its ability to support compliance monitoring with privacy and data protection regulations. For instance, it was used to verify whether personal data flows within a cloud system respected contractual or legal constraints. Auditors subscribes to provenance streams and applied path constraints in provenance graphs to detect violations. This was implemented and tested in the context of European data protection law [115].

Provenance Model

CamFlow explicitly adopts the W3C PROV-DM standard. Entities (e.g., files, inodes, packets), activities (e.g., processes, system calls), and agents (e.g., users, groups) are represented as graph nodes. Edges represent interactions such as *used*, *wasGeneratedBy*, or *wasAssociatedWith*. To ensure property of being acyclic, kernel objects are versioned whereby every new flow into an object creates a new state-version node. For example, network packets are represented as entities with identifiers derived from immutable packet fields, ensuring consistent cross host matching.

Provenance Capture

Figure 3.7 illustrates the architecture overview of CamFlow. CamFlow builds on the LSM framework to hook into kernel events. LSM provides security hooks at two levels: (i) allocation of kernel objects (e.g., files, sockets, inodes) and (ii) access to those objects (e.g., read, write, mmap). CamFlow is implemented as a major LSM module, meaning it allocates a provenance *blob* associated with kernel objects. The provenance records are produced at system call boundaries. NetFilter hooks complement this by enabling capture of

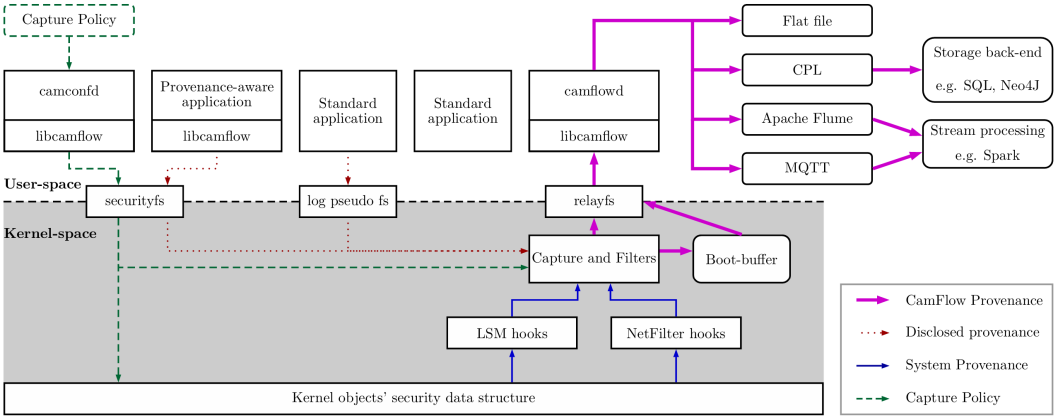


Figure 3.7: CamFlow architecture overview [8].

network packets that would otherwise be missed.

Captured data are transferred from kernel to user space through *relayfs* [116], which implements efficient per-CPU ring buffers. From there, a user-space service streams the data, in either binary or PROV-JSON format, to downstream applications using messaging middleware such as Apache Flume [117] or MQTT [118]. Tailoring mechanisms allow filtering based on object types, system calls, or user defined propagation rules, including taint tracking.

Provenance Storage

CamFlow itself does not enforce a particular storage backend but streams provenance to user space where it may be stored in SQL databases, graph databases such as Neo4J [86], or big data stores like Apache Accumulo. The native output is PROV-JSON, but a compact binary format is also available. Pasquier et al. have demonstrated that compression can reduce storage costs to under 10% of raw JSON size. Importantly, CamFlow enables selective capture, drastically reducing storage requirements when compared to all encompassing systems like PASS. Retention periods and compression are considered application level policies rather than kernel concerns.

Provenance Query

Similar to most other systems, CamFlow itself provides capture, but querying is delegated to the applications that subscribe to its streams. Provenance queries can be run against the graph representations in storage, or replaced with constant time taint checks for enforcement scenarios such as data loss prevention. This dual approach balances fine-grained retrospective analysis and low latency enforcement of security or compliance policies.

Evaluation

The authors evaluated CamFlow on three criteria: maintainability, performance, and generated data volume. Maintainability was assessed through comparisons with PASS and LPM, demonstrating that CamFlow required modifications to substantially fewer kernel files and could be maintained across multiple Linux versions with relatively little effort. Performance evaluations using *LMbench* [119] and macrobenchmarks (Postmark, Phoronix tests, Redis, Memcached, and Apache) showed generally modest overhead. While most microbenchmark slowdowns were below 20%, certain system calls experienced considerably higher overhead, reaching 100% for *stat* under whole-system capture. Macrobenchmark overhead ranged from negligible to 22%, depending on the workload, with selective capture further reducing overhead in several cases. Finally, data volume experiments demonstrated that tailoring capture policies could reduce generated provenance data by approximately 20%, illustrating the trade-off between capture completeness and application-specific utility.

Other Considerations

CamFlow supports distributed provenance by treating network packets as first-class entities. Through consistent identifiers across machines, packet flows can be stitched into end-to-end provenance graphs. This facilitates preservation of provenance across hosts in cloud environments, making dissemination to

tenants feasible. Applications can thus integrate provenance streams across containers, VMs, and machines.

The central limitation of CamFlow is that tailoring capture undermines completeness. Selective provenance is inherently partial and risks omitting critical flows if policies are over- or under-specified. Moreover, while the LSM hooks provide strong coverage, later research showed that certain information flows may escape capture, prompting ongoing refinements to the LSM infrastructure [120]. CamFlow represents the third generation of whole-system provenance capture, following the PASS, Hi-Fi, and LPM. Its importance lies in realising a maintainable and deployable design.

3.11 Progger 3

Progger 3 is the third iteration of the Progger system, introducing notable advancements over the original Progger 1. While Progger 2 served as an intermediate version, it was never publicly released or documented, and therefore, specific details about its features or developments remain unavailable. As a result, this section focuses on the enhancements made in Progger 3 relative to its original implementation (Progger 1), as these are the only versions with accessible and verifiable information.

The motivation behind this third iteration is rooted in the shortcomings of its predecessors. Earlier versions either introduced instability, relied on user space components that could be tampered with, or imposed performance overheads so severe that they became unusable in real workloads. Progger 3 distinguishes itself by enforcing a kernel-only operation, integrating TPMs, and employing strong cryptography. This design decision is moves away from partial kernel–user space hybrids towards kernel-only approaches addresses decades old debates around provenance trust boundaries.

```
1 { "id": "openat", "tp_src": "sys_exit", "ts": 8901772142832, "ret":
    6, "pid": 12300, "uid": 0, "euid": 0, "args": [ 4294967196,
    140735053210000, 591872, 0, 140735053211440, 32 ], "strings": [
    "\/etc\/gss\/mech.d" ] }
2 { "id": "setresuid", "tp_src": "sys_exit", "ts": 8901772503698,
    "ret": 0, "pid": 12300, "uid": 105, "euid": 105, "args": [ 105,
    105, 105, 7, 94176267341920, 94176267341824 ], "strings": [ ] }
3 { "id": "setuid", "tp_src": "sys_exit", "ts": 8901772511526, "ret":
    -1, "pid": 12300, "uid": 105, "euid": 105, "args": [ 0, 0, -1,
    7, 94176267341920, 94176267341824 ], "strings": [ ] }
4 { "id": "setresuid", "tp_src": "sys_exit", "ts": 8901772513575,
    "ret": -1, "pid": 12300, "uid": 105, "euid": 105, "args": [ -1,
    0, -1, 7, 94176267341920, 94176267341824 ], "strings": [ ] }
5 { "id": "openat", "tp_src": "sys_exit", "ts": 8901826793176, "ret":
    27, "pid": 217, "uid": 0, "euid": 0, "args": [ 4294967196,
    140736600249456, 524288, 0, 0, 1 ], "strings": [
    "\/proc\/12299\/comm" ] }
6 { "id": "openat", "tp_src": "sys_exit", "ts": 8901827120638, "ret":
    -2, "pid": 217, "uid": 0, "euid": 0, "args": [ 4294967196,
    140736600249184, 524288, 0, 0, 1 ], "strings": [
    "\/run\/systemd\/units\/log-extra-fields:ssh.service" ] }
```

Figure 3.8: Example: Progger 3 JSON logs [9].

Provenance Model

Progger 3 does not adopt W3C PROV, OPM, or any existing formal provenance ontology. Instead, it defines a bespoke binary format optimised for kernel efficiency, later translated into JSON for consumption at the server side. Figure 3.8 is an example of such JSON log. This reflects a divergence from much of the provenance research community, which has often emphasised interoperability over raw performance.

Provenance Capture

The capture mechanism is strictly kernel based. Progger 3 attaches to Linux tracepoints for *sys_enter* and *sys_exit* events, allowing it to record system call identifiers, arguments, return values, process and user IDs, and associated C-string arguments. These are placed into structured records, first with a compact binary header and then a detailed body containing the captured metadata. Importantly, it avoids capturing arbitrary pointer structures due to the complexity and risks of serialisation at the kernel level, opting instead for integers and strings to balance completeness with stability.

Provenance Storage

Storage in Progger 3 is network oriented rather than local. Provenance records are encrypted and authenticated using *XChaCha20-Poly1305* and transmitted over Transmission Control Protocol (TCP) to a remote server, which is assumed to operate in a trusted environment. The server stores the received provenance and can present it in JSON. This design choice, where confidentiality and integrity in transit are enforced through TPM protected cryptographic keys, shows a shift towards provenance as a distributed and auditable service rather than a local log file. Considerations such as retention and long term archiving are left largely to the server side, indicating that Progger 3 focuses on secure capture and transport rather than lifecycle management.

Provenance Query

Querying is not conducted within the kernel itself. Instead, the provenance records are output as JSON on the server, leaving the burden of querying and analysis to external systems or scripts. While secure capture is robust, analytic usability remains relatively underdeveloped compared to systems like PASS, which integrated storage and query mechanisms.

Evaluation

Corrick benchmarked Progger 3 by compiling the Linux 5.8.3 with and without tracing, measuring both throughput and latency. The results show that the overhead of Progger 3 is significantly lower than that of Progger 1 and 2, to the point that it can be realistically deployed. Latency increases are measurable when tracing system calls, but for untraced calls the penalty is negligible. The correctness of TPM usage is also formally demonstrated, strengthening its claim to tamper-proof.

Other Considerations

By encrypting data in transit and ensuring keys cannot be extracted from user space, Progger 3 enables multiple clients to transmit tamper-proof provenance to central servers. In this sense, provenance is preserved across machines, though interoperability with other provenance systems or models is not addressed.

Prooger 3 does not attempt to cover the full data management pipeline, but it makes a decisive contribution by showing that provenance can be captured securely, with cryptographic protection, and with acceptable performance on general workloads. This careful balance between pragmatism and rigour defines its place in the broader literature on provenance.

3.12 A Word About eBPF

At the time of writing, only limited systems explicitly identify themselves as provenance systems that utilise eBPF. A notable example is ProvBPF, which was introduced not as a standalone system or paper, but as a proof-of-concept use case within the saBPF [121] framework to demonstrate the feasibility of using eBPF for whole-system provenance capture. However, provenance research has historically drawn upon techniques such as LSM, system call interception, or kernel instrumentation as its principal implementation choices, and these approaches remain dominant in mature systems that self describe as provenance frameworks.

Nonetheless, there is a discernible trend in which systems performing tasks adjacent to provenance, such as monitoring, tracing, or enforcing runtime policies, are increasingly moving towards eBPF. The transition reflects both the demand for higher performance and the reduced intrusiveness that eBPF provides compared to traditional kernel modifications.

Examples of this trend can be seen in domains adjacent to provenance research. Security oriented frameworks such as Cilium [122] leverage eBPF

for network observability and fine grained enforcement, offering capabilities that were previously implemented through mechanisms like NetFilter or kernel hooks. Similarly, Falco [123] has incorporated eBPF as a backend for system call monitoring, replacing earlier reliance on kernel modules. Tracee [124], developed by Aqua Security, is another example of a system employing eBPF to capture system events for security observability, highlighting a shift away from older kernel patching and LSM based approaches. These systems, although not describing themselves as provenance frameworks, nonetheless capture and analyse execution time information in ways that resemble provenance style monitoring.

Taken together, these developments indicate that while provenance frameworks have only just begun to adopt eBPF, with ProvBPF serving as an early proof-of-concept. The broader systems community is already embracing eBPF as the mechanism for monitoring and enforcement. This trajectory suggests that provenance research is likely to benefit from further integration with eBPF based infrastructures, potentially inheriting both the efficiency and ecosystem support that these developments offer. In this respect, eBPF represents not only a technical opportunity but also a conceptual alignment with ongoing trends in related areas of systems design.

3.13 Discussion & Conclusion

The exploration of provenance systems in this chapter has highlighted an enduring limitation in the way that versioning is captured and represented. Current approaches to provenance tend to treat versions primarily as structural devices. Their existence is recorded to resolve issues such as cycles in provenance graphs, but they remain abstract placeholders, unaccompanied by information about the actual state of the data at those moments in time. As such, the record of a version is closer to a conceptual marker than to an evidential resource. This treatment renders provenance less effective in addressing ques-

tions that users most likely pose, such as *what happened to my data, how did it change, and what state was it in at a particular stage of its life cycle*? By restricting versioning to a mechanism for graph consistency, rather than recognising it as an opportunity for temporal awareness, provenance has historically limited its capacity to provide true explanatory power.

The earliest provenance systems were built with the objective of ensuring reproducibility and accountability. They concentrated on mapping processes and dependencies, creating abstract histories that allowed researchers to trace back to inputs and methods. Over time, these systems evolved towards standardisation and interoperability, but the conceptual treatment of versions remained largely static. Versions marked that a change occurred but were silent on the nature of that change. In this respect, provenance systems have remained decoupled from the lived experience of data, where users and organisations require more than an assurance that something altered. They seek clarity on what the alteration entailed, when it took place, and why it matters. Without temporal awareness in records of versions, provenance cannot serve as a reliable witness to the trajectory of data.

This chapter has also considered a second deficiency in the prevailing discourse on proximity between provenance and data. A recurring theme in the literature is the recognition that provenance should be kept close to data [93], which has led to the development of provenance at the data layer. While this shift has been significant, it has not gone far enough. Provenance situated alongside data remains an external artefact, and therefore it is vulnerable to entanglement with other records, inconsistency of retrieval, and difficulties of dissemination. Queries made over such provenance often necessitate complex disentanglement before meaningful insight can be extracted. The burden on systems and users alike is unnecessarily heightened by this design choice.

Embedding provenance within the data itself presents a more radical but necessary step. By embedding provenance, each piece of data carries with it its own lineage, untangled from the histories of unrelated artefacts. Such

embedding ensures that provenance is portable, enabling the dissemination of both data and its history as a unified entity. It also streamlines querying, since the scope of a query is automatically bounded by the data instance itself, reducing the risk of misinterpretation or accidental conflation with other records. In this sense, embedding restores to provenance the quality of integral with data that is demanded for effective interrogation and understanding.

The significance of these observations lies in their implications for the future of provenance systems. A system that merely acknowledges the existence of versions but remains agnostic to their substance cannot fully support critical data-centric enquiries. Similarly, a system that keeps provenance adjacent to, rather than embedded within data, cannot ensure independence, clarity, or portability of records. Addressing these shortcomings requires reconceptualisation. Provenance must not only be structurally consistent but also temporally expressive; it must not only be co-located with data but inseparable from it.

In the next chapter, attention turns from conceptual limitations to the realities of practice. The discussion will examine the adoption of provenance systems in real-world contexts, assessing how the principles outlined in these earlier chapters translate into operational deployments. This shift in focus enables a critical evaluation of the extent to which existing systems meet the requirements of transparency, accountability, and usability, and it highlights the gap between theoretical capability and practical implementation.

Chapter 4

Mainstream Adoption of Provenance Systems

As covered in previous chapters, despite decades of research efforts resulting in numerous data provenance systems and frameworks, developed within the confines of academic research, their broader adoption beyond these contexts has remained limited. In this chapter, we aim to examine the current scope of deployment for such systems and evaluate the practical applications where they have been effectively implemented. Specifically, our analysis will address two key objectives: first, assessing the extent to which provenance systems are adopted as enterprise solutions; and second, identifying the use cases that have demonstrated benefits in real-world scenarios. Of particular interest are domains that typically present strong justification for provenance tooling, namely, those governed by regulatory frameworks and subject to audit requirements. These include areas such as data protection [125], privacy policy enforcement [126], and compliance verification [115], where provenance is not just a technical nicety but a functional necessity. The analysis is guided by **RQ1**: *What are the drivers, barriers, and technological enablers influencing data provenance adoption in enterprise environments?*

4.1 Methodology

This chapter examines the extent to which data provenance systems have progressed beyond academic research into enterprise deployment. The investigation addresses RQ1 by examining three aspects of adoption: the regulatory and operational requirements that motivate provenance capture, the technical capabilities demonstrated by existing implementations, and the barriers that prevent these capabilities from being integrated into enterprise workflows. A structured review of academic and grey literature provides the basis for this analysis.

The inclusion of grey literature is necessary because enterprise provenance technologies are frequently documented through vendor specifications, implementation guides, technical reports, and industry standards rather than peer-reviewed publications. However, the availability of a provenance feature does not necessarily establish its adoption or effectiveness in production environments. The review therefore distinguishes between proposed provenance mechanisms, commercially available capabilities, and documented evidence of operational deployment.

4.1.1 Literature Search

The literature search drew upon Google Scholar [1], the University of Waikato Library search engine [2]. Specialist scholarly databases, particularly the ACM Digital Library [3], IEEE Xplore [4], USENIX Proceedings [5], arXiv [6], and Springer Nature Link [7], provide additional coverage of research published in computer science, software engineering, database systems, and information security.

The search strategy was organised around three concepts: provenance tech-

[1] https://scholar.google.com/
[2] https://waikato.primo.exlibrisgroup.com
[3] https://dl.acm.org
[4] https://ieeexplore.ieee.org
[5] https://www.usenix.org/publications/proceedings
[6] https://arxiv.org
[7] https://link.springer.com

nologies, enterprise adoption, and regulatory or operational requirements. The following Boolean search expressions specify the relevant combinations of these concepts:

- (“data provenance” OR “data lineage”) AND (“enterprise” OR “industry” OR “production”)
- (“data provenance” OR “data lineage”) AND (“adoption” OR “deployment” OR “implementation”)
- (“data provenance” OR “data lineage”) AND (“compliance” OR “auditability” OR “regulatory”)
- (“data provenance” OR “data lineage”) AND (“GDPR” OR “HIPAA” OR “SOX”)
- (“data provenance” OR “data lineage”) AND (“healthcare” OR “finance” OR “cloud computing”)
- (“provenance” OR “build provenance”) AND (“software supply chain” OR “SLSA”)

These expressions cover the principal technologies, application domains, and regulatory requirements examined in this chapter. The use of both “data provenance” and “data lineage” is important because enterprise platforms frequently employ the latter term to describe functionality associated with data origin, transformation, and traceability.

The scholarly search was complemented by targeted searches of primary industry and regulatory sources. Advanced search operators, for example *site:aws.amazon.com*, were used to locate technical documentation from cloud providers. Relevant sources included documentation from Amazon Web Services (AWS), Snowflake, and Google Cloud, together with regulatory and standards documentation concerning GDPR, HIPAA, SOX, NZISM, and SLSA.

Primary regulatory and standards documentation was considered separately from academic interpretations of those requirements. This distinction

was necessary to establish whether provenance was explicitly required by a regulation or standard, or merely proposed in the academic literature as a mechanism for satisfying broader requirements concerning auditability, accountability, and traceability.

4.1.2 Literature Selection

The literature was selected according to its contribution to understanding the drivers, barriers, and technological enablers of enterprise provenance adoption. Sources were considered relevant where they addressed at least one of the following: (i) the implementation of provenance systems in operational environments; (ii) the application of provenance to regulatory compliance; (iii) the technical or organisational limitations affecting deployment; or (iv) the capabilities of existing commercial and research based provenance technologies.

Academic publications were examined for proposed provenance models, implementation architectures, experimental findings, and identified limitations. Industry documentation was examined for evidence of implemented functionality, supported deployment environments, integration requirements, and operational constraints. Regulatory documents and industry standards were examined to determine the extent to which their requirements establish a practical need for provenance capabilities.

A distinction was maintained between conceptual proposals, prototype implementations, and operational systems. Studies demonstrating provenance functionality in experimental environments were relevant to establishing technological feasibility, but were not treated as evidence of widespread enterprise adoption. Similarly, vendor documentation describing an available feature was considered evidence of technological capability, rather than independent evidence of its operational effectiveness or prevalence across organisations.

This distinction is important to RQ1, as the existence of mature provenance models in academic research does not, by itself, demonstrate their adoption in enterprise environments.

4.1.3 Quality Assessment of Grey Literature

Grey literature requires particular scrutiny because technical documentation, corporate reports, and industry publications may not undergo independent peer review. The AACODS framework [127], comprising Authority, Accuracy, Coverage, Objectivity, Date, and Significance, was identified as the basis for evaluating the credibility and relevance of these sources.

The six dimensions were considered in relation to the specific evidence required by this chapter:

Authority concerns whether the information originates from an organisation or author with direct responsibility for, or recognised expertise in, the technology, regulation, or standard being examined. Primary documentation from technology providers and regulatory authorities is particularly relevant when establishing system capabilities or formal requirements.

Accuracy concerns whether technical and regulatory claims are supported by specifications, implementation details, references, or verifiable documentation. Claims concerning enterprise adoption require evidence beyond a description of the technology's intended functionality.

Coverage concerns whether a source provides sufficient information about the provenance capabilities under examination, including their scope, supported environments, limitations, and relationship to enterprise requirements.

Objectivity concerns the purpose of the publication and the potential for commercial or institutional bias. Vendor documentation may accurately describe a product's functionality while providing limited evidence of deployment challenges, implementation costs, or comparative limitations.

Date concerns the publication date and currency of the information. This is particularly important for cloud platforms and software supply chain technologies, where product capabilities and security specifications evolve, and for regulatory documentation that may be amended.

Significance concerns whether the source contributes directly to answering RQ1 by establishing a regulatory driver, demonstrating an available prove-

nance capability, documenting practical deployment, or identifying an obstacle to adoption.

These criteria provide a basis for distinguishing authoritative technical evidence from promotional or insufficiently substantiated claims. In particular, the assessment of industry documentation must account for the difference between evidence that a provenance capability exists and evidence that organisations have successfully adopted it.

4.2 Compliance, Security and Data Provenance

Regulatory compliance and information security represent significant drivers for the adoption of data provenance systems in enterprise environments. Across sectors such as healthcare, finance, and government, organisations are required to demonstrate accountability for how information is accessed, processed, transformed, and retained. Although regulatory frameworks do not necessarily mandate the use of dedicated provenance systems, their requirements for auditability, traceability, and data integrity align closely with the capabilities that provenance technologies seek to provide.

This section examines the relationship between data provenance and selected regulatory and security frameworks, including the General Data Protection Regulation (GDPR), the Health Insurance Portability and Accountability Act (HIPAA), the Sarbanes-Oxley Act (SOX), and the New Zealand Information Security Manual (NZISM). These frameworks illustrate how different regulatory environments establish requirements for accountability and information governance, while also introducing practical constraints on the collection and retention of provenance metadata.

4.2.1 General Data Protection Regulation

The General Data Protection Regulation (GDPR), has become a cornerstone of data privacy law within the European Union and significantly impacts data

governance and the processing of personal data. Its core objective is to empower individuals with greater control over their personal data while imposing strict obligations on organisations that process such data. The regulation emphasises principles such as transparency, accountability, data minimisation, purpose limitation, and the right to erasure or portability, among others. These principles collectively represent a fundamental shift towards responsible data stewardship, compelling enterprises to rethink the way data is collected, processed, and shared.

While data provenance is not explicitly stated as a requirement, the ethos of provenance is embedded in various provisions of the GDPR, which call for mechanisms to ensure that personal data is processed lawfully, transparently, and with respect to the rights of individuals. Provenance records appears naturally suited to support purpose limitation by verifying that data is not repurposed beyond its originally intended use without proper consent. Similarly, in the context of data subject access requests (DSARs), provenance can assist in locating all instances and derivations of a subject's data across complex systems [128].

In the literature, Ujcich et al. [43] developed a provenance model that interprets GDPR's rights and obligations through the lens of the W3C PROV-DM model. Their work emphasises temporal reasoning and traceability, offering a detailed mapping of GDPR concepts, such as consent, access, erasure, and portability onto provenance constructs. They define a set of specialised agents, entities, and activities that correspond to the roles and operations described in the regulation (e.g., data subjects, controllers, consent activities, and personal data entities). The work describes how provenance can be used to verify compliance. For example, they show how to validate whether a data processing activity occurred within the validity period of a given consent, or whether a withdrawal of consent has been respected by ceasing related activities. Their provenance graphs thus serve as both a record of compliance and a runtime mechanism for enforcing constraints on data usage.

Building upon Ujcich et al.'s model, Campagna et al. [129] propose a set of extensions aimed at improving the model's coverage of GDPR obligations. Their analysis identifies specific gaps in the original model, such as the representation of data retention periods, compatibility of processing purposes, and obligations for special categories of data and introduces eleven extensions to address these. Notably, they incorporate temporal validity annotations for consent and introduce new subclasses for special types of personal data (e.g., genetic, biometric, health-related) and justifications (e.g., public interest, legal obligation). Other use cases have also been explored and proposed, including data flow auditing [20], informed consent traceability [130] and data privacy protection [131].

Discussion

Based on a broad reading of academic and industry sources, is that, despite the theoretical maturity and frequent citation of provenance as an aid to GDPR compliance, its deployment in enterprise-grade systems remains rare. This is likely due to the inherent complexity and computational cost of capturing and managing provenance at scale. Legacy systems often lack the necessary instrumentation for provenance capture, and retrofitting them can be technically disruptive. Additionally, the legal status of provenance data remains ambiguous, particularly when it includes identifiers or metadata about individuals, introducing new compliance risks under GDPR.

Critically, the act of tracking detailed data provenance may itself introduce new privacy risks under GDPR. If a provenance system records all transformations and derivations of a dataset, especially in collaborative, multi-user environments, it may inadvertently reveal sensitive information about data subjects or processors. For instance, provenance logs may include usernames, IP addresses, or transformation parameters that qualify as personal data (Recital 30). In such cases, provenance logs themselves become subject to GDPR and must be protected.

This introduces a second conflict as a result of the principle of data minimisation under the GDPR, which stands in contrast to the core objectives of provenance systems, where comprehensive traceability and accountability are prioritised. Provenance capture and retention can conflict with the obligation to delete personal data once it is no longer necessary. While there is currently no literature that explicitly addresses this issue, existing work on provenance confidentiality [132, 133, 134] and privacy [135] offers partial mitigation. Techniques that ensure confidentiality while not directly targeting data minimisation, do help limit unnecessary exposure of personal data. Notably, some approaches support confidential provenance that remains auditable by authorised parties [136, 137]. However, these methods are not widely adopted and remain challenging to implement correctly in practice.

Further complications arise when a data subject exercises their rights to access, rectification, or erasure, over data that is deeply embedded within provenance chains. If their personal data appears in intermediate derivations, feature transformations, or aggregated statistics, it becomes technically and semantically challenging to determine whether such data falls under the scope of a DSAR, and how it should be surfaced or redacted without corrupting the broader provenance graph. Current provenance systems, especially those built atop relational or graph-based databases, are not well equipped to handle this granularity of access control or subject specific data projection.

The central argument of this thesis is that existing provenance systems are predominantly centralised and opaque to data subjects. Provenance metadata is typically confined to back-end systems, accessible only to administrators or auditors, and not available to data subjects. This lack of transparency violates the GDPR's core principle of data subject empowerment but also contributes to the limited adoption of provenance systems in practice. When provenance remains inaccessible and unusable to those it is meant to benefit, its perceived value diminishes.

This line of reasoning strengthens our research objective and reinforces the

central thesis that embeds provenance directly within the data it describes, rather than relying on centralised logging. This design introduces clear isolation and separation of concerns. Provenance becomes inherently accessible to data owners, is decoupled from unrelated provenance records, and remains usable even when individual data or provenance elements are deleted. This prevents corruption of the broader provenance landscape and strengthens the system's transparency, resilience, and trustworthiness.

4.2.2 Health Insurance Portability and Accountability Act

In the healthcare sector, the Health Insurance Portability and Accountability Act (HIPAA), enacted in the United States, governs the use and protection of sensitive patient information, ensuring privacy, security and compliance. Its objectives are rooted in ensuring data confidentiality, integrity, and availability, while simultaneously guaranteeing that data access and sharing do not occur outside of strict regulatory boundaries. HIPAA is not solely concerned with securing health records but also with ensuring that patient rights over their own data are respected.

HIPAA, particularly through its Security Rule (45 CFR §164.312(b)), mandates audit controls to record and examine access to electronic Protected Health Information (ePHI). While it stops short of explicitly mandating data provenance systems, it strongly implies the necessity of fine-grained logging and traceability that provenance inherently provides.

Data provenance has emerged in the literature as a promising mechanism to address the regulatory and security challenges of healthcare systems, particularly in the context of HIPAA compliance. Jaigirdar et al. [44] argue that IoT-based healthcare infrastructures typically lack real-time, end-to-end visibility regarding security and regulatory conformance. In response, they extend the PROV-DM provenance model into PROV-IoT [138], by incorporating security metadata at each stage of the data lifecycle. This includes embed-

ding contextual attributes such as device state, communication protocols, and authentication methods, enabling step-wise verification through digital signatures. Their framework allows for continuous risk estimation and compliance tracking.

Other work explores how provenance can be augmented to support both security and privacy in distributed healthcare settings. Tharaud et al. [139] propose a hybrid approach that combines data provenance with digital watermarking to protect EHRs. Provenance captures the history of data use, transformation, and access, while watermarking embeds persistent, invisible identifiers directly into the records, binding data to users and contexts. Velez [140] extends this trajectory by identifying shortcomings in W3C PROV [141] and Health Level 7 (HL7) Fast Healthcare Interoperability Resources (FHIR) and proposed a domain-specific, extensible provenance model for eHealth systems. Their model integrates fine-grained data tracking, dynamic consent management, and enhanced security controls.

Discussion

Based on a broad review of academic and industry sources, is that, despite the theoretical maturity of data provenance and its frequent citation as a facilitator of HIPAA compliance, its implementation in enterprise-grade systems remains limited. Although provenance is often discussed in principle, we have identified claims in the literature indicating that provenance information is either unavailable or, when present, obscured [142]. Furthermore, the consequences of insufficient provenance have been documented. For instance, the Surgisphere case, in which datasets of unclear provenance were provided to researchers, ultimately led to the retraction of articles in the New England Journal of Medicine and The Lancet [143, 144].

Similarly, data provenance in its current implementation risk violating HIPAA regulation if poorly implemented. Provenance records may inadvertently reveal sensitive information such as access patterns or role hierarchies,

which could be exploited to identify patients or infer diagnoses. This conflicts with the *minimum necessary* principle under HIPAA, which stipulates that only the least amount of information necessary to achieve a purpose should be disclosed or recorded.

Retention periods are also a factor. HIPAA mandates that access logs be retained for six years (45 CFR §164.316(b)(2)(i)). This implicitly affects how long provenance metadata must be stored if it is used to meet audit requirements. Long-term storage introduces costs and risks, particularly when the volume of provenance data is high. Instead of being isolated, current systems adopt a centralised approach, forming a dependency graph that interlinks entities, activities, and agents within a fine-grained chronology. Deleting data or its associated provenance naively can sever critical edges in this graph, leading to broken references and corrupting the integrity of the remaining provenance chain. This undermines both trust and verifiability, the very properties provenance is supposed to ensure.

The study of data provenance in the context of HIPAA and the healthcare sector reaffirms its potential to support regulatory compliance. However, the challenges identified in the literature, including the protection of sensitive provenance metadata, long-term storage requirements, and the preservation of provenance integrity during data deletion, suggest that practical deployment remains complex. These challenges may present barriers to broader adoption, particularly in healthcare environments where strict privacy and auditability requirements must be satisfied.

4.2.3 Sarbanes Oxley Act

The Sarbanes-Oxley Act of 2002 (SOX) [145] was enacted in response to a series of high-profile financial scandals, notably Enron, WorldCom, and Tyco. These events exposed deficiencies in corporate governance and financial accountability, leading to widespread loss of investor confidence. SOX's core objective is to enforce transparency, accountability, and reliability in financial

reporting by mandating rigorous internal controls and disclosure standards for publicly traded companies operating in or listed in the United States.

Section 302 of SOX requires senior corporate officers to personally certify the accuracy of financial statements, while Section 404 mandates that companies establish and maintain adequate internal controls over financial reporting, with both management and external auditors assessing their effectiveness. Despite this synergy, SOX does not explicitly mandate the implementation of data provenance or lineage tracking technologies. Instead, it specifies requirements for internal controls, audit trails, and data retention without prescribing how these must be technically realised. SOX requires financial records, including audit logs, to be retained for at least seven years under Section 802.

In the literature, the importance of data provenance for regulatory reporting in financial institutions is well established [146, 125, 147]. Appelbaum et al. [148] examined SOX in relation to its implications for audit evidence provenance in Big Data environments. Big Data Provenance Black Box (BDPBB) was proposed. This conceptual system is inspired by aircraft black boxes and would serve as a secure, tamper-evident repository of provenance data for Big Data workflows. Built on existing research and systems like HadoopProv [149], the BDPBB would use digital signatures and role based access to ensure the integrity and security of the data's history. Appelbaum et al. argues that without robust provenance systems like the proposed BDPBB, organisations risk SOX non-compliance due to unreliable and unauditable external data sources.

Discussion

The adoption of data provenance systems for SOX compliance in enterprise remains limited. Large enterprises continue to rely on traditional data governance tools, access logs, and change management systems, which when combined with manual oversight and tightly defined reporting processes, are typically sufficient to satisfy audit requirements [150, 151].

From a technical perspective, data provenance is more easily aligned with

the control and audit requirements of SOX than with those of GDPR or HIPAA. SOX places a strong emphasis on internal controls, traceability, and auditability. All areas where provenance systems can deliver immediate value. In contrast, GDPR and HIPAA impose stricter privacy constraints and external accountability, making the implementation of provenance more complex due to the need to manage sensitive personal data and ensure minimal exposure.

However, similar to GDPR and HIPAA, provenance also introduces risk. Exhaustive provenance trails can inadvertently reveal sensitive data flows and user behaviours, clashing with principles like *minimum necessary access* as defined in HIPAA, GDPR, and indirectly in SOX itself. Without strict access controls or secure handling of provenance, provenance metadata becomes a liability rather than a compliance asset. Moreover, for provenance data to be of practical use during audits, it must be presented through intuitive, interactive tools [148]. Current systems lack this usability; they are not designed with auditors in mind, nor do they integrate well with existing compliance workflows.

Ultimately, while provenance has clear potential to strengthen SOX compliance, existing implementations are not yet mature or usable enough for widespread deployment. This functional gap continues to stall its adoption.

4.2.4 New Zealand Information Security Manual

The New Zealand Information Security Manual (NZISM) serves as a foundational framework guiding information security practices across government agencies in New Zealand. While its primary focus is on protecting official information from unauthorised access, modification, or disclosure, the manual's requirements indirectly address elements central to data provenance, even if the term itself is not explicitly mentioned. The NZISM emphasises principles such as accountability, auditability, and integrity, which form the bedrock of tracking data origins and transformations which are key aspects of provenance.

For instance, the Accountability principle mandates that systems must be capable of identifying and authenticating users, ensuring actions can be traced to specific individuals or processes. This requirement aligns with provenance's goal of recording who interacted with data at each stage of its lifecycle.

The NZISM further requires robust audit trails, stipulating that logging mechanisms capture details such as user identities, access times, and system changes. These logs inherently document parts of a dataset's journey, though they often focus on security events rather than the comprehensive lineage needed for detailed provenance analysis. Similarly, data integrity controls in the NZISM ensure that information remains unaltered unless authorised, reinforcing trust in its validity over time which is a concern closely tied to provenance when assessing whether data has been tampered with or misused. However, these measures are framed within a security context rather than an explicit focus on documenting the full history of data generation, transformation, and use.

The closest approximation to provenance in the NZISM is found in its Asset Management controls, which require agencies to catalog information assets systematically. This includes tracking metadata such as classification levels, custodianship, and retention periods. While this does not equate to a granular provenance record, it establishes foundational practices for identifying data origins and managing lifecycle stages. The manual's emphasis on risk management also indirectly supports provenance principles by requiring agencies to assess risks associated with data handling, though such assessments typically prioritise confidentiality and availability over lineage specific concerns.

Discussion

From an analytical perspective, the absence of explicit provenance requirements in the NZISM reflects a broader trend where regulatory frameworks often lag behind technical concepts like data provenance. While audit trails and accountability mechanisms are critical components, they do not fully ad-

dress the nuances of tracking data transformations or integrating provenance into operational workflows. This gap is particularly evident when considering emerging challenges such as AI-driven decision-making, where understanding how algorithms process and alter datasets becomes imperative for transparency and compliance.

While the NZISM's existing controls provide a partial framework that can be adapted to support provenance practices, its lack of explicit requirements risks leaving critical gaps in accountability and traceability. For instance, without standardised guidelines for capturing lineage metadata or integrating provenance into system design, agencies may inconsistently implement these practices, undermining their effectiveness in audits or incident investigations. Future iterations of the NZISM could strengthen data governance by adopting provenance standards, aligning with global trends toward enhanced transparency and regulatory demands such as those under GDPR. Such integration would not only bolster New Zealand's information security posture but also position its frameworks to address evolving threats where data lineage becomes inseparable from trustworthiness and compliance.

4.3 Software Supply Chain

Supply-chain Levels for Software Artifacts (SLSA) [152] is a framework for securing the software supply chain. It defines a set of standards and practices to ensure the integrity and provenance of software, from source code through build systems to the final distributed artefact. SLSA provides progressive levels of assurance, where each level enforces stricter controls around build reproducibility, source verification, dependency management, and tamper resistance. Its goal is to make software supply chains more transparent, verifiable, and resistant to attacks such as code injection or build compromise.

The integration of data provenance into the SLSA framework is neither uniform nor absolute across its defined maturity tiers, reflecting instead an in-

cremental progression in requirements that culminates in explicit demands for structured and verifiable provenance at higher levels. SLSA's core objective to establish a standardised hierarchy of security guarantees for software artifacts naturally aligns with the principles underlying data provenance, as both seek to ensure trustworthiness by documenting lineage and integrity. However, the treatment of provenance within SLSA varies significantly depending on the level under consideration. At Level 0, where no meaningful guarantees are provided, provenance is entirely absent; artifacts may lack even basic metadata about their construction processes. By Level 1, foundational controls emerge, such as automated build pipelines and access restrictions to source repositories, yet provenance remains implicit rather than explicit. While the build process must be recorded at this stage, these records often take the form of logs or version control history without standardised structures or cryptographic guarantees, making them less reliable for verifying artifact integrity in a decentralised ecosystem.

The progression toward formalised provenance becomes evident in Level 2, where SLSA introduces requirements for reproducibility and authenticated inputs. Here, build processes must capture sufficient detail to recreate artifacts from their source materials, and dependencies are verified against trusted registries. This level begins to approximate the core tenets of provenance by mandating traceability, albeit still falling short of formal provenance assertions. The metadata generated at this stage might include timestamps, commit hashes, or dependency versions but lacks the cryptographic signing mechanisms necessary for tamper resistance a hallmark of robust provenance systems. It is not until Level 3 that SLSA explicitly enshrines data provenance as a requirement. At this highest level, artifacts must be accompanied by signed provenance metadata that comprehensively documents their build environment, inputs, and execution steps. This signed artifact often generated through tools like The Update Framework (TUF) [153] or cosign [154] serves as an authoritative record of the artifact's lineage, enabling independent ver-

ification of its integrity and origin. The cryptographic binding between the artifact and its provenance ensures that trust in the builder is transferable to downstream consumers, a critical feature for secure supply chains.

The distinction between these levels highlights SLSA's pragmatic approach: it does not universally mandate provenance but instead ties its requirements to escalating security guarantees. This tiered structure accommodates organisations with varying risk postures and resource capacities while setting aspirational targets for those prioritising top-tier security. For instance, a company deploying Level 3-compliant artifacts can confidently assert that their software's lineage is both documented and cryptographically verifiable, whereas teams operating at lower levels may rely on less formal assurances. The closest approximation to provenance in earlier stages such as build logs or dependency manifests is insufficient for high-stakes scenarios due to the absence of non-repudiation mechanisms.

SLSA's nuanced treatment of provenance strikes a balance between practicality and rigor but leaves room for improvement. While Level 3's requirements align with best practices in modern software supply chain security, mandating provenance at lower levels could accelerate broader adoption by establishing baseline expectations. The current framework risks creating a dichotomy where only organisations with significant resources can achieve the highest guarantees, potentially marginalising smaller entities that might still benefit from foundational provenance features. Additionally, SLSA's focus on cryptographic signing for provenance at Level 3 is commendable but could be further augmented by standardising formats (e.g., Software Package Data Exchange (SPDX) or In-Toto [155]) to enhance interoperability. Ultimately, SLSA positions data provenance as a critical component of secure software development but frames it within a maturity model that acknowledges the trade-offs between security and operational complexity a reflection of its design philosophy prioritising adaptability over universality.

4.3.1 Google Cloud Build Provenance

Google's Cloud Build Provenance [156] is a feature within Google Cloud's Build system that automatically generates verifiable metadata about how a given artifact was built. This metadata includes details such as the source repository, build steps, build tools, dependencies, and the environment in which the build was executed. The primary function of Cloud Build Provenance is to enhance transparency and trust in the software development process by providing a cryptographically verifiable record of the build process, thereby ensuring that the final artifact has not been tampered with or modified in an unintended manner.

The primary use case for Google's Cloud Build Provenance is in securing the software supply chain. Software is increasingly composed of numerous third-party dependencies and built in distributed environments, ensuring the integrity of the final artifact is critical. Cloud Build Provenance enables organisations to trace the lineage of a software artifact, verifying its origin and the build process it underwent. This becomes particularly important in industries with stringent security and compliance requirements, such as finance, healthcare, and government sectors, where unverified or tampered software could introduce significant security vulnerabilities.

The core problem that Cloud Build Provenance aims to solve is the lack of visibility and verifiability in the software build process. Traditionally, software builds have relied on implicit trust, with little to no cryptographic assurance that an artifact was produced from a specific source in a controlled manner. This implicit trust model has been exploited by attackers through supply chain attacks, such as dependency confusion and malicious code injection. By generating and storing provenance metadata, Cloud Build Provenance allows organisations to enforce stronger security policies, such as verifying that only trusted build environments and dependencies were used in producing a given artifact. This reduces the attack surface and improves overall software security posture.

As for its adoption, Cloud Build Provenance is seeing increased usage, particularly among organisations leveraging Google Cloud's broader security initiatives. It integrates well with Google's Binary Authorisation [157], which enforces policies on containerised applications before deployment, ensuring that only artifacts with verified provenance are allowed to run in production environments. While adoption has been growing, it is still largely concentrated within organisations that already prioritise DevSecOps and software supply chain security. The broader industry, while recognising the importance of provenance, is still in the process of adapting to these new security paradigms, often hindered by legacy development workflows and the perceived complexity of implementing provenance tracking at scale.

Cloud Build Provenance is a crucial development in the landscape of mainstreaming provenance, but it also highlights the challenges in widespread adoption. While Google has made a compelling case for provenance in software builds, the adoption gap between cloud-native enterprises and traditional organisations remains substantial. Many companies still lack the necessary infrastructure or expertise to leverage provenance effectively, which suggests that broader adoption will require more industry standard tooling, better integration with existing workflows, and increased education on the importance of provenance. Therefore, provenance must extend beyond build artifacts to encompass the entire data lifecycle, from raw data to final analytical insights, in order to fully realise the benefits of data provenance in a cloud first world.

4.3.2 DevOps in Amazon Web Services

Amazon Web Services (AWS) has developed the Well-Architected Framework DevOps Guidance [158] to assist organisations in adopting DevOps best practices. This guidance is structured into several "DevOps Sagas," each focusing on core domains within the software delivery process. Within these sagas are specific capabilities, each identified by unique codes, such as [AG.ACG.8] and [AG.DLM.9].

[AG.DLM.8] focuses on improving traceability with **data provenance** tracking. This guidance advocates for automated metadata capture throughout the data lifecycle, from origin through transformation to consumption, enabling fine grained auditing and reproducibility of results. In complex environments, such as distributed data lakes or machine learning pipelines, the absence of data lineage impairs debugging, compliance, and trust in outputs. Integrating lineage tools like AWS Glue or SageMaker Lineage allows practitioners to maintain a reliable record of data manipulations and actor involvement, which is particularly critical under regulatory regimes such as GDPR or HIPAA as seen in previous sections. Such provenance frameworks provides verifiable data integrity and bolster the operational resilience of data-centric architectures.

[AG.ACG.9] refers to the integration of **software provenance** tracking throughout the development lifecycle. This guidance emphasise the necessity of end-to-end tracking of software artefacts to mitigate supply chain risks and ensure component integrity. By incorporating Software Bill of Materials (SBOMs) [159, 160], cryptographic hashes, and digital signatures into the development pipeline, provenance can be enforced at each stage, from source code commits through builds and deployments. This becomes increasingly essential in environments with third-party dependencies or open-source components, where vulnerabilities and tampering risks propagate quickly. Tools supporting Software Composition Analysis (SCA) and automated verification mechanisms are instrumental in detecting unauthorised changes, enforcing compliance, and ensuring artefact trustworthiness. Together, these practices form a critical foundation for secure, transparent, and verifiable software delivery processes.

The Log4j vulnerability (Log4Shell) [161] and the SolarWinds Orion supply chain attack [162, 163] represent two of the most consequential security incidents in recent history, both stemming from a systemic lack of visibility into software components and their provenance. In the Log4j case, a critical zero-day exploit in a widely used open-source logging library went undetected

across thousands of systems for years. Many organisations had no reliable way to trace which software components, especially transitive dependencies, were embedded in their applications or infrastructure. Even when Log4j was present, they often could not identify whether it was the vulnerable version or if it was reachable in a way that exposed the exploit. Similarly, the SolarWinds breach involved a malicious code injected into the build pipeline of the Orion network management software, which was then distributed to over 18,000 customers, including government agencies and Fortune 500 firms.

It is evident from these two events that, while guidance and tools for software and data provenance exist, they are not widely used in practice. Had comprehensive provenance tracking been in place, covering dependency metadata, build artefacts, and system lineage, the impact of these breaches could have been drastically reduced, and recovery timelines significantly shortened. These failures expose a critical operational blind spot in modern software systems.

4.4 Coalition for Content Provenance and Authenticity

The Coalition for Content Provenance and Authenticity (C2PA) [164] is developing open technical standards for digital content provenance, aiming to strengthen trust in media authenticity amid growing concerns about misinformation and synthetic media. Established by Adobe, Arm, Intel, Microsoft, BBC, and Truepic, with close alignment to the Content Authenticity Initiative (which includes partners such as The New York Times), C2PA's mission is to provide interoperable specifications that allow content creators and distributors to embed cryptographically verifiable, tamper-evident provenance metadata into digital assets. This design avoids dependence on proprietary systems or blockchain-based infrastructures, instead relying on established cryptographic methods such as digital signatures, hashing, and Merkle trees.

Users can verify provenance through emerging tools like Adobe's Content Credentials, while broader industry adoption is encouraged by the standard's emphasis on openness, simplicity, and integration into existing media workflows. Unlike distributed ledger approaches, C2PA prioritises accessibility and interoperability, seeking to make provenance verification practical across diverse creative and journalistic contexts.

Despite its technical ingenuity, the coalition faces significant challenges in achieving mainstream penetration. The heterogeneous nature of digital ecosystems complicates interoperability; while Adobe's Photoshop now supports embedding C2PA-compliant metadata, many legacy systems lack native integration, forcing users to rely on third-party plugins or manual workflows that hinder seamless adoption. Further, the absence of regulatory mandates means participation remains voluntary, leaving companies to weigh the costs of implementation against perceived benefits. For instance, social media platforms, critical battlegrounds for content verification, have been slow to adopt C2PA standards comprehensively. As of 2024, three years after C2PA's formal inception in 2021, major social media platform adoption has remained limited, with Meta only then announcing its participation in the coalition [165].

Case studies of C2PA adoption show both promise and constraint. Reuters' proof-of-concept with Canon and Starling demonstrated that capture-time signing and provenance tracking can strengthen trust in photojournalism, but the pilot also introduced latency at the camera and required adjustments to established editorial workflows [166]. Adobe's integration of Content Credentials into creative software has lowered technical barriers by making provenance metadata easier to attach, yet evidence from industry reports suggests that awareness and consistent use remain limited, with many professionals still prioritising speed and delivery over metadata management under deadline pressure [167]. Taken together, these experiences highlight a recurring challenge: technical feasibility does not ensure routine adoption. Verification tools are often poorly understood outside specialist circles, and without stan-

dardised interfaces or broad support in mainstream platforms such as social media, the visibility and perceived value of provenance information for general audiences remain constrained.

Regulatory landscapes send mixed signals about C2PA's future. The EU's Digital Services Act (DSA), for instance, stresses transparency and accountability in online platforms but does not mandate particular technical standards, leaving firms free to interpret compliance in diverse ways. This flexibility may foster innovation but also risks fragmentation if companies develop incompatible provenance solutions. By contrast, sectors subject to strict audit and liability requirements, such as healthcare or financial reporting, are accustomed to traceability mechanisms and may be more inclined to adopt C2PA style provenance systems proactively, even though explicit mandates are not yet in place. This creates a paradox: heavily regulated industries may lead in deploying provenance technologies, while the social media and content sharing environments most exposed to misinformation remain under regulated and therefore slower to integrate such tools.

Looking ahead, C2PA's trajectory will depend on how effectively it adapts to emerging technologies and user expectations. The spread of AI-generated media has made provenance mechanisms essential for distinguishing synthetic from authentic content, and the standard now includes extensions designed for generative AI workflows. Scaling these innovations, however, requires collaboration beyond early adopters. Platforms such as TikTok and Meta have begun experimenting with Content Credentials, but consistent integration across the broader social media ecosystem remains uncertain. Equally important is public understanding. Without clear user-facing labels and literacy efforts that explain how provenance metadata works, technical advances risk going unnoticed by the very audiences they are meant to inform.

Critically assessing C2PA's progress reveals both promise and persistent gaps. Its industry led governance has ensured that standards align with practical workflows, yet widespread adoption depends as much on cultural change

as on technical capability, a shift that has been slower to materialise. While the specifications provide a solid foundation, mainstream uptake will require coupling technical innovation with behavioural design so that verifying provenance becomes nearly as seamless as sharing or downloading content. C2PA's greatest contribution may lie less in enforcing universal compliance than in offering a credible framework that encourages broader industry convergence. As misinformation continues to undermine trust, the standard could evolve into a de facto baseline, not through regulation, but because participants perceive value in mitigating reputational and legal risks. Whether this outcome materialises, however, will hinge on overcoming inertia and extending partnerships beyond the coalition's current core. The path forward thus demands not only technical refinement but sustained collaboration among creators, platforms, regulators, and the public to transform provenance from a niche feature into an expected attribute of digital content.

4.5 Cybersecurity

In cybersecurity, provenance information forms a critical foundation for understanding complex attack behaviours that traditional detection systems often miss. Because APTs typically use stealthy methods [168], blending malicious activities with normal system operations (living off the land), provenance offers a way to map and identify subtle changes in behaviour over time. It allows defenders to reconstruct attack chains and gain situational awareness, which is crucial for identifying sophisticated intrusions that unfold over weeks or months.

In the literature, Hassan et al. implemented NODOZE [169], a system that leverages data provenance to construct concise causal dependency graphs for each alert and assigns anomaly scores based on the rarity of system event patterns in historical enterprise data. These scores are propagated through the graph using a novel network diffusion algorithm, allowing NODOZE to auto-

matically rank alerts and highlight the most anomalous, and therefore likely malicious behaviours. The system's effectiveness is demonstrated through deployment in a real enterprise, where it reduced false positives by 84%, significantly decreased graph sizes, and saved over 90 analyst hours per week, all without sacrificing detection accuracy.

This is one of the closest systems we have observed that has been tested in a real, active enterprise network, making it particularly compelling. While the attacks themselves were partly simulated, comprising known APT scenarios and malware samples, the evaluation was conducted on live infrastructure at NEC Labs America, across 191 operational hosts generating genuine enterprise activity. This setup provides a rare and valuable bridge between academic prototypes and practical deployment.

Buchta et al. [170] provides a comprehensive analysis of the current landscape of advanced persistent threat (APT) detection systems based on data provenance. The authors highlight a critical shortcoming: despite extensive research efforts and promising laboratory results, provenance based systems have yet to achieve deployment in real world environments. They explicitly state that "recognition performance has not yet been achieved outside laboratory conditions," and their analysis repeatedly emphasises the gap between experimental success and operational deployment. Their findings reveal fundamental challenges, including difficulties in adaptability, integration into operational infrastructures, and the maintenance of provenance integrity in dynamic, large scale settings. These deficiencies suggest that while provenance holds significant theoretical value for security applications, its current practical implementations fall short of the requirements demanded by operational contexts.

The review reveals that a central reason for this gap is the fragility of provenance linkage to the data it describes. Provenance information, in the systems reviewed, is collected externally via provenance trackers like CamFlow and managed separately from the primary data objects. The review describes

how provenance is recorded through system auditing tools and structured into graphs, but also highlights that the integration layer between provenance collection and broader system operations is often weak or missing. This separation means that if files are moved, modified, or archived, the corresponding provenance may be lost, corrupted, or rendered incomplete. The problem of adaptability was discussed, noting that updates and system extensions must not cause permanent alarms and must be considered in the analysis by the detection system for correct detection. Such a challenge becomes significantly more manageable if provenance travels inseparably with the data it describes, rather than existing in a separate store vulnerable to desynchronisation.

To overcome these issues, it becomes evident that close association between data and provenance, specifically through the embedding of provenance information within files themselves, is essential. Embedding ensures that provenance remains inseparably linked to the data, preserving its integrity across system evolutions. Furthermore, supporting versioning of both data and its provenance would enable systems to capture the dynamic history of data objects, thus allowing for resilient provenance even when systems undergo updates, migrations, or partial failures. Buchta et al. acknowledge the difficulty of maintaining provenance through system evolution, pointing out that existing detection systems lack extensibility and adaptability layers capable of adjusting to operational change. Embedding and versioning directly respond to this deficit, ensuring that even as environments shift, the connection between data and its historical record remains intact.

4.6 Conclusion

The analysis in this chapter has shown that the adoption of provenance systems cannot be understood solely through the lens of technical capability. While the literature highlights an impressive breadth of use cases, ranging from compliance and reproducibility to accountability and optimisation, the realities of

enterprise adoption reveal a more complex picture. Provenance technologies sit within existing organisational, regulatory, and infrastructural landscapes, and it is those landscapes that ultimately determine whether provenance is valued or sidelined.

The discussion has surfaced three central forces. First, **drivers** such as regulatory mandates, sector-specific accountability requirements, and the strategic pursuit of data governance create conditions where provenance is not just desirable but necessary. Second, **barriers** including the overhead of integration, uncertainty over return on investment, cultural resistance to transparency, and the absence of compelling short-term incentives all contribute to a persistent reluctance to deploy provenance solutions at scale. Third, **technological enablers** such as cloud-native architectures, distributed ledger technologies, and standardised provenance models suggest that the cost of adoption may be lowered and the value of provenance better articulated in the near future.

Taken together, these insights indicate that the path towards widespread enterprise adoption of provenance systems will not be dictated by technical progress alone. Success will require the alignment of technology with regulatory environments, economic incentives, and organisational practices. The trajectory of provenance adoption is therefore contingent, shaped as much by institutional readiness and governance frameworks as by the sophistication of the underlying systems.

By answering **RQ1**, this chapter has shown that the future of provenance depends on recognising its role as a socio-technical intervention rather than a purely technical artefact. The next stages of this thesis will build on this understanding, exploring how provenance research can more directly engage with adoption contexts, and how design choices might anticipate and respond to the institutional realities in which provenance systems are ultimately embedded.

Chapter 5

Temporal Data Evolution

Earlier chapters have shown how provenance has been defined, modelled, captured, and stored, shaped both by research prototypes and regulatory demands. Existing approaches generally fall into two types. The first, exemplified by Progger, ignores versions entirely and records only transformation flows, as if data were fixed. The second, represented by systems such as PASS, does acknowledge versions, but only symbolically, as markers to prevent cycles in a directed acyclic graph. In both cases, the temporal dimension of provenance remains unexplored, with versions reduced to abstract placeholders rather than meaningful records of change.

It is against this trajectory that the problem of temporal data evolution arises. To understand provenance as more than a record of transitions, one must engage with the content of change itself. True versioning, therefore, is not a mere technical device for graph management or continuity markers, but a way to embed the evolving identity of data within its provenance. This chapter addresses these limitations directly by examining temporal data evolution and arguing for provenance systems that record versions as substantive content.

5.1 Motivating Example

Consider a scenario where a file, denoted as A, undergoes changes before being used to derive other files. Initially, a user concatenates file A into file B (`cat`

`A > B`), recorded as the provenance relationship A > B. Subsequently, file A is modified before another concatenation operation produces file C (`cat A > C`), yielding A > C.

A system such as Progger, which does not account for versions, will treat both relationships in a uniform manner. It will simply record that B is derived from A and that C is derived from A. The fact that A had one state when B was created and a different state when C was created is lost entirely. The provenance graph therefore gives the impression that both B and C are derived from the same instance of A, even though in practice they come from different historical states.

A system such as PASS, on the other hand, does introduce version markers, but only in a symbolic sense. It would represent B as derived from A version 1 and C as derived from A version 2. However, these versions are not enriched with information about the actual changes that took place in A, nor are they used to capture the semantics of temporal evolution. They exist primarily to preserve the acyclic property of the provenance graph by ensuring that a later use of A cannot be conflated with an earlier use. As a result, PASS avoids the misleading suggestion that B and C came from the same state of A, but it does not provide a deeper temporal account of how A changed between the two operations.

A similar issue arises in the context of software compilation and execution. Imagine compiling a program using the command:

```
gcc main.c -o myprog -lpthread
```

The compiler produces an executable, *myprog*, which depends on the POSIX threads library. At compile time, the dependency is not embedded in the binary; instead, the binary merely carries a reference to the shared object *libpthread.so*. The actual resolution occurs later, at runtime, when the dynamic linker binds *myprog* to whichever version of *libpthread* is available in the environment.

Here the importance of temporal provenance becomes clear. The executable *myprog* remains identical across time, yet its behaviour may change depending on which version of *libpthread* is linked at the moment of execution. If *libpthread* were upgraded, patched, or, more problematically, compromised after the compilation, then subsequent executions of *myprog* would reflect these changes. Provenance records that simply capture the fact that *myprog* was compiled with `-lpthread` do not suffice. What is required is a record that ties each execution of *myprog* to the precise version of *libpthread* that was loaded at that time, together with the verifiable lineage of that library's binary artefact.

This scenario illustrates why provenance must be treated as temporally indexed. Existing provenance systems such as CamFlow indeed make an important step forward by recording the versions of dependencies involved in an execution. Their model allows one to observe that a particular execution of *myprog* was bound to *libpthread.so* at version *v1*, and that later executions may have been bound to *v2*. This level of granularity is valuable because it reveals that the environment in which *myprog* ran has changed. However, the semantics of that change remain opaque. The system can flag that the library has altered between two points in time, but it cannot explain what those alterations mean in practice. A version identifier, detached from a deeper record of lineage and build history, functions more as a label than as an explanation.

If provenance is to provide genuine accountability, temporal indexing cannot stop at identifiers alone. Recording only that *myprog* executed with *libpthread.so* at *version 0.9.33* does little more than establish a correlation. It does not answer the critical question of whether this version was benign or malicious, whether it was built from the canonical upstream repository, or whether it had been silently backdoored and redistributed. To understand that, the provenance system must capture more than symbolic markers. It must also bind those markers to cryptographic attestations of the build artefact, to the precise commit from which the artefact was produced, to the toolchain used in its compilation, and ideally to an auditable chain of custody.

By contrast, a provenance system that treats time as integral rather than incidental would enable a much stronger form of accountability. It would not only record that *myprog* executed with *libpthread.so.0.9.33*, but also that this binary was derived from commit X, signed by developer key Y, compiled on date D using a reproducible toolchain, and subsequently distributed through package manager Z. Such a system transforms version identifiers from tokens into verifiable checkpoints along a temporal trajectory. The difference is that the former can only indicate that a change has occurred, while the latter can explain what that change entails and whether it undermines trust.

Ultimately, provenance must evolve from documenting *that* change has taken place to articulating *what* the change is to allow us to understand *why* it matters. Without this shift, provenance systems risk devolving into little more than bookkeeping, noting differences without providing the semantic depth needed to resist supply chain attacks as described in previous chapter. Temporal provenance, enriched with full lineage and context, is the only viable path to accountability in environments where binaries are linked, loaded, and replaced across time.

The need for temporally indexed provenance is not confined to a particular operating system. The underlying challenge arises wherever software artefacts depend on external components whose states may evolve independently of the artefacts that consume them. Although the mechanisms for resolving dependencies and observing execution differ across operating systems, the fundamental requirement remains the same: to associate each execution with the precise dependency artefacts used at that point in time, together with their verifiable lineage. Consequently, a provenance model that separates temporal semantics from operating system specific mechanisms could provide a general foundation for software supply chain accountability across heterogeneous computing environments.

5.2 Versioning

Versioning, in the context of software development and systems engineering, refers to the systematic tracking and management of changes made to digital artifacts over time [171]. Put simply, version control enables developers to record incremental modifications to files or datasets, ensuring that each iteration is preserved with metadata such as timestamps, authors, and descriptive summaries. This process transforms transient work into a structured history, allowing teams to inspect past states, roll back to previous versions, and collaborate without fear of irreversible data loss or conflicts. Early approaches to versioning involved rudimentary methods like manually appending suffixes (e.g., *file_v2.txt*) or copying directories, but these practices became unsustainable as projects grew in complexity and team size. Modern version control systems (VCS) emerged to address these limitations, providing robust mechanisms for tracking changes at the atomic level while enabling scalable collaboration across distributed teams.

Centralised version control systems (CVCS), such as Subversion (SVN), introduced a single authoritative repository where all users checked out code and submitted updates through a server. While this model simplified access control and historical tracking, it suffered from single points of failure and constrained offline workflows. Distributed version control systems (DVCS), notably Git, revolutionised the field by decentralising repositories, allowing each user to maintain a full copy of the project history. This architecture not only eliminated reliance on centralised servers but also empowered developers to work independently committing changes locally, branching freely, and merging divergent histories without immediate network access. The introduction of DVCS marked an important shift in how teams manage codebases, emphasising flexibility, concurrency, and fault tolerance while enabling advanced workflows like forking and pull requests that forms modern open source collaboration.

A core principle of versioning is the concept of atomic commits, where

each change is encapsulated as a self-contained unit with associated metadata. These commits form a directed acyclic graph in DVCS, representing both linear progressions and divergent branches. Branches allow developers to isolate feature development or experimentation from the primary codebase, reducing disruption while enabling parallel workstreams. Merging these branches later requires resolving potential conflicts through semantic understanding of code differences, as automated tools can only detect syntactic overlaps. The DAG structure also supports advanced operations like rebasing (reapplying commits onto a new base) and cherry-picking (selectively applying individual commits), which are critical for maintaining clean history and coordinating contributions across teams.

Beyond tracking source code, versioning principles extend to infrastructure configuration, documentation, machine learning models, and even binary assets. Systems such as Git LFS (Large File Storage) or specialised tools like DVC (Data Version Control) address the limitations of traditional VCS with large datasets, ensuring that non-textual artifacts are versioned efficiently without bloating repositories. Advanced workflows often integrate versioning with continuous integration/continuous deployment (CI/CD) pipelines, where tagged releases and semantic versioning (e.g., v2.1.0) automate testing, staging, and production deployments. Challenges persist in managing repository size, merge complexity at scale, and the need for fine-grained access control in security-sensitive environments. Despite these hurdles, versioning remains foundational to reproducible engineering practices, enabling traceability from commit to deployment while fostering collaboration in an increasingly distributed world.

5.2.1 Semantic Versioning (SemVer)

SemVer provides a standardised convention for assigning version numbers to software releases, addressing the ambiguity and inconsistency associated with ad hoc versioning practices. Rather than treating version identifiers as arbi-

trary labels, SemVer establishes a structured relationship between the version number and the nature of changes introduced during the development lifecycle. Under the conventional format *MAJOR.MINOR.PATCH*, a version number such as *2.4.1* communicates whether a release introduces breaking changes, adds functionality in a backward-compatible manner, or consists of backward-compatible bug fixes. This approach enables developers, maintainers, and downstream users to assess the potential impact of an update before adopting it, thereby improving transparency and predictability in software evolution.

The three components of a semantic version number have distinct meanings. The *MAJOR* component is incremented when incompatible changes are introduced to the public application programming interface (API) or other explicitly defined interfaces. For example, changing a function signature in a way that breaks existing client code may require a major version increment, such as moving from *1.5.2* to *2.0.0*. The *MINOR* component is incremented when new functionality is added in a backward-compatible manner, such as introducing an optional parameter or a new API endpoint. For instance, a release from *1.5.2* to *1.6.0* indicates an extension of functionality without intentionally breaking existing interfaces. The *PATCH* component is incremented for backward-compatible bug fixes, including corrections to existing behaviour that do not alter the public interface incompatibly. Consequently, a release from *1.5.2* to *1.5.3* communicates a more limited change than a minor or major release. When a project is still in its initial *0.y.z* development phase, the public interface is generally considered unstable, and breaking changes may occur before version *1.0.0* is reached.

An important feature of SemVer is its relationship with backward compatibility and dependency management. Software projects frequently depend on external libraries, frameworks, or services, and an update to one dependency can introduce failures in otherwise functional systems. SemVer helps mitigate this risk by providing a consistent signal about the expected compatibility of a release. Dependency managers can use version constraints to specify which

releases are acceptable, such as permitting compatible minor and patch updates while excluding major versions that may introduce breaking changes. For example, a project that depends on a library within the *1.x* series may accept updates from *1.4.0* to *1.7.2*, provided that the library follows SemVer and the dependency constraint permits those versions. However, semantic version numbers represent a compatibility contract rather than an absolute guarantee: incorrectly classified changes, undocumented behavioural differences, or insufficient testing can still introduce regressions. Consequently, version constraints should be complemented by automated testing, dependency locking, and integration validation.

SemVer also complements version control systems by distinguishing the history of individual changes from the identity of a released artefact. Whereas a Git commit identifies a particular state of a repository's history, a semantic version communicates the intended compatibility and significance of a release. Multiple commits may contribute to a single release, and a release can be associated with a Git tag to provide a stable reference to the corresponding source code. Pre-release identifiers, such as *2.0.0-alpha.1*, *2.0.0-beta.1*, and *2.0.0-rc.1*, further distinguish experimental, testing, and release-candidate builds from stable releases. Build metadata, such as *2.0.0+build.15*, can record additional information about how an artefact was produced without affecting its precedence under the SemVer specification. These conventions support traceability between source code, release artefacts, and deployment environments.

Within continuous integration and continuous deployment (CI/CD) pipelines, SemVer can be incorporated into automated release workflows. Changes merged into a repository can be tested and classified according to their intended impact, with release automation tools determining whether a major, minor, or patch increment is appropriate. In projects that adopt structured commit conventions, such as Conventional Commits, commit messages can provide evidence for this classification, although automated tools still depend on accurate developer annotations and appropriate release policies. Once a release is pro-

duced, its semantic version can be used to label packages, container images, and deployment artefacts, supporting consistent identification across development, testing, and production environments. Nevertheless, SemVer is most effective when combined with clearly documented public interfaces, disciplined compatibility policies, and automated verification. It does not replace the detailed change history maintained by a version control system; rather, it provides a higher-level abstraction that communicates the significance of changes and helps coordinate software evolution across dependent components.

5.2.2 The Need for Versioning

Versioning, in the context of data provenance, refers to the ability to track multiple iterations or states of an object over time, allowing for the identification of changes and transitions in the data or process it represents. Rather than only capturing the relationship between two objects at a single point, versioning enables the recording of all the states that an object undergoes throughout its lifecycle. This provides a historical view in which each version of an object is explicitly associated with the period in which it was active or relevant, offering a more comprehensive perspective on its evolution.

Applying versioning to data provenance can address the limitations of temporal awareness. Existing systems record relationships and metadata, but the detail of an object's state at the moment an event or transformation occurs is often limited. Some systems, such as PASS and CamFlow, maintain a notion of versioning, though the level of granularity is restricted. Others, such as Progger, do not incorporate versioning as a concept. It could be argued that post-processing techniques, such as writing a script to infer versions from timestamps, may provide a partial solution, although this remains indirect. By incorporating versioning more directly, each event can be associated not only with a specific object but also with its precise state at that time. This ensures that provenance reflects the nuances of how an object evolves, offering a richer and more informative account of its history.

Without Temporal Awareness

Represent provenance as:

$$A \xrightarrow{\text{used by}} P \xrightarrow{\text{generates}} B$$

or simply:

$$B \leftarrow P(A)$$

Suppose A changes over time:

$$A^{(1)} \rightarrow A^{(2)} \rightarrow A^{(3)}$$

The provenance graph still records:

$$A \rightarrow P \rightarrow B$$

Therefore, when querying the provenance of B:

$$\text{prov}(B) = A$$

With Temporal Awareness

Represent each historical state explicitly:

$$A^{(1)} \xrightarrow{\Delta_1} A^{(2)} \xrightarrow{\Delta_2} A^{(3)}$$

where Δ_i represents the change producing the next state:

$$A^{(i+1)} = A^{(i)} \oplus \Delta_i$$

Now suppose P uses $A^{(2)}$:

$$A^{(1)} \xrightarrow{\Delta_1} A^{(2)} \xrightarrow{\Delta_2} A^{(3)}$$

$$A^{(2)} \xrightarrow{\text{used by}} P \xrightarrow{\text{generates}} B$$

The provenance query becomes:

$$\text{prov}(B) = A^{(2)}$$

and therefore:

$$\text{state}(A, P) = A^{(2)}$$

Figure 5.1: Comparison of provenance representation with and without temporal awareness.

The need for versioning in temporal provenance arises from the distinction between an object's identity and its state at the time of an operation. As illustrated in Figure 5.1, conventional provenance records that process P uses object A to generate B, without distinguishing which historical state of A was involved. If A undergoes multiple modifications, the provenance query still returns A, obscuring the actual state that contributed to the derivation of B.

Temporal awareness addresses this limitation by explicitly representing each historical state as a distinct version, $A^{(1)}, A^{(2)}, A^{(3)}$, together with the changes Δ_i that produce successive states. Consequently, when process P consumes $A^{(2)}$, the provenance of B can be associated with that precise version rather than the logical object A alone. This distinction is fundamental to temporal provenance, as it preserves the relationship between an operation and the state of the data at the moment of its execution.

Versioning is therefore not merely a mechanism for retaining historical copies or maintaining provenance graph structure. It provides the foundation for accurately tracing data evolution, identifying the states involved in

transformations, and understanding how changes influence subsequent derived artefacts.

There are several key considerations when attempting to materialise this versioning idea in provenance systems. First, there must be a way to efficiently store and access multiple versions of an object. This requires careful design of the underlying database or storage system to support versioned data, ensuring that both new and old versions can be retrieved in a meaningful and efficient manner. Furthermore, the challenge lies in ensuring that the system not only captures versioning but also enables querying that reflects these temporal relationships. This could involve the creation of specialised queries that allow users to access an object's state at any given time or explore how an object has changed through different versions.

Another consideration involves defining the granularity of versioning. Should **every** change to an object be captured as a new version, or should changes only be recorded when they meet a certain threshold of significance? Balancing this trade-off is essential to avoid overwhelming the system with too many versions, while still preserving the granularity necessary for detailed temporal analysis. Additionally, how versions are identified and indexed becomes crucial in maintaining the integrity and accessibility of the provenance records over time.

Despite its potential benefits, introducing versioning into provenance systems comes with challenges. One of the primary obstacles is the sheer complexity of managing and maintaining a versioned system, especially as the amount of data and number of objects grows. Ensuring consistency across different versions, particularly when multiple events occur simultaneously or affect the same object, can be difficult. Furthermore, versioning introduces additional overhead in terms of both storage and computational resources. The system must handle increased data volume, ensuring that performance remains optimal even as the number of versions expands.

Another challenge lies in ensuring that versioning does not become too

rigid or cumbersome. While it is critical to capture enough detail, there is a fine line between recording meaningful information and creating a system that is difficult to manage or navigate. Too many versions or overly granular changes can lead to a situation where the provenance records themselves become difficult to interpret or query effectively.

In conclusion, the inclusion of versioning in provenance systems promises to address the temporal awareness gap, allowing for richer, more dynamic records of data's evolution. However, implementing versioning effectively requires careful thought on storage, granularity, and system design, and it comes with its own set of challenges. Nonetheless, if executed effectively, versioning has the potential to transform how we understand and utilise data provenance, enabling a deeper and more accurate understanding of data workflows over time.

5.2.3 Versioning Semantics: Snapshot vs. Derivation

In versioned systems, two dominant semantics often surface: snapshot and derivation. They represent fundamentally different ways of treating versions, both in how they are produced and in how they are interpreted.

A snapshot is an immutable capture of the state of an artefact at a particular point in time. It is context independent, meaning the snapshot carries all of the necessary content or metadata to be self sufficient. Retrieving a snapshot does not require replaying prior operations or recomputing lineage; it is a complete, frozen record. This model privileges stability and archival integrity. However, it can become costly in systems where state is large, since each snapshot risks duplication unless some form of structural sharing or deduplication is introduced.

A derivation, by contrast, encodes a transformation from one version to another. It is relational rather than absolute. A derivation makes explicit the process by which a new version comes into existence, whether through patching, delta encoding, or higher-level transformations such as applying an oper-

ator over data. This captures lineage and causality, which makes it valuable for auditing, reproducibility, and reasoning about provenance. The trade-off is fragility, whereby accessing a version requires materialising the derivation chain, and loss or corruption of an intermediate state can break access to descendants.

In practice, systems often take a hybird approach. For example, long-term checkpoints are materialised snapshots, while fine-grained changes between checkpoints are tracked as derivations. This allows both efficient storage and robust reconstruction without sacrificing too much in either direction.

5.2.4 Binary Equivalence and Artefact Identity

An additional consideration in temporal versioning is the distinction between artefact identity and binary equivalence. Although versioning systems typically associate changes with distinct version identifiers, the existence of different versions does not necessarily imply that their underlying binary representations or observable behaviours differ. Conversely, artefacts associated with the same source code revision or release identifier may exhibit differences in their binary representations due to variations in compilation environments, toolchains, build configurations, or embedded metadata.

Binary equivalence can be considered at different levels [172]. At the most restrictive level, two artefacts are byte-for-byte equivalent when their binary contents are identical. Such equivalence can be established through direct comparison or, with appropriate cryptographic assumptions, by comparing content hashes. However, binary identity should not be conflated with functional equivalence. Two binaries may differ in their internal representations while exhibiting equivalent observable behaviour under a specified execution environment. Establishing this stronger behavioural relationship generally requires additional analysis or verification beyond comparing binary contents.

This distinction has important implications for temporal provenance. A provenance system that records only version identifiers may fail to recognise

that differently labelled artefacts contain identical binary content. Equally, a system that relies exclusively on binary hashes can establish content identity but cannot, by itself, determine whether distinct binaries are behaviourally equivalent or explain their respective build histories. Consequently, binary equivalence should be considered alongside version identifiers, derivation relationships, and build provenance when reasoning about the evolution and identity of software artefacts over time.

5.2.5 Related Work: Elephant File System

The Elephant File System (EFS) is a file system that focuses on retaining the history of all user files, making every operation on files reversible [173]. Its key contribution is the integration of versioning at the file system level, which contrasts sharply with the conventional approach where only the latest version of a file is stored, and older versions are lost upon modification or deletion. This makes EFS particularly relevant in the context of data provenance, as it ensures that historical data is automatically retained, and every change in a file's contents is tracked in a way that is easily retrievable.

EFS shares similarities with our proposed temporal aware provenance systems in that both aim to capture the changes over time and preserve a history of data. Just like in data provenance, where each event or transformation is logged, EFS automatically records versions of files, which can be referenced at any point in the future by specifying a timestamp in the file path. This approach not only ensures that file deletions and overwrites are reversible but also makes it possible to view the state of a file at any moment, similar to how provenance systems offer insights into how data is transformed over time. However, EFS's focus is on file-level versioning and management, while provenance systems typically track the flow and transformation of data through workflows or processes, encompassing more than just the file state.

A significant difference between EFS and typical data provenance systems lies in the scope of their functionality. While data provenance is concerned

with tracking data across a variety of systems and processes, EFS is primarily concerned with file storage and ensuring that files' contents are retained across time. EFS also adds complexity in managing storage space efficiently. It allows users to specify retention policies for files, ensuring that storage is used wisely while still preserving important versions. Traditional file systems, in contrast, rely on the user to manually manage backups and versions, a process that is error-prone and inefficient.

One of the key challenges in the design of EFS is the issue of efficiently managing large amounts of versioned data without overwhelming the system's storage capacity. To address this, EFS incorporates a set of storage management policies. The file system uses user-specified policies to decide which versions of files to retain, which can be grouped into categories such as (i) Keep One, (ii) Keep All, or (iii) Keep Landmarks. These policies help guide the retention process based on the importance of the file versions, preventing unnecessary versions from taking up valuable storage. This type of version management is more flexible than the checkpointing approaches of systems like Andrew File System (AFS) [174] or Plan-9 [175], which apply the same retention rules across the entire file system. EFS also reduces the need for excessive file duplication by automatically managing versions based on user needs and usage patterns.

Moreover, EFS must strike a balance between retaining data and cleaning up obsolete versions to make space for new data. Its cleaner system works by reading the *inode* logs to reclaim storage for unimportant versions, with the added complexity of making sure that important versions are never deleted. The file system must intelligently determine which versions are *landmarks* that should be kept and which ones can be safely removed. This retention strategy aims to minimise the risk of losing important data while also ensuring that the file system does not become burdened with excessive storage usage.

5.3 Delta Encoding

Delta encoding is a technique designed to reduce redundancy by representing data as differences between successive versions or components of information rather than storing entire copies [176, 177]. This approach leverages the observation that many datasets, such as software updates, versioned documents, or multimedia streams that are often similar to prior states, with only localised changes occurring over time. By encoding these incremental differences, delta encoding minimises storage requirements and transmission costs while preserving the ability to reconstruct the original data through sequential application of deltas. Its utility lies in scenarios where efficiency is paramount, such as distributed systems, cloud computing, or real-time communication protocols.

Delta encoding was originally designed as a solution to transmit updates efficiently over bandwidth-limited channels. However, the formalisation of the concept emerged with studies on data compression and information theory such as *Edelta* [178], *Ddelta* [179] and *Gdelta* [180]. Early implementations focused on simple arithmetic differences between numerical values or textual characters, but modern delta encoding now encompasses algorithms capable of handling structured data, binary files, and even complex multimedia formats. Key milestones include the development of *rsync's* rolling checksum algorithm and Git's techniques for version control, which have become foundational to contemporary distributed systems.

Central to delta encoding is the distinction between base data and delta differences. The base represents a reference state from which subsequent versions derive their deltas. To reconstruct a target dataset, one applies the cumulative set of deltas to the base in chronological order. This hierarchical structure necessitates careful selection of bases. An optimal choice maximises redundancy elimination, while suboptimal bases may lead to larger deltas or increased computational overhead during reconstruction. Algorithms such as binary differencing (e.g., *xdelta* [181]) and context-based methods (e.g., bsdiff [182]) exploit structural similarities between the base and target to com-

pute compact differences by identifying common subsequences or aligning data blocks through hashing.

Longest Common Subsequence

The process of computing deltas between two datasets depends on their structure, the types of changes expected, and the differencing algorithm employed. In the case of text files, one common approach is to use line-based or token-based difference algorithms built on the Longest Common Subsequence (LCS) [183]. LCS identifies the largest ordered subsequence shared by both versions, effectively isolating the regions that differ.
For example, suppose we have two versions of a sentence.

Version A: "The quick brown fox jumps over the lazy dog."
Version B: "The quick brown cat leaps over the lazy dog."

The LCS between them is "The quick brown" and "over the lazy dog." What remains outside the subsequence are the mismatched regions: "fox jumps" in Version A versus "cat leaps" in Version B. The delta (or edit script) would then encode these differences with operations such as delete two words ("fox", "jumps") at a specific offset and insert two words ("cat", "leaps"). The precise representation depends on the diff algorithm, but the principle is to record minimal operations that transform A into B rather than retransmitting the whole string.

Binary Files

For binary files, where linearity or structural markers are absent, tools like *bsdiff* use a more granular approach. Consider two executable binaries: an original file (base) and its updated version (target). The *bsdiff* algorithm first divides both into overlapping blocks of bytes. It then computes the differences between corresponding blocks by solving equations to represent changes as arithmetic operations (e.g., target block = base block + delta patch). For

instance, if a function call in the executable is modified from call 0x1234 to call 0xABCD, *bsdiff* would encode this change as an offset adjustment rather than transmitting the entire code segment. The resulting delta file contains patches specifying where these byte-level modifications should be applied.

Rsync

Rsync's [184] rolling checksum technique exemplifies context-based differencing for networked synchronisation. Imagine syncing a large log file between two servers. *Rsync* uses a sliding window to compute a rolling checksum (e.g., Adler-32) over the sender's data, while the receiver simultaneously calculates checksums of its own blocks. When matching checksum values are found at overlapping positions, *rsync* infers that these segments match and need not be transmitted. Only differing regions such as newly added log entries or modified timestamps are sent as deltas with their byte offsets. For example, if a 10MB file has only a 5KB addition at the end, *rsync* would send metadata pinpointing the change location plus the new bytes, avoiding transferring the entire 10MB.

Multimedia

In multimedia encoding, delta computation is used to achieve compression, though the meaning of *delta* depends on the domain. In video codecs such as MPEG-4, deltas are temporal. Instead of storing each frame in full, the codec predicts one frame from another using motion vectors and residual errors [185]. The encoded data is therefore a set of parameters describing how Frame A transforms into Frame B, plus a correction term, a delta not in raw pixel values but in prediction space. This exploits temporal redundancy, while perceptual models (quantisation, chroma subsampling) ensure the discarded detail is visually negligible.

Database

Database transaction logs illustrate deltas in stateful systems. Consider a relational database table tracking user accounts. When a user updates their email address from "alice@example.com" to "alice@newdomain.com," the delta is recorded as an atomic operation:

```
UPDATE users SET email='alice@newdomain.com' WHERE id=123
```

This log entry serves as the delta, allowing the system to roll forward or backward through states. Advanced systems like PostgreSQL use write-ahead logs (WAL) [186] that store page-level changes as binary deltas of modified disk blocks, enabling efficient crash recovery and replication without retransmitting entire tables.

Version Control Systems

Version control systems such as Git optimise storage for repositories with many file revisions. Suppose a project has 100 commits where only the *README.md* file's first line changes incrementally. Instead of storing a complete copy of the file at each commit, Git stores new blobs only when the file changes, while unchanged files simply reference the existing blob.

To further reduce space, Git periodically repacks objects into *packfiles*. In these *packfiles*, Git applies delta compression, one version of a file is chosen as a base, and other similar versions are stored as deltas against that base. For example, one delta might represent the addition of "Version 2.0" to the first line, while another encodes the insertion of a new paragraph.

These deltas can be chained together, where one delta is applied on top of another, but the chain length is bounded (default: 50) to keep object retrieval efficient. Finally, Git applies *zlib* compression to both full objects and deltas, further reducing storage.

Thus, Git's storage savings grow with the similarity of revisions: instead of 100 full copies of *README.md*, only the unique changes and compressed deltas are stored, keeping repository size close to the sum of actual content

changes rather than scaling linearly with commit count.

Discussion

The primary advantage of delta encoding is its efficiency in storage and transmission. For instance, updating a 1GB file with only a 1MB change requires transferring just the delta instead of the entire modified file, drastically reducing bandwidth usage. This benefit is especially critical in distributed systems where nodes synchronise frequently or across low-latency networks. Additionally, delta based approaches enable incremental backups and versioning without requiring full data duplication, which is essential for scalability in cloud storage services and large scale databases. However, these efficiencies come at the cost of increased computational complexity during both encoding (generating deltas) and decoding (applying them to bases), necessitating trade-offs between resource utilisation and performance.

Challenges inherent to delta encoding include determining an appropriate base selection strategy and managing dependency chains among deltas. If successive deltas are applied without reusing intermediate bases, the reconstruction process becomes computationally intensive as each prior delta must be sequentially processed. This *delta chain* problem exacerbates latency in systems requiring real time access to historical states [187]. Furthermore, certain data types such as encrypted files or highly compressible datasets may yield limited redundancy reduction, diminishing the effectiveness of traditional delta techniques. Researchers continue exploring adaptive algorithms that dynamically select bases and adjust encoding parameters based on data characteristics to mitigate these issues.

Delta encoding manifests in two primary approaches: context-based and content-based differencing. Context-based methods, like *rsync*'s rolling checksums, rely on identifying overlapping regions between the base and target by analysing local context windows. These approaches excel at finding commonalities without prior knowledge of the data structure, making them robust for

heterogeneous file types. In contrast, content based techniques, however, leverage domain specific knowledge to exploit semantic relationships. For example, binary diff tools like *bsdiff* analyse instruction semantics in executable files or use patching algorithms optimised for code changes. In multimedia contexts, delta encoding might involve perceptual hashing or feature extraction to capture visual or audio differences while preserving quality thresholds. The choice between these paradigms depends on the data's nature and the required reconstruction fidelity.

Performance optimisation in delta encoding involves balancing the time required for delta generation against the resulting size of the encoded differences. For instance, exhaustive search algorithms may produce highly efficient deltas but incur prohibitive computation costs, whereas heuristic methods prioritise speed at the expense of redundancy reduction. Adaptive techniques dynamically adjust their strategies based on data entropy or change distribution. Sparse modifications might warrant fine grained differencing, while bulk changes could benefit from coarser alignment. Hardware acceleration through specialised processors or GPU-based parallelisation further mitigates latency in real-time applications like live video streaming, where delta encoded frame differences enable smooth transmission over constrained networks.

5.4 Conclusion

This chapter has demonstrated that the temporal dimension of provenance cannot be reduced to symbolic identifiers or abstract placeholders. Through the motivating examples, it has become clear that transformations and executions unfold within evolving contexts, and that ignoring these changes obscures the accountability that provenance ought to provide. Systems such as Progger and PASS highlight the limitations of approaches that either neglect versioning entirely or treat it as little more than a mechanism for maintaining graph structure. By contrast, the scenarios involving compilation and dynamic

linking illustrate how temporal evolution materially affects the behaviour of artefacts, and why recording only identifiers without lineage or verification is insufficient.

In response, the chapter has argued that provenance systems must integrate versioning not as an afterthought but as a substantive element of their design. By treating versions as meaningful records of change, provenance moves from the realm of bookkeeping into a framework that conveys the identity, lineage, and trustworthiness of artefacts across time. The discussion of versioning practices in software engineering and data management has shown how principles such as atomic commits, branching, and distributed collaboration provide lessons for designing provenance systems that can manage complexity while preserving temporal fidelity.

The exploration of semantics, distinguishing snapshots from derivations, further emphasises that versioning is not a monolithic concept. Rather, it entails trade offs between stability and lineage, between storage efficiency and interpretability. Systems such as the Elephant File System illustrate how automatic version retention offers practical benefits, but also how careful policy design is required to balance completeness with efficiency. Similarly, the study of delta encoding demonstrates the necessity of techniques that can reconcile storage constraints with the need to preserve incremental change, thereby supporting both scalability and accountability.

Taken together, these considerations establish that provenance must evolve to capture not only that change has occurred but also what the change is and why it matters. Temporal provenance, enriched by explicit lineage, meaningful versioning, and efficient representation, is essential to ensuring that provenance can meet the demands of modern computing environments, from reproducibility to supply chain integrity. In doing so, it provides the conceptual and technical foundation upon which the next chapter will build to articulate the wider implications for provenance research and practice.

Chapter 6

Embedding Provenance

The previous chapters have progressively established the foundations necessary to examine how provenance ought to be managed in practice, with the previous chapter introducing the first major element of the thesis's argument: temporal data evolution. The chapter broadened the view of provenance from a descriptive add-on to a record that is integral to the identity of data itself. Building on that argument, this chapter turns to the second major element of this thesis: spatial relationship between provenance and data.

The notion of embedding provenance with data requires careful consideration, both in conceptual framing and in practical application. At its simplest, embedding provenance means attaching provenance records directly to the data they describe, rather than storing them externally in a centralised system. This coupling of provenance and data alters the way in which provenance is accessed, disseminated, and preserved. It contrasts with the dominant practice in existing systems, where provenance is stored in relational databases, graph structures, or even remote servers. Those designs separate provenance as a distinct entity, often justifying the separation as a matter of efficiency, scalability, or architectural clarity. However, the separation has the consequence of creating large, entangled pools of provenance that are difficult to partition and disseminate in a manner faithful to ownership and accountability.

To make sense of embedding, one must first consider why previous systems have opted for separation. Provenance has been treated as an additional layer of information that accompanies the primary data but does not necessarily belong to it. The assumption is that provenance is auxiliary metadata, i.e., descriptive rather than constitutive, contextual rather than integral. In this view, provenance is managed in repositories akin to audit logs or transaction histories, separate from the main data repositories. This design choice simplifies certain operations, such as indexing or querying provenance across multiple datasets, but it does so at the cost of weakening the relationship between the provenance and the specific data it describes. It also produces situations where provenance becomes co-mingled, making dissemination back to the rightful data owners difficult without significant reconstruction effort.

A critical question, therefore, is whether we are correct in treating provenance as *metadata*. Definitions across literature frequently characterise provenance as information about "the origins, context, and history of data." This is aligned with common definitions of metadata, which is broadly understood as *data about data*. The conceptual parallel suggests that provenance can, and perhaps should, be managed in the same way as other forms of metadata. To test this, one must examine how metadata is currently handled in practice.

6.1 Metadata

Metadata is "data about data," a set of descriptive elements that provides details, sometimes essential details about a digital resource. Metadata can be embedded directly in files, serving as a layer of description or instruction that helps users and systems interpret the underlying content. For example, in a JPEG image file, the initial portion contains structural metadata that identifies the file format and guides software in interpreting the raw image data. Alongside this, descriptive metadata such as Exchangeable Image File Format (EXIF) information may record the date the photograph was taken, the

camera settings used, or even geographic coordinates. This duality, metadata both as a companion to the content and as a framework for interpreting it, makes it indispensable for managing the vast volumes of information generated in contemporary research, commerce, and daily life.

In the context of an operating system's file system, metadata plays such a critical role that it is often given its own dedicated structure on disk. In Unix-like systems for instance, the *inode* stores key administrative metadata about each file such as ownership, permissions, file size, and pointers to the disk blocks where the file's data resides. Importantly, this *inode*-based structure separates metadata from the actual data, allowing the operating system to efficiently manage files, locating them, enforcing security, and maintaining integrity, without needing to inspect or load their substantive contents. In this way, metadata at the file-system level functions as an invisible but indispensable scaffold for the organisation and governance of digital information.

Metadata can be classified into various categories depending on its function and purpose. Common categories include:

- **Descriptive** metadata provides information about a resource to facilitate its identification, discovery and citation. Examples include the title, author, publication date, abstract and keywords of a document.

- **Structural** metadata describes how the components of a resource are organised and related, supporting navigation and interpretation. Examples include the sequence of pages in a document, the hierarchy of chapters and sections, and relationships between tables in a database.

- **Administrative** metadata supports the management, maintenance and preservation of resources. Examples include file creation and modification dates, access permissions, copyright information, data provenance and retention requirements.

These classifications highlight the multifaceted nature of metadata and its role in different domains.

The utility of metadata extends beyond organisation into active problem-solving domains. Search engines leverage it to rank pages by relevance, while recommendation algorithms use user behavior metadata (like browsing history) to predict preferences. In healthcare, electronic health records embed metadata about patient demographics, medications, and lab results, enabling clinicians to track treatments over time [188]. As already anticipated, metadata is equally integral to data provenance, providing the means to trace the lineage of information back to its source, an essential requirement for verifying claims or conducting audits. For example, climate scientists analysing temperature trends depend on metadata documenting sensor calibration and maintenance to evaluate the reliability of recorded measurements [189]. These applications reveal how deeply embedded metadata is within systems that shape decision making across sectors.

Formal standards also exist for defining and managing metadata. For example, the Dublin Core Metadata Element Set (ISO 15836-1) [190] specifies a standard set of descriptive metadata elements, including title, creator, subject and date, to support consistent resource description and interoperability across systems.

The technical implementation of metadata hinges on standardised formats and protocols that enable interoperability. eXtensible Markup Language (XML) and JavaScript Object Notation (JSON) are widely used to encode metadata in ways machines can parse efficiently. For instance, EXIF standards record photographic parameters, while TEI (Text Encoding Initiative), a popular format used in the digital humanities, scaffolds structural and descriptive metadata in the addition to the document text itself. These frameworks facilitate data exchange but only reduce, rather than eliminate ambiguity. A mislabelled dataset or missing creator attribution can still lead to errors in analysis or credit disputes. The heterogeneity of metadata schemas across disciplines presents further challenges. Combining agricultural census data with climate models, for example, demands not just harmonising conflicting stan-

dards but also reconciling differences in resolution, units and definitions to ensure compatibility and accuracy.

6.1.1 Example: Exchangeable Image File Format

Exchangeable Image File Format (EXIF) is a metadata standard primarily used in digital photography to store information about images captured by cameras. It provides a means to embed metadata directly within image files, offering valuable insights into the conditions and settings under which a photograph was taken. EXIF data is widely supported in JPEG, TIFF, and certain RAW file formats. Its utility extends beyond just photography, playing a key role in forensic analysis, image authentication, and digital asset management.

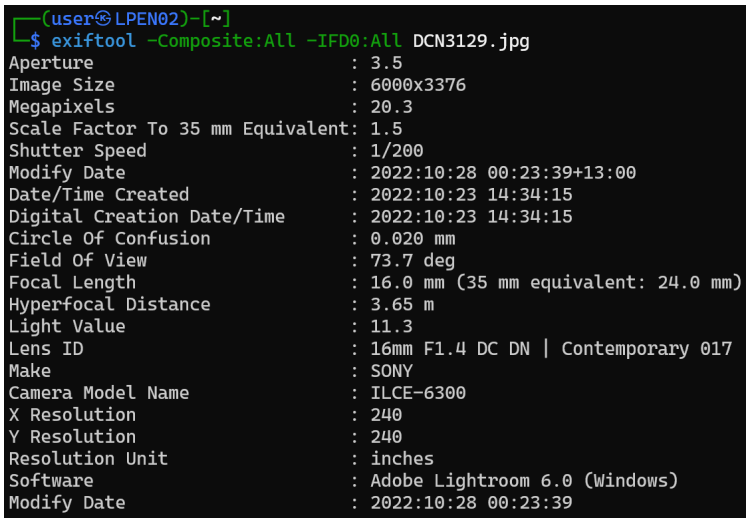


Figure 6.1: Example: Metadata extraction from a digital image using *exiftool*.

EXIF metadata contains a broad range of information, including camera settings such as shutter speed, aperture, ISO sensitivity, focal length, and white balance. Additionally, it often includes timestamp data, geolocation coordinates (if GPS is available and enabled), and even manufacturer-specific information. This structured metadata helps photographers analyse their shooting conditions and optimise future captures. Furthermore, it assists software applications in automatically adjusting image processing workflows based on the embedded data.

Figure 6.1 illustrates the metadata obtained from a photographic image file through the use of *exiftool* [191]. The extracted data in this example include information relating to aperture, shutter speed, focal length, and hyperfocal distance, alongside details of the camera make, model, and lens. The software used to process the image is also recorded, together with resolution values and modification timestamps. Metadata such as these are essential in providing contextual and technical detail that supports reproducibility and analysis of photographic data.

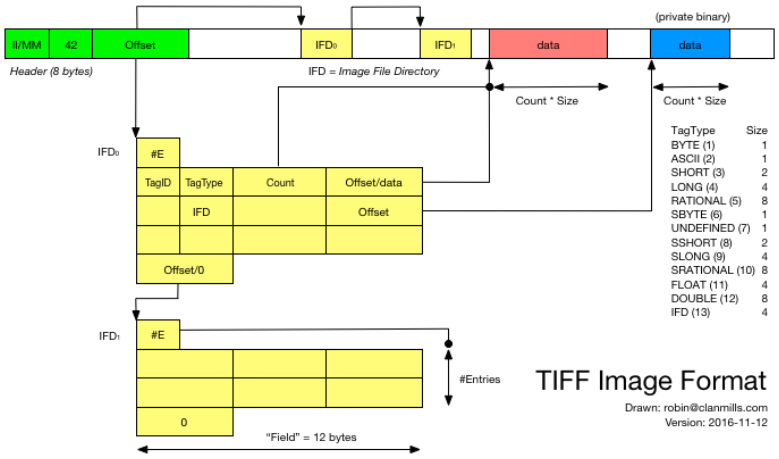


Figure 6.2: TIFF image format [10].

The structure of EXIF metadata follows a specific format defined by the Japan Electronics and Information Technology Industries Association (JEITA) standard. EXIF metadata is organised into tagged fields using a structure derived from TIFF, as illustrated in Figure 6.2, specifically through Image File Directories (IFDs). In JPEG files, EXIF metadata is typically stored within an *APP1* application marker segment near the beginning of the file, whereas TIFF files store metadata through their native IFD structures. This organisation enables metadata to be extracted without modifying the underlying image data. EXIF metadata can be parsed using specialised tools and libraries, such as *ExifTool* and *libexif*, which provide programmatic access to metadata fields. *OpenCV* also provides limited metadata-reading functionality, although it is primarily designed for image processing rather than metadata extraction.

One of the most significant advantages of EXIF is its role in image organisation and cataloging. Digital asset management systems and photo editing software rely on EXIF metadata to sort, filter, and search large collections of images. Applications like Adobe Lightroom and Google Photos leverage EXIF data to group images by date, camera model, and even geographical location. This automated categorisation simplifies workflow efficiency for both amateur and professional photographers.

When an image is captured, the associated EXIF metadata is ordinarily embedded directly within the file, ensuring that the descriptive information travels with the image itself. If instead this metadata were to reside exclusively on the camera, detached from the picture, it would introduce a series of significant implications. The separation would create a dependency on the device as the sole repository of contextual data, making access to the metadata contingent upon maintaining a connection to that specific source. This would undermine the self-contained integrity of the image file, as the descriptive information would not follow the photograph once it is transferred or shared.

Such a system would complicate workflows that rely upon the seamless portability of metadata, including archival processes, cataloguing, or forensic analysis, as the metadata would not automatically accompany the image across platforms. It would also expose the data to a heightened risk of loss or corruption, since a failure of the camera or storage mechanism could sever the link irreversibly. Furthermore, reliance on a centralised point of storage within the device would weaken resilience against security breaches or unauthorised access, as control of metadata would be concentrated rather than distributed.

In addition, offline accessibility would be compromised, since the metadata could not be consulted independently of the camera. This would impede the ability to verify, interpret, or repurpose the images in contexts where the originating device is unavailable. Overall, the dislocation of metadata from the image would erode the reliability, portability, and security that currently characterise the embedded model.

This scenario is directly analogous to data provenance. Provenance metadata, like EXIF information, provides essential context that defines the origins, transformations, and integrity of data. If provenance information were decoupled from the data itself, the same issues would arise, i.e., increased fragility of links, heightened risks of loss or corruption, reduced accessibility, and weakened trustworthiness. The EXIF example offers an intuitive illustration of why provenance metadata must be tightly coupled with the data it describes, ensuring that context and content remain inseparable throughout their lifecycle.

6.1.2 Example: Flexible Image Transport System

The Flexible Image Transport System (FITS) [192, 193] is the standard archival data format for astronomy, designed with the explicit goal of being self describing and portable. It emerged in the late 1970s from the recognition that astronomers needed a universal format to exchange images between institutions. From its inception, it was endorsed by the International Astronomical Union, and over time it has become the dominant archival and interchange format in astronomy.

FITS files are structured as a sequence of Header and Data Units (HDUs). The primary HDU contains a header and, optionally, a primary data array. Headers are made of keyword-value-comment records, fixed at 80 characters, echoing the punch card era. This structure ensures long-term readability whereby future researchers can decode the file by relying on the specification, rather than on transient software implementations. Mandatory keywords such as *SIMPLE*, *BITPIX*, *NAXIS*, and *END* provide structural guarantees, while reserved keywords extend descriptive power, including fields for dates, instrument origins, or observation conditions. Beyond the primary HDU, FITS supports extensions, including image extensions, ASCII table extensions, and binary table extensions. Tables may hold complex data structures, including variable-length arrays and, in later revisions, 64-bit integer support. This has allowed FITS to evolve from a simple transport medium into a flexible format

for heterogeneous astronomical data.

FITS is widely used across astronomical archives, observatories, and analysis software. Data from ground-based and space-based telescopes are almost invariably stored and distributed in FITS. Researchers can open these files in general-purpose astronomy tools, and because of the *once FITS, always FITS* principle, old files remain valid in perpetuity.

Yet FITS is not without shortcomings. Its strict block alignment of 2880 bytes reflects the tape era, leading to padding and inefficiencies. Large images or multi-extension files can become unwieldy, and though no limit is imposed on file size, software implementations sometimes cannot handle files beyond certain thresholds. The rigidity of the header format, limited to uppercase ASCII and fixed-length keyword names, constrains expressiveness. Extensions and conventions have attempted to overcome these limits, but fragmentation has followed, with unofficial practices spreading in the community. Moreover, compared with more modern hierarchical formats such as the Hierarchical Data Format version 5 (HDF5), FITS can be less efficient in handling massive, complex datasets, especially when data volumes continue to grow with new observatories.

HDF5 organise data into groups and datasets, allowing multidimensional data and associated metadata to be stored within a single file. Unlike FITS, HDF5 supports chunked storage, compression and efficient partial data access, enabling retrieval or modification of specific portions of massive datasets without processing the entire file. Its flexible structure also facilitates embedding provenance information alongside the corresponding data. However, this flexibility introduces greater complexity and requires shared conventions to ensure long term interoperability and interpretability.

When considered in the light of provenance, FITS embodies goals similar to those in provenance research. Both seek long term interpretability, independence from the original software environment, and robust embedding of metadata with the data itself. FITS shows that binding metadata in the same

file as data ensures preservation of meaning across decades, fulfilling the requirement that an archive be self-describing. Design considerations such as rigid standards, universal endorsement, and backwards compatibility mirror those in provenance where trust and reproducibility depend on stability. The challenges are also aligned: (i) how to balance metadata expressiveness with compactness, (ii) how to ensure that metadata is complete without inflating file size and (iii) how to allow evolution without undermining the guarantee that old files remain valid. FITS demonstrates that embedding metadata with data is a viable long term strategy, though its inefficiencies and constraints highlight the trade-offs that also arise in provenance design.

6.1.3 Example: GeoTIFF

Georeferenced Tagged Image File Format (GeoTIFF)[194] is a specification that builds on TIFF 6.0 by reserving specific tags to encode geographic information. The innovation lies in its introduction of *GeoKeys*, a form of meta tagging that permits a rich set of geospatial metadata to be stored using only a handful of TIFF tags. These include information about coordinate reference systems, projections, datums, ellipsoids, prime meridians, and units. The *GeoKeyDirectoryTag* serves as the central registry of these metadata entries, with auxiliary tags such as *GeoDoubleParamsTag* and *GeoAsciiParamsTag* holding floating point and text values.

Through this mechanism, a GeoTIFF embeds both raster data (the image) and the cartographic model (its spatial meaning) into a single, portable file. When loaded into a Geographic Information System (GIS), the image can be positioned on the earth's surface without requiring auxiliary "world files" or external metadata databases. This simplifies workflows, that is one file carries both content and context.

GeoTIFF is used extensively in remote sensing, environmental monitoring, topographic mapping, and digital elevation modelling. It allows a satellite image, for instance, to be opened in different GIS platforms while preserving

its spatial reference, thereby ensuring interoperability between producers and consumers of spatial data.

A scanned aerial photograph saved as a GeoTIFF might include tiepoints mapping its corners to Universal Transverse Mercator (UTM) coordinates, with a pixel scale that defines ground resolution. A digital elevation model encoded in GeoTIFF would include a vertical coordinate system definition, ensuring that elevation values are expressed in metres above a given datum. In both cases, applications ranging from cartographic visualisation to hydrological modelling can directly exploit the file.



Figure 6.3: Hexadecimal representation of a GeoTIFF file.

Figures 6.3 and 6.4 illustrate the internal structure of a GeoTIFF file containing a hillshade raster derived from a Digital Surface Model (DSM). The file contains a raster of 1086 × 1445 pixels, stored using *LZW* compression, alongside the metadata required to interpret and geographical position the image. The figures demonstrate how this information is represented at the binary level and subsequently interpreted by metadata extraction software.

Figure 6.3 presents the hexadecimal representation of the beginning of the GeoTIFF file, obtained using the *xxd* utility. The first eight bytes constitute the TIFF header. The initial two bytes, *49 49*, correspond to the *ASCII* characters *II*, indicating that the file uses little-endian byte ordering. The following two bytes, *2A 00*, represent the TIFF identifier, with the hexadecimal value *0x002A* corresponding to decimal 42. The subsequent four bytes, *08 00 00 00*, specify the byte offset of the first Image File Directory (IFD), which begins at byte 8. At this location, the value *12 00* indicates that the IFD contains 18 directory entries. Each entry occupies 12 bytes and consists of a



Figure 6.4: ExifTool interpretation of a GeoTIFF file.

tag identifier (2 bytes), data type (2 bytes), value count (4 bytes), and either the value itself or an offset pointing to its location elsewhere in the file (4 bytes). For example, the *ImageWidth* entry, identified by tag 256 (0x0100), specifies a *SHORT* data type, a count of one, and a value of 1086 pixels. This illustrates how TIFF encodes image properties within a structured directory rather than storing them as descriptive text.

Figure 6.4 shows the same file interpreted using *ExifTool*, which extracts and presents the TIFF directory entries in a human readable form. In addition to the image dimensions, the IFD specifies properties such as the number of samples per pixel, compression method, and tile configuration. In this example, the raster is divided into 30 tiles, each nominally measuring 256×256 pixels. The compressed pixel data is not stored within the IFD itself; instead, the *TileOffsets* tag (324) contains the byte positions at which the individual tiles begin, while the *TileByteCounts* tag (325) records their compressed sizes.

For instance, the first tile begins at byte offset 664 and occupies 80,722 bytes. These entries allow software to locate, decompress, and reconstruct the raster image from the binary data stored elsewhere in the file.

Although the figures focus on the TIFF header and raster organisation, the same IFD also contains the GeoTIFF-specific tags responsible for georeferencing. In this file, the *ModelPixelScaleTag* (33550) specifies a spatial resolution of one metre, while the *ModelTiepointTag* (33922) establishes the correspondence between raster coordinates and projected map coordinates. The *GeoKeyDirectoryTag* (34735) contains seven *GeoKeys* describing the spatial reference, including the NZGD2000 / New Zealand Transverse Mercator 2000 coordinate reference system (EPSG:2193). Together, these entries demonstrate how GeoTIFF extends the TIFF directory mechanism to associate raster data with geographic information. Consequently, GIS software can interpret both the image content and its spatial position directly from the same file, without requiring an external world file.

GeoTIFF's reliance on structured, extensible keys echoes provenance's need for schemas that can evolve yet remain interpretable. The challenges it faces, which are (i) opaque coding systems, (ii) risks of private divergence, and (iii) balancing compactness with expressivity, mirror those in provenance. For provenance, the question is how to represent complex workflows and dependencies; for GeoTIFF, it is how to represent complex geodetic systems. Both wrestle with the challenge between standardisation and extensibility.

6.1.4 Example: Web Metadata

Web metadata serves to bridge the gap between raw content and its usability. Without metadata, web pages would exist in isolation, with little capacity for search engines or other digital tools to understand their relevance. Metadata allows web crawlers to categorise and rank content, ensuring that users find the most pertinent information in response to their queries. This function is critical where the sheer volume of digital content continues to expand exponentially.

Web metadata takes many forms, each serving distinct purposes. Descriptive metadata includes elements such as title tags, meta descriptions, and keywords, all of which help summarise the content of a webpage. Structural metadata, on the other hand, defines the relationships between different pieces of content, enabling functionalities such as linking, navigation, and hierarchical categorisation. For example, hyperlinks connect related web pages, breadcrumb navigation indicates a page's position within a website, and hierarchical structures organise content into categories and subcategories, such as products grouped by type in an online shop. Additionally, administrative metadata includes information on file formats, access rights, and content authorship, playing a crucial role in digital rights management and content governance.

One of the most widely adopted frameworks for web metadata is the *schema.org* initiative. Developed collaboratively by major search engine providers, *schema.org* offers a standardised vocabulary for structured data markup, enabling webmasters to embed metadata within HTML using formats such as *JSON-LD*, *RDFa*, or *microdata*. This structured approach allows search engines to better interpret web content and, in some cases, display enhanced search features such as product details, event information, or review ratings. While the presence of structured metadata does not guarantee rich snippets or higher search rankings, it can improve visibility indirectly by making content more comprehensible to search engines.

The Semantic Web extends the role of metadata by aiming to transform the web into a more intelligent, interconnected space where data is machine readable and interoperable. Standards such as the Resource Description Framework (RDF) and Web Ontology Language (OWL) enable the creation of explicit relationships between datasets, supporting reasoning and more advanced forms of data integration and retrieval. In principle, these technologies allow web applications to deliver more personalised, context aware, and semantically rich user experiences, though large scale deployment of full Semantic Web technologies remains limited compared to lighter approaches like *schema.org*.

Despite its benefits, the implementation of web metadata presents challenges. Consistency and adherence to standards are essential, yet many developers neglect best practices, leading to incomplete or invalid metadata. Moreover, metadata has historically been misused for manipulative practices such as keyword stuffing or deceptive markup. Search engines have responded by refining their algorithms to assess the credibility, accuracy, and relevance of structured data, penalizing or ignoring manipulative uses.

Looking forward, embedding metadata is essential for maintaining the principles of Findability, Accessibility, Interoperability, and Reusability (FAIR) outside of centralised repositories. When metadata is tightly coupled with the resource it describes, the data remains self-contained; it carries the necessary context for discovery and reuse regardless of where it is stored or how it is accessed. This ensures that the data remains FAIR even when separated from its original infrastructure, a critical requirement in domains such as open science and research data where assets frequently move between systems.

6.2 The Case for Embedding Provenance

The examples in the previous section illustrate that established metadata standards and data provenance share similar goals, objectives, and challenges. Understandably, it make sense to treat provenance as a specialised form of metadata, with the assumed implication, as pursued by prior research, that this is handled externally, as it is *data about data*. However, embedding provenance within the data itself also aligns with the principles of metadata and is one worth exploring.

The foundational advantage of embedded provenance is portability. When provenance is stored separately, transferring data often requires additional steps to ensure provenance information is included. This reliance on external references creates opportunities for provenance data to become lost or outdated. Organisations transfers data across platforms, cloud environments,

or institutional boundaries, but external provenance systems may not interoperate seamlessly in these transitions. Embedded provenance ensures that lineage and context remain attached regardless of where the data moves. This is particularly beneficial for collaborative projects spanning multiple institutions or sectors, where maintaining a unified view of data origins is essential to reproducibility and accountability. For example, sharing sensor data between universities and industry partners becomes less error prone when each dataset carries its own history, eliminating the need for manual reconciliation or external metadata synchronisation.

Storing provenance externally introduces risks of desynchronisation between datasets and their metadata. For instance, updates to a dataset may not be reflected in an external provenance store due to delays or procedural oversights. Embedded provenance inherently avoids this discrepancy. Any modification to the data triggers corresponding updates to its embedded provenance record. This ensures that the two remain synchronised by design, preserving the integrity of the historical narrative. Such consistency is particularly vital for applications requiring auditable trails, such as scientific research, regulatory compliance, or financial transactions, where even minor inconsistencies can lead to cascading errors.

Embedded provenance offers an opportunity to utilise cryptographic safeguards against tampering. By integrating checksums, hashes, or digital signatures into the data object itself, any unauthorised alteration of either the dataset or its provenance becomes immediately detectable. For example, blockchain inspired techniques [195] can be adapted to embed immutable timestamps and authorship records directly within files, creating an auditable chain that cannot be retroactively altered without leaving traces. This approach contrasts sharply with external systems, where provenance data might be compromised through breaches of centralised repositories or manual errors during updates. When paired with cryptographic primitives, embedded provenance becomes a self-contained integrity mechanism, fostering trust in

the authenticity and reliability of datasets.

Users interacting with a dataset would benefit from immediate access to its provenance without navigating fragmented systems. Embedded provenance eliminates the cognitive burden of piecing together lineage information from disparate sources. A researcher analysing climate data can instantly view its collection methods, calibration parameters, and prior analyses all embedded within the file, without querying a separate database or relying on outdated documentation.

The computational efficiency of embedded provenance arises from its monolithic structure. Queries for both data and its lineage can be executed in a single operation, avoiding the overhead of cross-system joins or API calls. For large-scale datasets stored in distributed systems like Hadoop or cloud repositories, this could reduce latency and optimises resource usage.

Beyond technical advantages, embedded provenance serves as a tool for ethical and legal compliance. Regulations such as GDPR mandate transparency about how personal data is collected and processed. By embedding this information within datasets themselves such as consent records or anonymisation techniques organisations can demonstrate compliance more effectively during audits without relying on external systems that might be incomplete or outdated. Similarly, in healthcare research, patient derived data with embedded provenance reduces the risk of misattribution or misuse by ensuring lineage details accompany the dataset wherever it travels.

6.3 Techniques for Embedding Provenance

Building on these advantages, the next step is to examine how provenance can be practically embedded within data. In this section, we explore potential techniques that can realise this possibility, each offering different trade-offs in terms of portability, interoperability, and ease of use. Exploring these approaches helps illuminate not only the technical possibilities but also the considerations

required to balance efficiency, scalability, and long-term accessibility.

6.3.1 Custom File Structure

The most straightforward and naïve approach to embedding data provenance directly within the data is to define a custom file structure. This involves designing a novel format where provenance metadata is interwoven with the primary data, ensuring that both are inseparable. Unlike traditional systems where provenance is stored in external databases or linked files, this technique integrates lineage, transformations, and contextual metadata directly into the file, reducing the risk of loss or inconsistency.

The design of such a file structure requires careful consideration of how to structure and encode provenance information without excessively bloating the file size or introducing inefficiencies in data retrieval. A hierarchical format could be employed, where data blocks are interspersed with their corresponding provenance records. Alternatively, a segmented approach could be used, where a dedicated provenance section exists within the file, indexed in a manner that facilitates rapid access and updates. To maintain usability, the file format must support efficient querying, ensuring that provenance details can be extracted without needing to read the entire file. A schema-driven approach, potentially using self-describing formats such as JSON, XML, or even binary structures with embedded metadata, could allow for flexibility and extensibility.

However, this naïve approach introduces several limitations. One major concern is compatibility. Existing data processing tools are not designed to handle custom file structures, meaning extensive modifications or entirely new software would be required to read and write such files. This creates a barrier to adoption, as users would need to transition away from widely accepted formats. Additionally, embedding provenance within the data can lead to significant file overhead, especially when dealing with frequent updates or high-granularity provenance tracking. Without efficient compression or deduplication techniques, files could grow exponentially in size, negatively impacting

storage and transmission costs.

Another critical limitation is the complexity of updates. If a portion of the data needs to be modified, ensuring the consistency of embedded provenance while maintaining efficiency becomes a challenge. Simple edits may necessitate restructuring the entire file, leading to performance bottlenecks. Furthermore, access control becomes more complicated when provenance and data are tightly coupled. In scenarios where different users require different levels of access, separating sensitive provenance details from general data consumption becomes problematic.

Despite these drawbacks, the custom file structure approach serves as an important starting point in exploring embedded provenance. It highlights key challenges in ensuring provenance integrity, accessibility, and efficiency while demonstrating the potential benefits of tighter data-provenance coupling. Addressing these limitations may involve hybrid strategies, such as leveraging standardised file formats with embedded provenance extensions or adopting more sophisticated indexing mechanisms. Ultimately, while this naïve method may not be a practical long term solution, it offers valuable insights that guide the development of more refined provenance embedding techniques.

6.3.2 Extended File Attributes

Extended file attributes (*xattr*) provide a mechanism for associating additional metadata with files and directories beyond the standard attributes such as permissions, ownership, and timestamps. This feature is available in modern file systems such as ext4, XFS or ZFS, including those used in Linux, BSD, and macOS. Extended attributes are stored as key-value pairs and allow users or system processes to attach additional descriptive or functional metadata to a file. Unlike conventional file metadata, which is limited in scope and structured according to predefined system parameters, extended attributes enable more flexible data annotation and storage.

Extended attributes play a significant role in applications that require en-

hanced metadata management, including security frameworks, system auditing, and data provenance tracking. By providing a means to store extra information without modifying the file content, *xattr* facilitates the embedding of contextual data at the file system level. This capability is particularly relevant for maintaining provenance metadata.

Extended attributes are categorised into different namespaces, each with specific access restrictions and use cases. The primary namespaces include:

- User Namespace (*user*): This namespace is available for general-purpose metadata storage. Users can set attributes related to file descriptions, categorisation, or application-specific data. However, access to these attributes is governed by file permissions, meaning that read and write permissions are required to retrieve or modify attribute values.

- Trusted Namespace (*trusted*): Attributes in this namespace are accessible only to processes with administrative privileges (*CAP_SYS_ADMIN*). These attributes are useful for system-level metadata that must be protected from regular user modifications.

- Security Namespace (*security*): Used by security frameworks such as Security-Enhanced Linux (SELinux), this namespace stores security-related attributes that control access policies and authentication mechanisms.

- System Namespace (*system*): This namespace is used by the operating system to store structured metadata, such as access control lists (ACLs), that help manage file permissions and system policies.

Extended attributes offer a compelling approach for embedding provenance metadata directly within the file system. By attaching provenance records as *xattr*, the metadata remains closely associated with the data file without requiring external databases or additional log files. To implement this, provenance can be set using system calls such as *setxattr()* in Linux, which allows writing extended attributes to files. For example, a provenance-aware application could invoke *setxattr(filepath, "user.provenance", provenance_data, size,*

0); where *provenance_data* contains structured information such as the origin, transformations, and ownership history of the file. This data could be stored in formats such as JSON or binary representations optimised for efficient parsing. Similarly, retrieving provenance would be achieved using *getxattr()*, ensuring that any process with appropriate permissions can verify the integrity and lineage of the file without requiring external lookups.

One major limitation of *xattr* is that file systems impose limits on the size of extended attributes, which vary by implementation [196]. For example, in ext4, a single *xattr* cannot exceed 4 KB, making it impractical for storing complex, high-fidelity provenance records. While one could distribute provenance data across multiple attributes or compress it, this complicates retrieval and management. Furthermore, some applications and backup tools do not account for extended attributes, meaning provenance could be unintentionally stripped during data migrations or archival processes.

Security of provenance is also not guaranteed. While *xattrs* can be protected using file system permissions, they are not inherently tamper-proof. An adversary with sufficient privileges could modify or erase provenance data without affecting the file contents, which undermines trust in the provenance record. One possible mitigation is cryptographic signing, where provenance entries include digital signatures that can be verified upon retrieval. However, this increases complexity and requires key management infrastructure.

Finally, not all file systems supports *xattr* uniformly, which can create inconsistencies when working across different platforms or storage solutions. Some file systems, such as *ext4* [197], *XFS* [198], and *Btrfs*, provide robust support for *xattr*, allowing users and applications to attach metadata to files beyond standard attributes like permissions and timestamps. However, others, like *FAT32* and *exFAT*, either lack support for extended attributes entirely or offer only limited functionality. This discrepancy means that transferring files between file systems with different *xattr* capabilities may lead to loss of metadata, affecting applications that rely on these attributes for security,

access control, or indexing purposes.

Operating systems also handle *xattr* differently, which can complicate file transfers, backups, and migrations. For example, macOS makes extensive use of extended attributes, particularly for features like Finder tags, quarantine flags for downloaded files, and application metadata. However, if a macOS user transfers a file to a USB drive formatted with *FAT32*, these attributes will not be preserved, potentially leading to unexpected behavior when the file is moved back to macOS. Similarly, Windows NTFS supports alternate data streams (ADS), which function similarly to *xattr*, but these streams are lost when files are copied to file systems that do not support them, such as Linux *ext4* or cloud storage solutions that strip out metadata.

6.3.3 Alternate Data Streams

```
PS C:\Users\m365-\Downloads> Get-Item .\paper.pdf -Stream *

PSPath        : Microsoft.PowerShell.Core\FileSystem::C:\Users\m365-\Downloads\paper.pdf::$DATA
PSParentPath  : Microsoft.PowerShell.Core\FileSystem::C:\Users\m365-\Downloads
PSChildName   : paper.pdf::$DATA
PSDrive       : C
PSProvider    : Microsoft.PowerShell.Core\FileSystem
PSIsContainer : False
FileName      : C:\Users\m365-\Downloads\paper.pdf
Stream        : :$DATA
Length        : 1308973

PSPath        : Microsoft.PowerShell.Core\FileSystem::C:\Users\m365-\Downloads\paper.pdf:Zone.Identifier
PSParentPath  : Microsoft.PowerShell.Core\FileSystem::C:\Users\m365-\Downloads
PSChildName   : paper.pdf:Zone.Identifier
PSDrive       : C
PSProvider    : Microsoft.PowerShell.Core\FileSystem
PSIsContainer : False
FileName      : C:\Users\m365-\Downloads\paper.pdf
Stream        : Zone.Identifier
Length        : 900

PS C:\Users\m365-\Downloads> Get-Item .\paper.pdf | Get-Content -Stream Zone.Identifier
[ZoneTransfer]
ZoneId=3
ReferrerUrl=https://www.researchgate.net/profile/Thomas-Pasquier/publication/278413831_CamFlow_Managed_Data-sharing_for_
Cloud_Services/links/55f3d3ea08ae1d980394af9f/CamFlow-Managed-Data-sharing-for-Cloud-Services.pdf?origin=publication_det
ail&_tp=eyJjb250ZXh0Ijp7ImZpcnN0UGFnZSI6InB1YmxpY2F0aW9uIiwicGFnZSI6InB1YmxpY2F0aW9uRG93bmxvYWQiLCJwcmV2aW91c1BhZ2UiOiJw
dWJsaWNhdGlvbiJ9fQ&__cf_chl_tk=.TUCvOAuJbuVOPkPLDdKT_EyK3PjDSOJcPVErlusVFs-1758977227-1.0.1.1-U_Nj7QtHyNAf452GwGnmpYYadu
inc1yLamTVLIaVoYE
HostUrl=https://www.researchgate.net/profile/Thomas-Pasquier/publication/278413831_CamFlow_Managed_Data-sharing_for_Clou
d_Services/links/55f3d3ea08ae1d980394af9f/CamFlow-Managed-Data-sharing-for-Cloud-Services.pdf?origin=publication_detail&
_tp=eyJjb250ZXh0Ijp7ImZpcnN0UGFnZSI6InB1YmxpY2F0aW9uIiwicGFnZSI6InB1YmxpY2F0aW9uRG93bmxvYWQiLCJwcmV2aW91c1BhZ2UiOiJwdWJs
aWNhdGlvbiJ9fQ
```

Figure 6.5: Examination of ADS metadata and Zone.Identifier.

Alternate Data Streams (ADS) provide a compelling mechanism for embedding provenance directly within files, particularly on NTFS file systems. By leveraging ADS, provenance metadata can be associated with a file without altering its primary content. This approach ensures that provenance remains tightly coupled with the data while maintaining backward compatibility with

applications that do not recognise ADS. In such design, each file would store its provenance metadata in a named alternate stream, separate from the main data stream but still inherently bound to the file itself. The naming convention of these streams can be standardised, such as *provenance* or a structured namespace like *prov.version1*, allowing for systematic retrieval and modification.

Setting provenance within ADS can be accomplished using system utilities or API calls. For example, on Windows, the *SetFileInformationByHandle* function or PowerShell's *Set-Content* with the *::$DATA* stream suffix can be used to embed provenance metadata within an alternate stream. A dedicated application or service can intercept file creation and modification events, ensuring provenance is captured at the moment of data generation or transformation. The system could implement cryptographic signing within the stream to prevent unauthorised modifications, ensuring provenance integrity. Additionally, versioning can be achieved by maintaining a chronological sequence of ADS entries, each representing a snapshot of provenance history, further supporting accountability and reproducibility.

Retrieving provenance from ADS is straightforward but requires explicit handling, as most applications do not automatically process alternate streams. The *Get-Content* command with the appropriate stream name allow for extraction. If the file is copied within an *NTFS* environment, the provenance remains intact; however, when transferred to a non-*NTFS* file system, such as *FAT32* or *exFAT*, the alternate streams are lost, leading to provenance detachment. This poses a significant limitation, as data mobility across diverse storage mediums is a fundamental requirement in many systems.

Figure 6.5 demonstrates the use of PowerShell commands to inspect alternate data streams (ADS) associated with a file named *paper.pdf* located in the Downloads directory. The *Get-Item -Stream ** command reveals two streams: (i) the primary *:$DATA* stream, which holds the file content, and the *Zone.Identifier* stream, which contains metadata added by Windows when

files are downloaded from the Internet. This metadata records the origin and security context of the file. By invoking *Get-Content -Stream Zone.Identifier*, the contents of this metadata stream are displayed, showing the *[ZoneTransfer]* section with *ZoneId=3* (indicating the Internet zone) and detailed referrer and host URLs pointing to the source of the document, in this case, a ResearchGate publication link.

Despite its advantages, ADS-based provenance embedding has notable limitations. The main concern being the lack of cross-platform support [199]. While *NTFS* supports ADS, many other file systems, including Linux's *ext4* and macOS's *APFS*, do not recognise or preserve these streams. Even within Windows, many applications disregard ADS [200], making provenance effectively invisible unless explicitly accessed. Another issue is security; since ADS metadata is not immediately visible in directory listings, it can be exploited by malware to conceal information, raising concerns about its use in security-sensitive environments. Moreover, traditional backup solutions often ignore ADS, leading to silent data loss when files are restored [201].

In contrast to *xattrs*, ADS offers a more flexible and scalable approach to embedding provenance, allowing for extensive metadata storage without affecting file size as reported by traditional file management utilities. However, the platform specific nature of ADS makes it less suitable for scenarios requiring cross-platform compatibility and long-term preservation. A hybrid approach, where ADS is used for provenance storage within NTFS environments and complemented by *xattrs* or embedded provenance files for broader compatibility, could mitigate these limitations while preserving the benefits of direct data-provenance coupling.

6.3.4 i-node Modification

Incorporating provenance directly into the *inode* structure of a filesystem offers a novel approach to ensuring provenance data remains inseparable from the associated data. By modifying the *inode*, which primarily stores metadata

such as file size, access permissions, and timestamps, additional fields can be introduced to capture and store provenance related information. These fields could include details about data origin, transformations, lineage, and responsible entities. The extended *inode* would serve as a self-contained unit where provenance is embedded alongside conventional metadata, eliminating the need for an external provenance management system.

To implement this, a portion of the *inode* structure must be repurposed or expanded to accommodate provenance records. One approach is to utilise unused or reserved fields within the *inode*, repurposing them to hold a compact representation of provenance data. However, given the typically limited space in an *inode*, a more scalable solution involves linking the *inode* to dedicated provenance blocks stored within the filesystem. These blocks, referenced through additional pointers in the *inode*, could store provenance metadata in a structured format, such as a serialised representation of provenance graphs or cryptographic hashes of historical changes.

Setting provenance within this extended *inode* structure occurs at various stages of file system operations. Upon file creation, the *inode* is initialised with its first provenance entry, documenting its origin, creator, and initial context. Subsequent modifications, such as edits, copies, or transformations, trigger updates to the linked provenance blocks, ensuring that each modification is appended in a tamper-resistant manner. Provenance integrity can be further reinforced by employing cryptographic techniques such as hashing and Merkle trees, ensuring that any tampering attempts are detectable. A versioning mechanism can be introduced to enable efficient retrieval of historical provenance states, allowing users to reconstruct the complete lineage of the data.

Retrieving provenance data from an *inode*-enhanced filesystem involves traversing the *inode*'s pointers to its associated provenance blocks. This retrieval process can be efficiently integrated into filesystem operations by extending system calls such as *stat()* to expose provenance related attributes

alongside traditional metadata. Additionally, indexing mechanisms can be introduced at the filesystem level to accelerate provenance queries, ensuring that users can efficiently access and verify data history without significant performance overhead.

Despite its advantages, embedding provenance within the *inode* structure presents several limitations. The most immediate concern is storage overhead, as *inode*s are traditionally designed to be minimalistic [202], and embedding provenance information increases their size. This overhead could become significant, especially for files with extensive modification histories, necessitating efficient storage strategies such as delta encoding or compression. Another challenge is performance degradation, as frequent provenance updates may introduce additional disk I/O operations, potentially impacting write performance. Moreover, ensuring compatibility with existing filesystem architectures requires careful design choices, as modifying the *inode* structure necessitates changes to filesystem drivers, tools, and utilities [202, 203, 204]. Additionally, provenance integrity enforcement must be carefully designed to prevent unauthorised modification of provenance records while still allowing legitimate updates.

Overall, embedding provenance within the *inode* structure represents a promising but technically intricate solution. It offers strong guarantees of provenance and data coupling but requires careful trade-offs between efficiency, scalability, and compatibility with existing systems. Addressing these challenges through optimised storage techniques and efficient update mechanisms will be key to realising a practical implementation.

6.4 Conclusion

This chapter has argued that embedding provenance within data represents a fundamental shift in how we conceive of the relationship between context and content. Where existing systems typically externalise provenance, treating it

as supplementary metadata stored in separate repositories, embedding insists upon a model where provenance is inseparable from the data itself. The examples of established metadata practices, from EXIF in photography to FITS in astronomy and GeoTIFF in geospatial science, demonstrate that embedding contextual information directly within files not only safeguards portability and integrity, but also enhances trust, accessibility, and longevity. These domains illustrate that metadata embedded at the point of creation ensures resilience against loss, detachment, or corruption, challenges that equally beset provenance when it is managed externally.

The case for embedding provenance rests upon its ability to deliver both practical and conceptual benefits. Practically, it strengthens synchronisation, portability, and interoperability while reducing dependence on fragile external systems. Conceptually, it challenges the notion of provenance as a mere adjunct to data, instead framing it as integral to the identity of the dataset itself. To separate data from provenance is to weaken both, producing inconsistencies and inefficiencies that compromise reproducibility and accountability. Embedding, by contrast, offers a means of coupling the two in a manner that better reflects their interdependence.

Nevertheless, embedding is not without cost. The technical strategies explored, from custom file structures, extended file attributes, and alternate data streams, to modifications of *inodes*, reveal a landscape of trade-offs. Portability competes with platform specificity, efficiency with expressiveness, and extensibility with compatibility. These challenges do not undermine the principle of embedding, but they highlight the importance of carefully balancing practical design constraints with the normative demand for robust provenance.

In situating provenance as a form of metadata, this chapter has shown that embedding is neither unprecedented nor infeasible. Rather, it is a logical extension of practices already deeply woven into digital infrastructures. What remains, however, is to explore how these approaches can be translated into a working system. The next chapter therefore turns to the design and im-

plementation of a prototype, demonstrating how the principles of embedded provenance can be operationalised in practice and tested against the challenges identified here.

Chapter 7

Design & Implementation

This chapter presents the design and implementation of the prototype developed for this research. The prototype is not presented as a full system, nor as a substitute for any existing provenance infrastructure. Its role is to validate the concepts introduced in the thesis by showing that the proposed approach is workable in practice. In particular, it demonstrates the feasibility of (i) integrating temporal awareness and (ii) embedding data provenance directly into files. The mechanisms implemented are deliberately system agnostic so they can be incorporated into a broad range of existing provenance frameworks without requiring significant architectural changes. The prototype therefore serves as a proof of concept whose value lies in confirming that the design principles set out in this thesis can be retrofitted into current systems to deliver the benefits described.

The chapter begins by introducing the requirements and design principles guiding the prototype, then outlines the overall architecture, including components for delta extraction, provenance modelling, embedding, and exploration. The chapter describes how changes are captured as deltas, how provenance is stored directly within files using a block-based format, and how historical states can be reconstructed and validated. It concludes by framing the prototype as a proof of concept that shows feasibility of the system.

7.1 Requirements and Design Goals

This prototype is guided by three sets of requirements. Together they describe what the system must achieve overall, what it must provide specifically for delta capture, and what is required to embed provenance directly within files. These requirements form the basis for the architectural and implementation choices described later in the chapter.

7.1.1 General Requirements for Provenance System

1. **GR1: Automated provenance tracking**. The system must possess the capability to automatically capture provenance information without intervention. Provenance capture should occur as an integral part of routine system operations.

2. **GR2: Accuracy of lineage representation.** The system must capture the causal relationships between artefacts and the processes that transform them. Lineage information should reflect both high level workflow structure and fine grained operations where available. For example, when a file is modified, its new version is represented as an artefact entity linked to the previous version through a *wasDerivedFrom* relationship, with the responsible activity and associated delta recording how the transformation occurred.

3. **GR3: Completeness and consistency.** Provenance capture must occur automatically, and without relying on user discretion, to avoid omissions or inconsistent recording. Any provenance record should be internally coherent, with no ambiguous or partially linked nodes.

4. **GR4: Interoperability with existing standards.** While the prototype is exploratory, it must be compatible in spirit with established provenance models such as W3C PROV. This enables mapping between representations and future integration with external tools.

5. **GR5: Scalability of storage and querying.** As provenance graphs grow quickly in real systems, the prototype must store records efficiently and allow responsive queries across large histories, including searches over versions, artefacts, and transformations.

6. **GR6: Minimal intrusion into user workflows.** In addition to the provenance capture not requiring any special actions from the user (GR1), the added instrumentation should be as expedient as possible and must not alter expected behaviour.

7. **GR7: Security and integrity guarantees.** The system must ensure that provenance cannot be silently altered. Where full cryptographic mechanisms are not implemented, the design should at least allow them to be added without architectural changes.

8. **GR8: User Configurable Provenance Scope**. Recognising that different users and use cases impose varying requirements on the granularity and type of provenance information required, the system must provide mechanisms for users to define the scope of provenance capture. For instance, if a user deems network level provenance to be irrelevant for their application domain, they should be able to explicitly disable its collection.

9. **GR9: Support for Custom Annotation**. The system must allow users to annotate provenance data with custom metadata, providing additional context and descriptive information about file changes, interactions, or events. These annotations should be flexible in nature, enabling users to include relevant details such as user comments, reasons for changes, or tags for classification.

7.1.2 Requirements for Temporal Awareness

The prototype treats changes between versions over time as explicit provenance artefacts. The design requirement in this area need to ensure that these deltas are recorded, stored, and usable as part of the provenance graph. In more detail:

1. **TA1: Detection of content level changes.** The system must identify modifications at the level of content rather than relying solely on timestamp or file replacement heuristics. Detection should be robust to small edits and capable of distinguishing meaningful differences from incidental ones.

2. **TA2: Compact and queryable delta representation.** Deltas must be stored in an efficient form to minimise storage overhead. At the same time, the representation must remain amenable to provenance queries. For example, compression techniques are acceptable so long as they remain query compatible.

3. **TA3: Linking deltas into the provenance graph.** Each delta must be a first-class node or edge within the provenance model, linked to both its predecessor and successor states as well as to the process that generated it. This allows change events to be queried, visualised, and reasoned about alongside other provenance entities.

4. **TA4: Replay and reconstruction of states.** The system must be able to reconstruct any historical state by replaying deltas. This supports verification, auditing, and reproducibility use cases, and ensures that deltas are not merely descriptive but operational.

5. **TA5: Temporal coherence.** Versioning must preserve a consistent temporal ordering, ensuring that reconstruction and dependency analysis remain well defined even in concurrent environments.

7.1.3 Requirements for Embedding Provenance

Embedding provenance metadata within files allows artefacts to carry their own histories. This strengthens portability and resilience but introduces constraints that the prototype must meet. More specifically:

1. **EP1: Non-disruptive readability and graceful degradation.** Embedding provenance must not break the file's usability by standard applications. At a minimum, software should be able to read the primary content even if it ignores the embedded provenance metadata. Where full compatibility cannot be achieved, degradation should be predictable and safe.

2. **EP2: Embedding structured metadata together with deltas.** The embedded content must include both high level provenance descriptors (e.g., process identifiers, authorship, timestamps) and fine grained deltas where appropriate. Structure is essential for later parsing, validation, and querying.

3. **EP3: Verification and tamper detection.** The embedded provenance should support mechanisms that allow recipients to check whether the provenance is intact.

4. **EP4: Performance within acceptable bounds.** Embedding operations must not introduce prohibitive overheads in file size or processing time. The system should remain responsive for typical file sizes encountered in target use cases.

5. **EP5: Extensibility for future provenance elements.** The encoding mechanism should accommodate future extensions, such as additional metadata fields, cryptographic signatures, digital object identifiers, or policy annotations, without requiring redesign of the file layout.

6. **EP6: Separation of concerns between content and metadata.** Although metadata is embedded, the file format must maintain a clear

conceptual division between the semantic content and the provenance payload, ensuring that tools can parse or ignore the metadata as needed.

7.2 Design Principles

The development of the prototype was guided by a set of design principles that evolved through an iterative process of conceptual reasoning and empirical exploration. Rather than being conceived in a purely top-down or bottom-up manner, the design and the prototype implementation informed one another. Early theoretical considerations about temporal awareness and provenance integration shaped the system's initial architecture, while successive cycles of implementation and evaluation refined these principles in response to practical challenges and emerging insights.

This process ensured that the resulting design framework was both theoretically grounded and pragmatically viable. The principles aim to balance conceptual clarity with technical feasibility, offering a foundation that is general enough to inform future provenance systems yet specific enough to address the implementation realities of the prototype.

7.2.1 Principle 1: Everything is File

The concept of *everything is a file* is a foundational abstraction in Unix and Unix like operating systems [205]. It expresses a unifying design principle: disparate system resources such as regular files, directories, hardware devices, inter-process communication (IPC) channels, and more, are all treated uniformly as files, or more precisely, as file descriptors with common interfaces. While this abstraction simplifies interaction with diverse resources, it is also a deliberate stance on how systems should expose complexity.

Directories, although abstracted as hierarchical containers, are in fact special files storing mappings. Devices appear under */dev* and are manipulated using file semantics. IPC mechanisms like named pipes, Unix domain sockets,

and even certain network sockets fit into the same conceptual model. Procfs (*/proc*) and sysfs (*/sys*) extend this to introspection and configuration, exposing kernel state and runtime parameters as virtual files, allowing processes to monitor and modify system behaviour using ordinary file operations.

We adopt and extend the Unix design principle that everything is a file as a foundational axiom in our system architecture. By treating all entities such as data, processes, interactions, configurations, and even ephemeral system events, as files, we enforce a unifying interface that simplifies reasoning, enables composition, and crucially, allows for universal provenance tracking.

In our framework, the axiom extends naturally. If everything is a file, and if every file has provenance, then by transitivity, everything has provenance. This approach removes the divide between system components and their histories. Provenance is no longer an add-on to certain data, it becomes a built-in feature of the entire system.

7.2.2 Principle 2: Atomic unit of Provenance

We introduce data provenance as a chain of atomic units. Each atomic unit encapsulates the minimal yet sufficient metadata required to describe a single transformation, interaction, or causal relationship in the lifecycle of a data entity or version. Crucially, each unit contains the necessary information to resolve or locate the next element in the provenance chain, whether that be the agent responsible for its modification, or the context under which the modification occurred.

The atomic unit can be viewed as a discrete, immutable record, conceptually similar to a node in a linked data structure or a commit object in a version control system. Rather than attempting to store or infer the complete history in one place, provenance is reconstructed by traversing these individual units. This atomic design yields multiple advantages. It promotes modularity, since each unit can be inspected, transmitted, or validated in isolation. It supports distributed operation, because different nodes in a networked system can con-

tribute to or access portions of the provenance chain independently. And it encourages composability, since atomic units can be combined dynamically to reconstruct complex provenance narratives without rigid schema coupling.

This atomisation of provenance also aligns well with emerging standards, such as the W3C PROV model, which defines entities, activities, and agents as first-class citizens within a graph structure. By treating each node in this graph as a discrete atomic unit, the system can scale to accommodate asynchronous, federated, or ephemeral computations while retaining semantic rigour. Moreover, atomic provenance facilitates trust and verification: each unit can be cryptographically signed or hashed, enabling tamper-evident chains of provenance that are auditable and secure.

7.2.3 Principle 3: Lazy Evaluation

The adoption of atomic provenance units naturally enables a complementary approach in provenance querying and reconstruction: lazy evaluation. In contrast to eager or up-front provenance resolution, where the full lineage or causal graph of a data object is computed and stored at the point of creation or access, lazy evaluation defers the resolution of provenance relationships until they are actually required.

This deferral is particularly advantageous in environments where provenance graphs are deep, highly interconnected, or sparsely accessed. In such settings, computing the entire provenance chain eagerly is computationally inefficient. A lazy evaluation strategy allows the system to begin with a single atomic unit, such as the most recent transformation record, and traverse the provenance chain on demand, resolving only the specific relationships, entities, or agents pertinent to the current query.

Operationally, lazy evaluation hinges on the atomic unit's ability to encode a pointer or reference to its predecessor. This reference may take the form of a globally unique identifier, a content hash, or a URI that supports retrieval from a distributed storage system. When a provenance query is issued, the system

resolves these references recursively, stepping backwards through the chain of atomic units until the query is satisfied. In many cases, queries will not require full traversal. For example, determining whether a specific user initiated a change, or verifying the origin of a dataset, may only require resolution of a short prefix of the chain.

Lazy evaluation also introduces flexibility in access control and privacy. Since only a subset of the provenance graph is resolved at any given time, it becomes possible to implement fine-grained permissions or redaction policies that expose only the necessary segments of the chain. Furthermore, because evaluation is demand driven, the system can prioritise low latency queries by caching high frequency prefixes, or exploit distributed resolution strategies that parallelise traversal across network nodes.

7.3 Architecture Overview

This section outlines the overall architecture of the prototype system developed to explore the two core ideas of this thesis: (i) capturing deltas as part of data provenance, and (ii) embedding that provenance directly within files. The architecture is organised into five interacting components, as also depicted in Figure 7.1:

- Delta Extraction Component
- Provenance Model Component
- Provenance Embedding Component
- Provenance Access Interface
- Provenance Explorer Component

Collectively, these components support end-to-end capture, representation, storage, embedding, and inspection of provenance information.

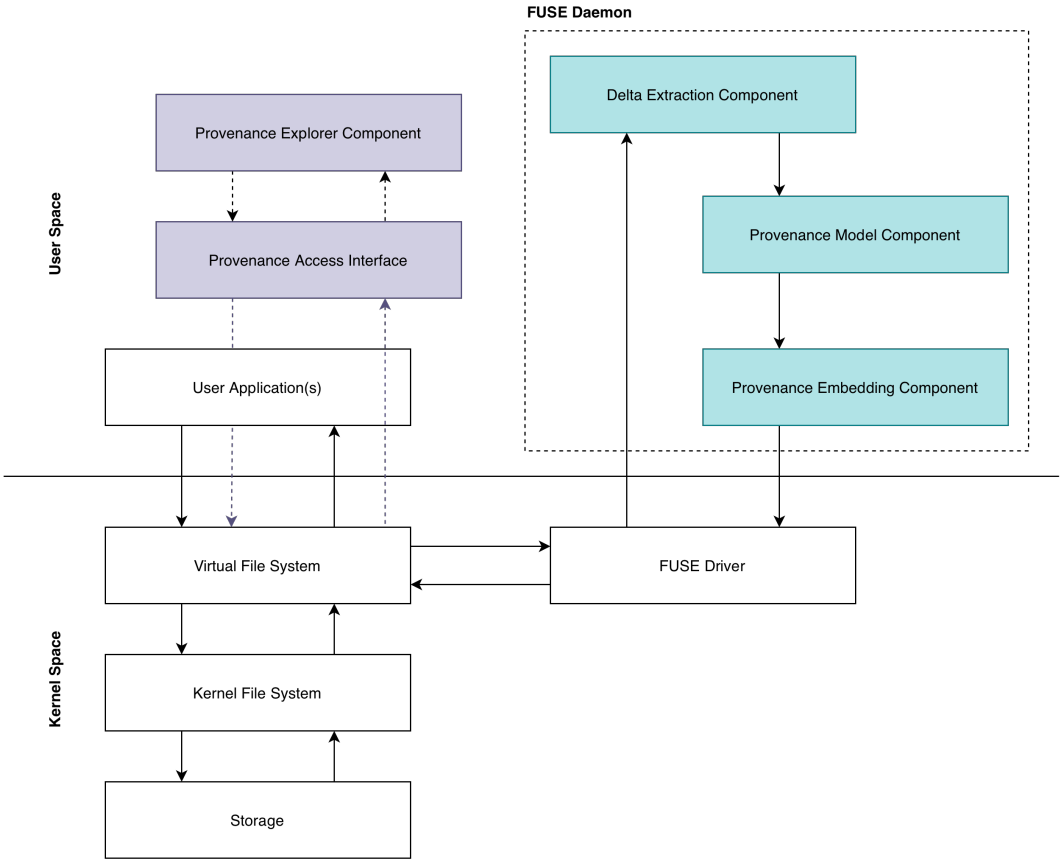


Figure 7.1: Prototype architecture overview.

7.3.1 Delta Extraction Component

The Delta Extraction Component is responsible for computing and representing the differences between successive versions of files. It exposes an interface through which users, developers, or researchers can introduce their own delta algorithms, acknowledging that no single differencing strategy suits every format and that the compressibility and expressiveness of deltas are ultimately shaped by the entropy of the underlying data.

The module offers a standardised abstraction: given two sequences representing consecutive versions, an algorithm implementation must return a structured delta object describing the detected changes. The prototype ships with a straightforward byte level delta algorithm that serves as both a reference and a fallback. Crucially, the architecture avoids dictating any particular technique. Instead, it enables alternative algorithms to be loaded at runtime, from

traditional line based diffs for text, to semantic diffs for structured formats, to block matching methods optimised for large or complex binary data.

This extensibility matters because delta quality is tightly coupled to data characteristics. Methods that produce concise deltas for low entropy text often struggle with compressed or encrypted material, multimedia content, or domain specific formats where meaningful changes cannot be inferred from surface byte patterns. Conversely, algorithms designed for binary data can generate overly complicated deltas when applied to human readable sources.

To support this flexibility, the system uses a file's extension as a routing key for algorithm selection. When a known extension is encountered, the corresponding specialised algorithm is invoked automatically. Adding a new algorithm is straightforward. A developer registers an implementation against one or more extensions, and the module incorporates it without further modification.

When no specialised handler exists for a given extension, the module reverts to the generic byte level algorithm. This baseline ensures consistent, format agnostic behaviour across all files and prevents provenance capture from failing simply because an appropriate optimised differencing is unavailable. In this way, the generic algorithm provides a dependable safety net, preserving the continuity and completeness of the provenance chain even when the system processes unfamiliar or heterogeneous data.

7.3.2 Provenance Model Component

The provenance model of the architecture provides the conceptual and operational foundations for representing how data evolves over time. Its task is not only to record deltas, but to organise them within a structured provenance graph that captures the semantics of change. The model used in the prototype draws on the W3C PROV-DM specification, interpreting file and content level operations as activities that generate, use, derive, or invalidate entities. Observable operations that result in new content, new metadata, or a new

structural state are treated as activities that leave a provenance trace linking the prior entity to its successor.

The rationale behind the mapping is that any interaction with a data object can be understood through its effect on the state of that object. An activity that accesses data without altering it is interpreted as *using* an existing entity. An activity that produces new content or a new version of an existing entity is viewed as *generating* an entity. An activity that modifies an object without creating an entirely new one constitutes a *derivation*, because it expresses that the new state depends on the previous one.

Read-oriented operations, including opening a file, reading its contents, or querying its attributes, map to the PROV-DM *used* relationship. The assumption is that these operations rely on the continued existence and integrity of a prior entity. Even in a prototype that embeds provenance within a file, such operations remain relevant, that is when reconstructing history or validating embedded provenance, the system must record which activities consumed specific states, thereby capturing dependency chains between entities and their use in later processes.

Operations that introduce new content or materialise a new state of an object correspond to the *wasGeneratedBy* relationship. Writing new bytes, truncating a file to produce a shorter version, or creating new entries in a directory all fall into this category. The delta extraction module captures the fine grained change, while the provenance model captures the semantic notion that this state is newly generated. This is especially important for the purposes of reconstruction, since the graph explicitly records when and how each version came into being.

Transformative operations that alter properties, attributes, or structure without creating a distinct object contribute to *wasDerivedFrom* relationships. These are changes that can be traced to one or more predecessor states and are therefore derivations rather than independent generations. Renaming a file, altering its permissions, or updating timestamps fall into this category.

The provenance model records these as derivations because the resulting state is meaningfully connected to the previous one. This distinction is important for later reasoning tasks. For example, when tracing provenance, derivations clearly express continuity between two states, whereas generations indicate discrete steps in the lifecycle of an entity.

It should be noted that this mapping is not presented as a definitive or universal standard. Provenance semantics depend heavily on domain conventions, policy interpretations, and the granularity at which data is observed. A different prototype might treat the same operations differently; for example, a system with strong notions of logical objects could choose to treat renaming as generating a new entity, while another system might collapse generation and derivation entirely. The intention here is to provide a mapping that is internally consistent, conceptually meaningful, and effective for the purposes of experimenting with embedded provenance and delta based reconstruction. The relationships chosen are therefore adequate rather than authoritative, and they leave room for extension.

Within the prototype, each observed operation is translated into an atomic provenance unit that captures both the PROV-compliant representation of the event and the content level change it introduces. Each unit consists of an entity representing the new state of the object, an activity denoting the operation being performed, and an agent responsible for invoking that operation, together with the PROV-DM relations that describe how the activity interacts with the entity.

To incorporate the delta based design of the system, the atomic unit also includes the delta itself, treated as a first-class component rather than an implicit property of the entity. This ensures that the provenance graph records not only the semantic relationship between states but also the concrete material difference that produced the new version. A *write()* operation, for example, becomes a unit in which the entity reflects the post write state: the activity corresponds to the write, the agent denotes the process owner, the relation

indicates that the new entity was generated by the activity, and the delta encodes the precise modification from the previous version. A *rename()* becomes a derivation event linking two states, with the delta capturing the structural alteration.

Formally, each provenance event is represented as a 5-tuple:

$$U = (e, a, g, R, \Delta)$$

where e is the resulting entity, a the activity, g the agent, R the applicable PROV-DM relations, and Δ the computed difference from the immediately preceding entity. These 5-tuples form the atomic building blocks of the provenance graph, allowing the system to express both the semantics and the mechanics of change.

7.3.3 Provenance Embedding Component

The Naïve Approach

A natural starting point when considering how to embed data provenance into files is to treat the data provenance as an extension of the data file, either by prepending or appending to the data. In broad brushstrokes, this is essentially how the Flexible Image Transport System [193] is implemented, as discussed in Chapter 6.

In addition to the linear growth driven by file changes, provenance data expands disproportionately during intensive access and transformation events such as edits, reads, migrations, and replication. Maintaining provenance contiguously with content therefore becomes a performance bottleneck in large files and high throughput systems. Each content extension can trigger cascades of provenance rewrites or relocations, eroding locality guarantees and reducing overall throughput.

Moreover, placing provenance before or after the content assumes a predictable and uniform pattern of data evolution, an assumption that breaks down in real world usage. Provenance systems must be resilient to partial

writes, crashes, and concurrent modifications. A monolithic file layout, where provenance and content share physical space in a naïvely linear structure, makes atomicity and recovery significantly harder to guarantee.

Block-based File Structure

Rather than treating the file as a flat byte stream with provenance appended at the beginning or end, the embedding layer treats the file as a lightweight container composed of several logical block categories, as described below. These blocks are defined and managed within the file itself, independently of the underlying filesystem's storage blocks (e.g., 4, 8, or 16 KB) or physical disk sectors. Conceptually, this approach is similar to how modern file systems represent data on disk, where metadata and content are managed separately through structures such as *i-nodes* rather than as a single linear stream. For data provenance, this model is extended by introducing additional, well-defined logical block types that embed provenance information alongside content while remaining logically distinct from it. The underlying filesystem remains responsible for mapping these logical structures onto physical storage.

Data Blocks holds the primary content of the file. These blocks are simply the present state that ordinary applications care about. For a text document, the data block contains the current text; for a binary format, it contains the current binary payload. When the user edits the file and saves it, the data block is rewritten to reflect the new content, just as it would be in a traditional file without embedded provenance. The embedding layer treats this payload as opaque: it does not require any particular structure beyond being addressable as a contiguous region identified in the header.

Version Blocks stores information about each version. These blocks act as a contiguous storage region for the delta stream, as illustrated in Figure 7.2. Deltas are written sequentially and may span multiple blocks depending on their size. A small delta might occupy only part of a block, while a larger one may run across several blocks. The version blocks themselves do not define

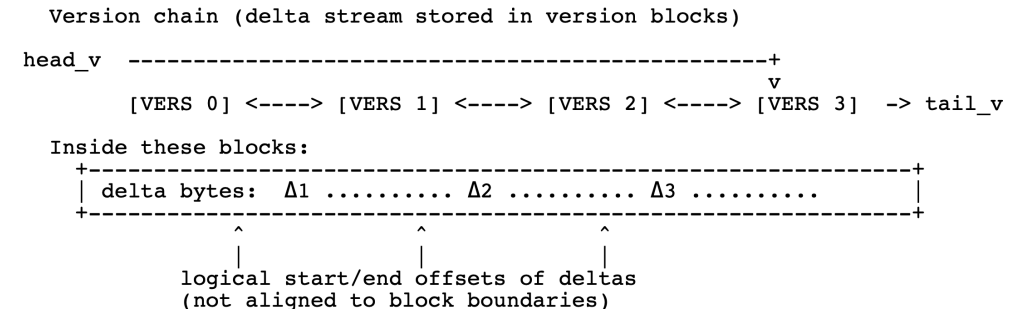


Figure 7.2: Illustration of version blocks.

where one delta ends and the next begins; they simply provide the space into which the delta bytes are written. The blocks are linked in a doubly linked list so that a reader can traverse the delta stream forwards or backwards without scanning the entire file.

Provenance Blocks form a separate doubly linked list that introduces the semantic layer of the format. They define how deltas should be interpreted, which activities produced them, and how those activities relate to successive versions of the file. Each block contains a provenance record describing an operation and its associated metadata, and each record anchors itself in the delta stream by storing the offsets of the delta or sequence of deltas it concerns. Through this linkage, the provenance blocks provide the semantic structure that the version blocks do not encode.

Separating the storage of deltas from the provenance that interprets them has several advantages. The delta stream can grow efficiently as an append-only sequence, making it straightforward to write and replay. The provenance can evolve without disturbing the underlying delta storage: additional annotations, alternative interpretations, or higher level workflow information can be added simply by appending new provenance blocks. The data block is similarly isolated, so updating the file's content does not force any rewriting of existing provenance or delta data.

The file is divided into fixed size blocks, which are grouped into rows. Each row starts with a bitmap block followed by a number of regular blocks. The bitmap encodes, bit by bit, which blocks in that row are free and which are

```
Bitmap Row 1 bits (conceptual):

  [1][1][1][0][1][0]
   |  |  |  |  |  |
   |  |  |  |  |  +--> block 5 = FREE
   |  |  |  |  +-----> block 4 = OCCUPIED
   |  |  |  +--------> block 3 = FREE
   |  |  +-----------> block 2 = OCCUPIED
   |  +--------------> block 1 = OCCUPIED
   +-----------------> the bitmap block itself
```

Figure 7.3: Example of bitmap representation of a file.

in use as demonstrated in Listing 7.3. New data, versions, or provenance are written by finding free blocks via the bitmap and allocating them as required. When a block becomes unused, it is marked `FREE` in the bitmap, and only when an entire row is free can that row be deallocated. This design allows blocks to be allocated and freed without shifting the rest of the file, avoiding the need to rewrite long tails of data whenever something changes.

Bitmap Blocks sit at the start of each row and act as the allocator for the row beneath them. A reader or writer first consults the relevant bitmap to find unused blocks, then allocates or frees those blocks as needed. Because allocation happens in units of blocks and never requires shifting existing data, the cost of inserting new provenance, or new delta bytes, is independent of the size of the file. The bitmap therefore improves the efficiency of the embedding scheme, it gives a compact way to manage sparse usage over a file that may contain interleaved data, version, and provenance blocks.

Header Block acts as the entry point to the file's internal structure, and ties these components together. The header contains pointers to the heads and tails of the data, provenance, and version block, as well as structural metadata, checksums, schema identifiers, and any additional information required to interpret the rest of the file. By keeping the header compact and fixed at a known offset, the file can be opened and navigated deterministically, without scanning or parsing the entire content.

Figures 7.4 and 7.5 illustrate the logical and physical organisation of the proposed block based file structure. As shown in Figure 7.4, the *Header Block*

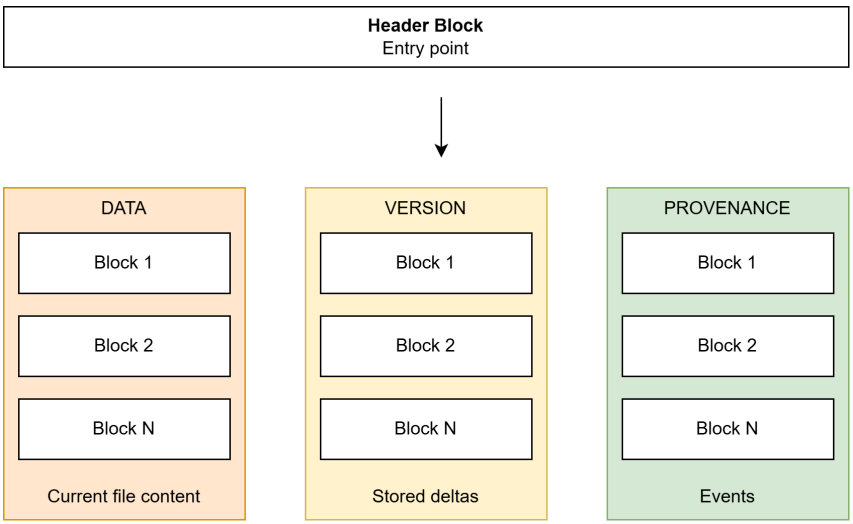


Figure 7.4: Logical structure of provenance file.

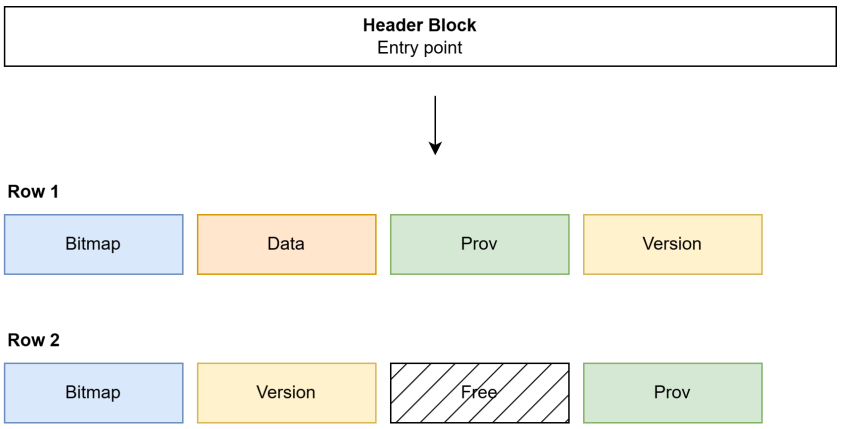


Figure 7.5: Example physical allocation.

serves as the entry point to the file, containing structural metadata and pointers to the *Data*, *Version* and *Provenance Blocks*. The *Data Blocks* store the current content of the file, the *Version Blocks* maintain the sequence of deltas representing changes between versions, and the *Provenance Blocks* record the activities, agents and relationships associated with those changes. These blocks are logically independent rather than nested within one another. Figure 7.5 illustrates their physical arrangement, where the file is divided into fixed-size blocks organised into rows. Each row begins with a *Bitmap Block* that tracks allocated and available blocks. *Data*, *Version* and *Provenance Blocks* can therefore be distributed across different rows rather than stored in a predefined sequence. The *Header Block* remains at a fixed, known offset, allowing

the system to locate the relevant blocks through pointers without scanning the entire file. Figure 7.5 presents a conceptual arrangement; the precise placement of the header relative to the bitmap rows is not specified.

When an application requests a file read, the operation passes through the VFS layer to the FUSE filesystem, which invokes the corresponding *pfs_read()* handler. The embedding layer first accesses the *Header Block* to identify the location of the required *Data Blocks*, then retrieves the requested bytes from those blocks. In parallel with servicing the read, the provenance capture mechanism obtains the calling process information, including its identity and timestamp, and constructs a provenance record representing the operation using the *PROV-DM used* relationship. This record is stored in the *Provenance Blocks*, with the necessary block allocation and metadata updates managed by the embedding layer. Since a read does not modify the file's content, no new delta is generated and the *Version Blocks* remain unchanged. The requested data is then returned to the application through FUSE. This sequence illustrates the intended interaction between the block types.

7.3.4 Provenance Access Interface

The Provenance Access Interface provides an entry point for applications that need to interact with the provenance embedded within a file. Its role is to expose the internal block based structure used by the embedding component without performing any provenance computation itself. It does not generate deltas, interpret semantic meaning, or manage graph operations. Instead, it offers an abstraction over how provenance is stored, retrieved, and navigated, so that higher level tools can rely on a consistent representation regardless of the embedding strategy chosen for a particular file.

The interface understands how blocks are arranged, how they are indexed, and how they reference one another. It presents these elements as structured objects that can be queried, streamed, or passed to other components that perform analysis or reconstruction. This separation ensures that format awareness

remains isolated within one layer of the system. The provenance access layer can therefore accommodate variations in how provenance is stored, whether in container subdirectories, metadata streams, or appended segments, while offering a unified view to client applications.

The interface supports operations that expose the stored record of deltas, identifiers, checksums, state descriptors, and any schema defined metadata. It allows applications to examine the lineage encoded in the file, determine the presence and completeness of provenance information, and retrieve the raw data needed for downstream processes. By presenting an ordered and addressable view of the embedded provenance blocks, it becomes possible for external components to reconstruct earlier states, validate integrity markers, or integrate provenance into broader workflows without needing awareness of the underlying file format intricacies.

The design focuses on reliable access rather than interpretation. Decisions about how to process deltas, validate histories, or integrate provenance into a graph engine remain outside this layer. This boundary keeps the interface stable even as diffing strategies, provenance schemas, or embedding formats evolve. The result is an access mechanism that provides predictable structure for the rest of the architecture while remaining agnostic to the semantics of the provenance it exposes.

Figure 7.6 illustrates the conceptual class diagram of the Provenance Access Interface and its relationships with the other components of the system. The interface provides a unified abstraction for accessing provenance information embedded within files, exposing operations for retrieving delta, identifiers, checksums, state descriptors, and associated metadata. It also supports navigation of the underlying block structure, allowing applications to locate and retrieve provenance records without requiring knowledge of the physical storage layout.

The interface operates between the Provenance Explorer and the block based storage structure managed by the Provenance Embedding Component.

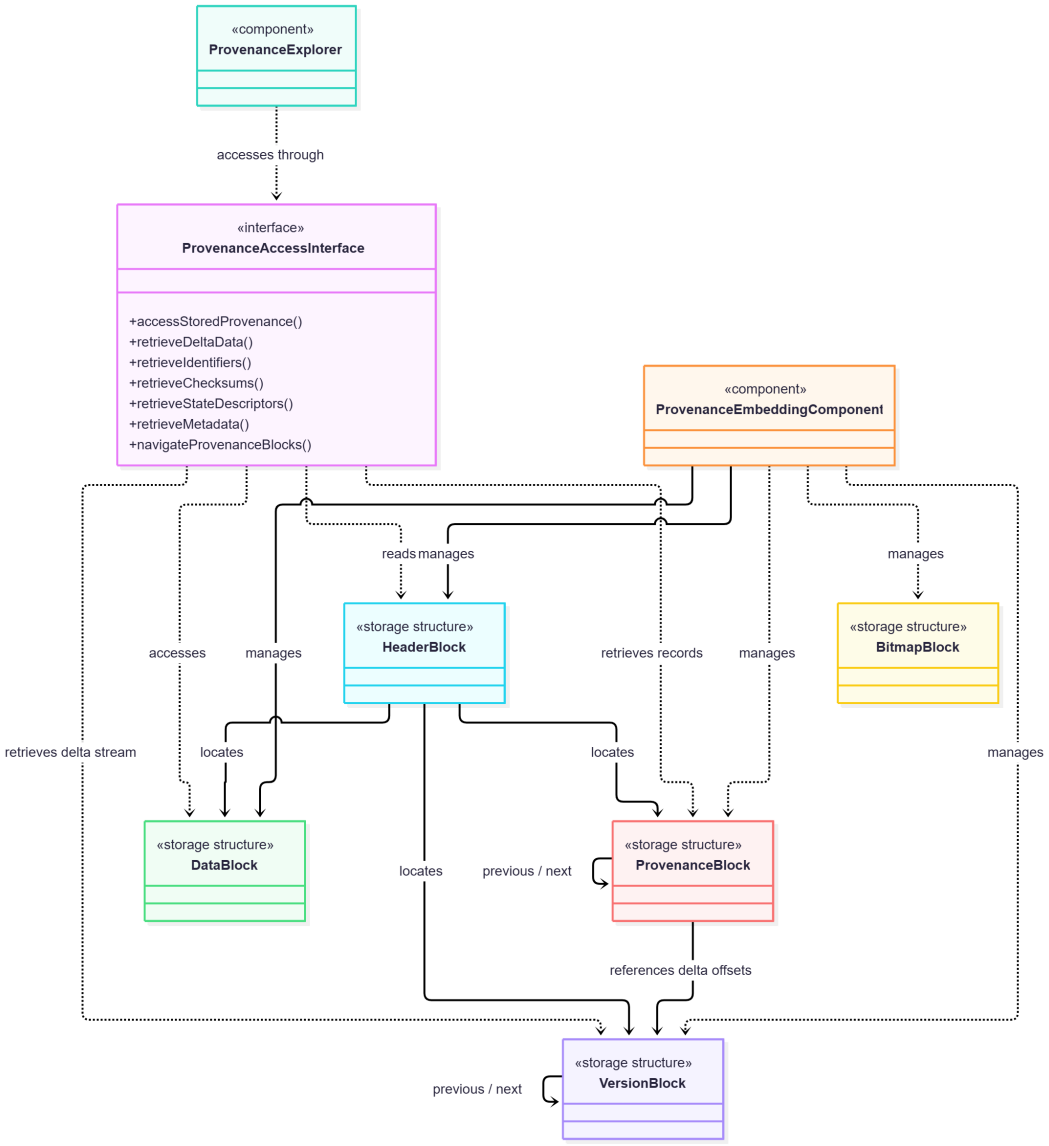


Figure 7.6: Class Diagram of the Provenance Access Interface.

Through this separation, the Provenance Explorer can access the header, data, version, and provenance blocks without directly handling their storage organisation. The interface is deliberately limited to data access and navigation, leaving provenance interpretation, historical reconstruction, and integrity validation to higher level components. This separation of concerns allows the underlying embedding mechanism to evolve without requiring corresponding changes to applications that consume provenance information.

7.3.5 Provenance Explorer Component

The Provenance Explorer acts as the mechanism for interacting with files that contain embedded provenance. Its purpose is to provide a provenance-aware environment that can interpret the internal structure produced by the embedding layer and to give users, developers, and researchers direct access to the history, semantics, and evolution of a file without relying on external provenance stores. Within the broader architecture, the explorer tool functions as the counterpart to the embedding process: where the embedding layer writes deltas and provenance information into the block based file structure, the explorer consumes and interprets those structures to reconstruct, verify, and analyse the lifecycle of the data.

Because the embedding layer organises a file into a header block, data blocks, version blocks, and provenance blocks, the tool must be able to parse this set of block types. The process begins by reading the header block, which provides the offsets and pointers required to locate each structural component. By using these fixed pointers rather than scanning the file from start to finish, the tool can load the internal state efficiently, even for files with long provenance histories or large delta streams. Once the header is parsed, the tool can access the current data, the delta stream, or the embedded provenance graph independently of one another, mirroring the architectural separation between the content, the raw deltas, and the semantic interpretation of change.

A central task for the tool is to interpret the provenance blocks. Each provenance block contains a series of entries, each of which represents a provenance unit in the form $U = (e, a, g, R, \Delta)$. The tool reconstructs the provenance graph by reading these units in sequence and instantiating the entities, activities, agents, and PROV-DM relations they describe, while also resolving their references into the delta stream. The provenance blocks are arranged as a doubly linked list, which allows the tool to traverse the units in either direction. Forward traversal is useful when incrementally exploring the lineage of a file, whereas backward traversal supports analysis that begins with the current state and works towards earlier states.

For reconstruction, verification, or analytical tasks, the tool must also navigate the delta stream stored in the version blocks. These blocks contain an append-only sequence of raw deltas. Unlike the provenance blocks, the version blocks do not mark where one delta ends and the next begins. Instead, segmentation is provided by the offsets stored in the provenance entries. The tool uses these offsets to extract the precise delta bytes corresponding to each provenance unit. This approach preserves the architectural separation between the raw material of change and the semantic information that interprets it. When reconstructing a historical version, the tool identifies the target entity in the provenance graph, retrieves the deltas associated with that entity, reads the corresponding segments of the delta stream, and then replays them using the algorithm appropriate to the delta type. The tool does not need to know whether a delta originated from, a semantic diff, a byte level comparison, or a domain specific transformation. Instead, it relies on the routing and algorithm selection mechanisms introduced in the Delta Extraction Component, ensuring continued compatibility as new delta algorithms are introduced.

Reconstruction is straightforward. The tool locates the entity to be materialised, determines the ordered sequence of deltas associated with that entity through its provenance links, reads the relevant sections of the delta stream, and applies the deltas in order to rebuild the file's content at that moment in

its history. The ability to materialise any past state is essential not only for auditing and forensic tasks but also for validating the correctness and integrity of the embedded provenance itself.

The tool also functions as a validator. Because the file format contains checksums, pointers, and structural metadata, the tool can examine the internal consistency of the file. It verifies that header pointers resolve correctly, that the linked lists of version and provenance blocks form coherent sequences, that deltas reconstruct to content whose hash matches the stored hash in the corresponding entity, and that each provenance unit contains the information required to interpret its associated delta. These checks provide assurances about the integrity of embedded provenance and the reliability of reconstruction, which is especially important when files may have been transmitted, replicated, or subjected to partial updates.

To support external analysis and integration with existing provenance tools, the tool is able to export the reconstructed provenance graph into common formats. These include serialisations of the W3C PROV model as well as generic graph representations suitable for analytical frameworks. For human facing inspection, the tool can render a textual account of the file's evolution, showing how the states, activities, and agents interrelate and annotating each step with delta information such as size, type, and timestamp. This narrative view can be particularly valuable when demonstrating the behaviour of the prototype, as it makes the internal provenance directly interpretable without requiring specialised tooling.

In this way, the provenance explorer completes the architecture. It provides deterministic parsing of the file's internal block structure, reconstruction of provenance graphs, reliable extraction and replay of deltas, and thorough validation of integrity and consistency. By acting as the interpretive layer that reveals the semantics and mechanics of change, it demonstrates the feasibility of embedding delta based provenance directly into the artefact it describes.

7.4 Prototype Implementation

In this section, we present the implementation details of the proposed system. We outline how the system captures versioning, focusing on the delta between versions rather than just acknowledging the creation of new versions. We describe the mechanism by which provenance is embedded directly into the file, ensuring that each file carries its own historical context. Additionally, we explain how provenance is associated with specific versions of data, rather than the files themselves, and how this approach enables querying the relationships between files based on their interactions. This section provides the technical foundation of the system, detailing the underlying architecture, data structures, and algorithms used to achieve these goals.

7.4.1 Filesystem in Userspace (FUSE)

As previously introduced in Chapter 2, our prototype leverages FUSE to facilitate rapid development. The choice of FUSE aligns with our primary objective: to demonstrate the feasibility of the proposed approach rather than to optimise for performance or completeness. While FUSE introduces certain limitations, such as an inability to capture transient data that exists solely in memory, it nevertheless validates the core premise of this thesis.

Importantly, this limitation is not intrinsic to the design itself but rather to the choice of implementation. The same mechanism, if realised using alternative technologies such as LSM (Linux Security Modules) or eBPF (Extended Berkeley Packet Filter), could achieve broader coverage and greater efficiency. Indeed, as noted in the earlier review of related work, other approaches [121, 8] to provenance have demonstrated the computational benefits that result from utilising such kernel-layer features in their implementations.

FUSE has been successfully used in other research prototypes such as StoryBook [3] and factFUSE [206], reinforcing its reliability and ease of integration. It is also widely used in mainstream software systems such as VeraCrypt [207]

and EncFS [208], further affirming its practicality. However, a key drawback of FUSE is its performance overhead, primarily due to costly context switches between kernel and user space. Our aim, however, is not to optimise performance, but to show that even under these constraints, the mechanism is functional. It follows that with a more efficient implementation, the potential performance gains would be significant. These considerations, while relevant, fall outside the scope of this thesis.

The code base is split into four static libraries and two main executables:

- *libcapture.a* implements the FUSE file system and the capture path that invokes delta extraction and provenance encoding.
- *libdelta.a* contains the pluggable delta algorithms and their registration and routing logic.
- *libprovmodel.a* implements the in-memory representation of entities, activities, agents and provenance units and serialises them into the on-disk format.
- *libembed.a* implements the block allocator, header and bitmap management and the (de)serialisation of version and provenance blocks into the container file.
- *pfsd* is the FUSE-based provenance daemon process.
- *pfsexplorer* is a command line explorer and reconstruction tool that attaches to a mounted file or a raw container.

7.4.2 fuse_operation

The foundation of the FUSE based implementation is the *fuse_operations* struct, a *C* structure defined in the FUSE API. This struct contains function pointers corresponding to the various filesystem operations that the kernel might invoke through the Virtual File System (VFS) layer. Implementers populate this struct with callbacks that handle file and directory operations such

as reads, writes, attribute queries, and metadata management. An example of the *fuse_operations* struct is shown in Listing 7.1.

```
static const struct fuse_operations pfs_oper = {
    ...
    .init       = pfs_init,
    .open       = pfs_open,
    .create     = pfs_create,
    .read       = pfs_read,
    .write      = pfs_write,
    .statfs     = pfs_statfs,
    .release    = pfs_release,
    .fsync      = pfs_fsync,
    .lseek      = pfs_lseek,
    ...
};
```

Listing 7.1: *fuse_operation* struct example.

The set of operations that can be implemented through *fuse_operations* corresponds to a reasonably complete subset of the standard POSIX file operation interface. Notably, this includes support for:

- Basic I/O operations: (*open*, *create*, *read*, *write*, *truncate*, *release*, *fsync*, *fallocate*, *copy_file_range*, *lseek*).
- Metadata and permission operations: (*getattr*, *chmod*, *chown*, *utimens*, *setxattr*, *getxattr*, *listxattr*, *removexattr*).
- Namespace operations: (*mknod*, *mkdir*, *symlink*, *unlink*, *rmdir*, *rename*, *link*).
- Directory traversal (*readdir*, *readlink*, *access*).
- Filesystem-level operations (*statfs*).

In our implementation, we defined and implemented the full set of operations in the *fuse_operations* structure in order to construct a fully functional filesystem. This approach ensures that the prototype would behave correctly under a wide range of file and directory operations, and is especially important when we conduct benchmarks, such as those reported on in Chapter 8.

While it is technically possible to implement only a subset of operations for more constrained use cases, omitting handlers for standard calls often results in undefined or inconsistent behaviour, which we sought to avoid.

Although the complete implementation was necessary for correctness and compatibility, the majority of operations remain largely pass-through in nature. For example, operations such as *getattr*, *readdir*, *unlink*, and *rename* simply forward the request to the underlying filesystem after capturing relevant provenance data. The *read* and *write* operations sees the most amount of modification as this is where the embedding of data provenance and versioning takes place. Implementation details will be covered in subsequent sections.

The principle behind the mapping is the interpretation of file operations as either consuming, generating, transforming, or invalidating data. These semantics are then reflected through PROV-DM relations: *used*, *wasGeneratedBy*, *wasDerivedFrom*, and *wasInvalidatedBy*. Table 7.1 details the *fuse_operation* to PROV-DM mapping used in our prototype.

Each handler performs the following steps:

1. Resolve the calling context: process ID, user and group IDs, executable path and timestamp.

2. Forward the call to the underlying file system (the "real" path), performing any reads needed to capture overwritten bytes.

3. Invoke the delta extraction library with the old and new byte sequences.

4. Construct a provenance unit and write it, together with the delta, into the container via *libembed*.

5. Return the kernel's result to FUSE.

Table 7.1: *fuse_operation* to PROV-DM mapping.

Operation	**Relation**
.access	used
.chmod	wasDerivedFrom
.chown	wasDerivedFrom
.copy_file_range	used
.create	wasGeneratedBy
.fallocate	wasDerivedFrom
.fsync	wasGeneratedBy
.getattr	used
.getxattr	used
.link	wasGeneratedBy
.listxattr	used
.lseek	used
.mkdir	wasGeneratedBy
.mknod	wasGeneratedBy
.open	used
.read	used
.readdir	used
.readlink	used
.release	used
.removexattr	wasInvalidatedBy
.rename	wasDerivedFrom
.rmdir	wasInvalidatedBy
.setxattr	wasDerivedFrom
.statfs	used
.symlink	wasGeneratedBy
.truncate	wasGeneratedBy
.unlink	wasInvalidatedBy
.utimens	wasDerivedFrom
.write	wasGeneratedBy

```c
static void fill_context(struct pfs_context *ctx)
{
    ctx->pid = fuse_get_context()->pid;
    ctx->uid = fuse_get_context()->uid;
    ctx->gid = fuse_get_context()->gid;

    char path[64];
    snprintf(path, sizeof(path), "/proc/%d/exe", ctx->pid);
    ssize_t len = readlink(path, ctx->exe_path, sizeof(ctx->
    exe_path) - 1);
    if (len >= 0) {
        ctx->exe_path[len] = '\0';
    } else {
        strcpy(ctx->exe_path, "<unknown>");
    }

    clock_gettime(CLOCK_REALTIME, &ctx->ts);
}
```

Listing 7.2: Context resolution helper function.

To associate each file operation with a specific agent and execution context, the prototype uses */proc* to obtain process identifiers and the executable path. The helper in Listing 7.2 populates a *pfs_context* structure. It is invoked by all provenance-aware handlers and ensures that provenance units carry a consistent representation of the agent, activity time and binary/executable. It queries FUSE for the current process, resolves the executable symlink in */proc/<pid>/exe*.

7.4.3 Write Handler

```c
static int pfs_write(const char *path, const char *buf, size_t
    size, off_t offset, struct fuse_file_info *fi)
{
    struct pfs_context ctx;
    fill_context(&ctx);

    uint8_t *old = malloc(size);
    ssize_t old_read = pread(fi->fh, old, size, offset);
    if (old_read < 0) {
        free(old);
        return -errno;
    }
```

```
12
13      ssize_t written = pwrite(fi->fh, buf, size, offset);
14      if (written < 0) {
15          free(old);
16          return -errno;
17      }
18
19      struct delta_buffer delta = {0};
20      const char *ext = get_ext(path);
21      const struct delta_algorithm *alg = get_delta_alg(ext);
22      if (!alg) alg = delta_default();
23
24      if (alg->compute((const uint8_t *)buf, written,
25                       old, old_read, &delta) != 0) {
26          delta_init_raw(&delta, old, old_read);
27      }
28
29      struct pfs_unit u;
30      pfs_init_write(&u, path, &ctx, &delta, offset, written);
31      embed_append_unit(path, &u);
32
33      free(old);
34      delta_free(&delta);
35      return (int)written;
36  }
```

Listing 7.3: Write handler with delta capture.

The write operation introduces new content versions. The *pfs_write* handler performs four steps: (i) it copies out the bytes about to be overwritten, (ii) executes the actual write, (iii) computes a delta using the selected algorithm, then (iv) constructs and embeds a provenance unit that records the change and links it to the delta in the version stream. Listing 7.3 gives the code for the write handler with capture. It demonstrates how old content is collected, how the delta module is invoked and how the provenance unit is passed to the embedding layer.

pfs_truncate mirrors this structure but reads the truncated tail as the old payload, and stores a delta that either holds the truncated bytes or a compact encoding of the size change. The *pfs_read* handler records provenance as *used* relationships but does not produce deltas.

7.4.4 libdelta

```
typedef enum {
    DELTA_TYPE_RAW = 0,
    DELTA_TYPE_BYTE_DIFF = 1,
    DELTA_TYPE_SEMANTIC = 2,
} delta_type_t;

struct delta_buffer {
    delta_type_t type;
    uint8_t     *data;
    uint32_t     length;
};

struct delta_algorithm {
    const char *name;
    const char *extensions[8];
    int (*compute)(const uint8_t *new_buf, size_t new_len,
                   const uint8_t *old_buf, size_t old_len,
                   struct delta_buffer *out);
    int (*apply)(const uint8_t *base, size_t base_len,
                 const struct delta_buffer *delta,
                 uint8_t **result, size_t *result_len);
};
```

Listing 7.4: Delta types and algorithm interface.

The delta extraction module *libdelta* wraps individual algorithms behind an abstraction. Algorithms can be swapped or extended without changing the rest of the system. The core types record which kind of delta is produced, the buffer that holds the delta bytes, and a function pointer table for computing and applying deltas.

Listing 7.4 defines the delta-related types. *delta_type_t* distinguishes raw copies, simple byte level differences and more semantic forms. *struct delta_algorithm* represents a single pluggable algorithm and declares C function pointers for forward and reverse application.

```
#define MAX_ALGORITHMS 16
static struct delta_algorithm *g_algs[MAX_ALGORITHMS];
static size_t g_alg_count = 0;

void delta_register(struct delta_algorithm *alg)
{
```

```
    if (g_alg_count < MAX_ALGORITHMS)
        g_algs[g_alg_count++] = alg;
}

const struct delta_algorithm *delta_find_for_extension(const
    char *ext)
{
    for (size_t i = 0; i < g_alg_count; ++i) {
        for (size_t j = 0; g_algs[i]->extensions[j]; ++j) {
            if (strcmp(g_algs[i]->extensions[j], ext) == 0)
                return g_algs[i];
        }
    }
    return NULL;
}
```

Listing 7.5: Algorithm registry.

Algorithms are registered in a simple in-memory registry. At run time, the capture layer queries this registry to select an algorithm based on file extension, falling back to a default when no specialisation is available. The registry uses a fixed size array to keep the implementation simple. Listing 7.5 shows the algorithm registry implementation. It provides functions to register new algorithms and to look up an algorithm by extension.

```
function compute_diff(new_buf, old_buf):
    i := 0
    out := empty buffer
    while i < len(new_buf) or i < len(old_buf):
        run_eq := 0
        while i + run_eq < len(new_buf) and
              i + run_eq < len(old_buf) and
              new_buf[i + run_eq] == old_buf[i + run_eq]:
            run_eq++
        if run_eq > 0:
            emit(OP_SKIP, run_eq)
            i += run_eq
        run_diff := 0
        while i + run_diff < len(new_buf) and
              (i + run_diff >= len(old_buf) or
               new_buf[i + run_diff] != old_buf[i + run_diff]):
            run_diff++
        if run_diff > 0:
            emit(OP_REPLACE, run_diff, new_buf[i .. i+run_diff
    ])
```

```
            i += run_diff
    return out
```

Listing 7.6: Example: Pseudocode of byte-diff algorithm.

Listing 7.6 is an example algorithm that encodes byte level differences between two buffers. It scans the buffers in lockstep, emitting skip instructions for unchanged runs and replace instructions for differing runs. This produces a compact representation when content changes are localised. Its *compute* function walks *old* and *new* and emits a simple patch stream.

```
static struct delta_algorithm text_alg = {
    .name = "text-line",
    .extensions = { ".txt", ".md", ".tex", NULL },
    .compute = text_diff_compute,
    .apply   = text_diff_apply,
};

void delta_init_all(void)
{
    delta_register(&text_alg);
    delta_register(...);
    delta_register(...);
}
```

Listing 7.7: Initialising delta algorithms.

Specialised algorithms will implement the same compute/apply pair and are registered with appropriate extensions at initialisation as in Listing 7.7.

7.4.5 Provenance Model

```
typedef uint64_t pfs_id_t;

typedef enum {
    REL_USED        = 1u << 0,
    REL_WAS_GEN_BY  = 1u << 1,
    REL_WAS_DERIVED = 1u << 2,
} pfs_rel_t;

struct pfs_entity {
    pfs_id_t id;
    uint64_t file_hash[4];
    uint64_t size;
```

```
    uint64_t mtime_ns;
    uint64_t path_offset;
};

struct pfs_activity {
    pfs_id_t id;
    uint64_t ts_ns;
    uint64_t kind;

struct pfs_agent {
    pfs_id_t id;
    uint32_t uid, gid;
    uint32_t pid;
    uint64_t exe_offset;
};

struct pfs_unit {
    struct pfs_entity  ent;
    struct pfs_activity act;
    struct pfs_agent   ag;
    uint32_t rel_mask;
    uint32_t reserved;
    uint64_t delta_offset;
    uint32_t delta_length;
    uint8_t delta_type;
    uint8_t pad[7];
};
```

Listing 7.8: Provenance tuple data structures.

The provenance model uses a fixed record layout so units can be written directly into provenance blocks without further serialisation. Each unit contains one entity, one activity, one agent, a relation mask and a link to the delta in the version stream. String attributes such as paths are not stored directly but through offsets into a separate string table. Listing 7.8 defines the provenance data structures. *pfs_entity* records file level state, *pfs_activity* records the operation that produced that state, and *pfs_agent* records the process and user responsible. *pfs_unit* ties them together and adds pointers to the delta bytes.

For each write captured by the FUSE layer, the prototype initialises a new provenance unit. This helper fills in the entity, activity and agent fields using the context collected earlier, sets the relation mask and records the metadata

about the delta. The embedding layer later attaches a concrete on-disk offset for the delta bytes.

```
void pfs_init_write(struct pfs_unit *u,
                    const char *rel_path,
                    const struct pfs_context *ctx,
                    const struct delta_buffer *delta,
                    off_t offset, size_t len)
{
    memset(u, 0, sizeof(*u));
    u->ent.id = pfs_next_id();
    u->ent.size = file_current_size(rel_path);
    u->ent.mtime_ns = now_ns();
    u->ent.path_offset = strings_intern(rel_path);
    hash_file(rel_path, u->ent.file_hash);

    u->act.id = pfs_next_id();
    u->act.kind = ACT_WRITE;
    u->act.ts_ns = timespec_to_ns(&ctx->ts);

    u->ag.id = pfs_next_id();
    u->ag.uid = ctx->uid;
    u->ag.gid = ctx->gid;
    u->ag.pid = ctx->pid;
    u->ag.exe_offset = strings_intern(ctx->exe_path);

    u->rel_mask = REL_WAS_GEN_BY | REL_USED;

    u->delta_type = (uint8_t)delta->type;
    u->delta_offset = 0;
    u->delta_length = delta->length;
}
```

Listing 7.9: Provenance unit construction.

Listing 7.9 computes a content has, records the path and executable into the string table and sets the linkage to the delta buffer.

7.4.6 Block-based embedding

The embedded format follows the block design described in the previous section. It consists of a header block, followed by rows comprising a bitmap block and a sequence of data, version, and provenance blocks, all of fixed size. In the current implementation, the block size is set to $BLOCK_SIZE = 4096$ bytes,

chosen to match the typical page size and the block size assumed by the Linux kernel. However, the format and implementation are agnostic to the specific block size used. *BLOCK_SIZE* is a configurable parameter, and no structural assumptions depend on its concrete value beyond uniformity across blocks. Let the block size be defined as:

$$B = 4096$$

The address of block i is given by:

$$A_i = i \times B$$

Every block begins with a 12 bytes long block header (as defined in Listing 7.10) and the structure is padded to a fixed size. In this design, pointer blocks reserve 16 bytes at the front for metadata and alignment. That leaves the pointer capacity per block to be:

$$P = \left\lfloor \frac{B - 16}{8} \right\rfloor = \left\lfloor \frac{4096 - 16}{8} \right\rfloor = 510$$

The maximum addressable data per pointer block is:

$$D_{\max} = P \times B = 510 \times 4096 = 2,088,960 \text{ bytes}$$

The bitmap offset recurrence is:

$$\begin{aligned} \text{BitmapOffset}_n &= n \times (8 \times B + B) \\ &= n \times (8 \times 4096 + 4096) \\ &= n \times 134{,}221{,}824 \end{aligned}$$

The on-disk layout is as follows:

- Block 0: global bitmap for the first row.
- Block 1: header block.
- Block 2: metadata block.
- Subsequent blocks: data pointer blocks, data blocks, version blocks, provenance blocks, further bitmaps.

Each block begins with a small header containing the previous and next block indices and a type tag,

```
#define BLOCK_SIZE 4096
typedef uint32_t block_index_t;

typedef enum {
    BLK_FREE        = 0,
    BLK_HEADER      = 1,
    BLK_BITMAP      = 2,
    BLK_META        = 3,
    BLK_DATA_PTR    = 4,
    BLK_DATA        = 5,
    BLK_VERSION     = 6,
    BLK_PROVENANCE  = 7,
    BLK_STRINGS     = 8,
} block_type_t;

#pragma pack(push, 1)
struct block_hdr {
    block_index_t next;
    block_index_t prev;
    uint16_t type;
    uint16_t reserved;
    uint8_t pad[BLOCK_SIZE - 12];
};
#pragma pack(pop)
```

Listing 7.10: Block header definition.

Listing 7.10 defines the generic block header and the enumeration of block types. This header appears at the start of every block and allows the explorer to traverse typed block lists without extra indexes.

```
struct header_block {
    char magic[8];
    uint32_t version;
    block_index_t data_head, data_tail;
    block_index_t ver_head, ver_tail;
    block_index_t pfs_head, pfs_tail;
    block_index_t str_head, str_tail;
    uint64_t created_ns;
    uint64_t last_change_ns;
    uint8_t reserved[BLOCK_SIZE - 48];
};
```

Listing 7.11: Header block.

The header block holds global metadata for the container: version, timestamps and the head and tail pointers for the various block lists. This provides an entry point for traversing data, version, provenance and string blocks. Listing 7.11 shows the layout of the header block. It is placed at a fixed location and read at mount time by both the capture layer and the explorer.

```
function alloc_block(type):
    for each bitmap_row in file:
        load bitmap bytes into memory
        for bit_index from 0 to N-1:
            if bit is 0 (free):
                set bit to 1
                write bitmap back
                idx := row_start_block + 1 + bit_index
                init block header at idx with type
                if type participates in a list:
                    link into head/tail in header_block
                return idx
    append new row at end of file and repeat
```

Listing 7.12: Pseudocode for Block Allocation.

```
static block_index_t alloc_block(block_type_t type)
{
    ...
    struct bitmap bm;
    block_index_t row_start;
    if (!bitmap_find_free(&bm, &row_start))
        bitmap_extend(&bm, &row_start);

    int bit = bitmap_alloc_bit(&bm);
    block_index_t idx = row_start + 1 + bit;

    struct block_hdr hdr = {0};
    hdr.type = type;
    hdr.prev = 0;
    hdr.next = 0;
    pwrite(global_fd, &hdr, sizeof(hdr), (off_t)idx *
   BLOCK_SIZE);

    return idx;
}
```

Listing 7.13: C Implementation of Block Allocation.

Listing 7.12 presents the pseudocode of the block allocation algorithm and Listing 7.13 details the core data structure for the allocator. The procedure allocates a new storage block from a bitmap and initialises its metadata. It targets storage systems that represent free/used block state via a bitset, organised in consecutive bitmap rows. Each bit corresponds to one data block: a value of 0 denotes a free block, and 1 denotes a block in use. Beyond marking allocation, the procedure optionally links the newly allocated block into a type-specific list within a header block, supporting fast enumeration of blocks of a particular class. This implementation balances space efficiency (via bitmaps) with operational efficiency (via typed lists), providing $O(1)$ write per allocation and O(number of bits scanned) read in the common case.

```
#define DATA_PTRS_PER_BLOCK ((BLOCK_SIZE - sizeof(struct
    block_hdr)) / sizeof(block_index_t))

struct data_ptr_block {
    struct block_hdr hdr;
    block_index_t ptrs[DATA_PTRS_PER_BLOCK];
};
```

Listing 7.14: Pointer block structure.

Data is not stored directly in a contiguous linked list because that introduces high overhead for random access. Instead, pointer blocks hold arrays of pointers to 4 KiB data blocks, as described in Listing 7.15.

```
struct version_block {
    struct block_hdr hdr;
    uint32_t used;
    uint8_t data[BLOCK_SIZE - sizeof(struct block_hdr) - sizeof
    (uint32_t)];
};
```

Listing 7.15: Version block.

Version and provenance information is stored in separate block lists. A version block contains a contiguous region of delta bytes and a counter of bytes used. A provenance block contains several fixed-size provenance units. This decouples provenance structures from the raw delta stream and allows efficient iteration over either. Listing 7.15 defines the version block structure.

Deltas are appended sequentially into the data region, and used tracks the current fill level.

```
#define PROVENANCE_UNITS_PER_BLOCK ((BLOCK_SIZE - sizeof(struct
    block_hdr)) / sizeof(struct pfs_unit))

struct provenance_block {
    struct block_hdr hdr;
    struct provenance_unit units[PROVENANCE_UNITS_PER_BLOCK];
};
```

Listing 7.16: Provenance block.

Similarly for the provenance block layout, the number of *provenance_unit* entries per block is computed from the block size and the unit size, as in Listing 7.16.

```
void embed_append_unit(const char *rel_path, struct pfs_unit *u
    )
{
    block_index_t vblk_idx = ver_tail();
    struct version_block vblk;

    pread(global_fd, &vblk, sizeof(vblk), (off_t)vblk_idx *
    BLOCK_SIZE);
    if (vblk.used + u->delta_length > sizeof(vblk.data)) {
        vblk_idx = alloc_block(BLK_VERSION);
        memset(&vblk, 0, sizeof(vblk));
        vblk.hdr.type = BLK_VERSION;
        vblk.used = 0;
        link_new_version_block(vblk_idx);
    }

    off_t delta_off = (off_t)vblk_idx * BLOCK_SIZE +
                      sizeof(struct block_hdr) + sizeof(
    uint32_t) +
                      vblk.used;

    pwrite(global_fd, current_delta_buffer.data, u->
    delta_length, delta_off);
    vblk.used += u->delta_length;
    pwrite(global_fd, &vblk, sizeof(vblk), (off_t)vblk_idx *
    BLOCK_SIZE);

    u->delta_offset = (uint64_t)delta_off;

```

```
    block_index_t pblk_idx = pfs_tail();
    struct pfs_block pblk;

    pread(global_fd, &pblk, sizeof(pblk), (off_t)pblk_idx *
   BLOCK_SIZE);

    size_t idx = find_free_pfs_slot(&pblk);
    if (idx == (size_t)-1) {
        pblk_idx = alloc_block(BLK_PROVENANCE);
        memset(&pblk, 0, sizeof(pblk));
        pblk.hdr.type = BLK_PROVENANCE;
        link_new_pfs_block(pblk_idx);
        idx = 0;
    }

    pblk.units[idx] = *u;
    pwrite(global_fd, &pblk, sizeof(pblk), (off_t)pblk_idx *
   BLOCK_SIZE);
}
```

Listing 7.17: Appending provenance and delta.

When the capture layer records a change, it has a *prov_unit* plus a *delta_buffer*. The embedding function first appends the delta bytes to the tail of the version list, then appends the provenance unit to the tail of the provenance list. It also updates the unit to point at the byte offset of the delta in the version stream. This is the operation that binds provenance and embedded deltas. Listing 7.17 shows *embed_append_unit*. It takes care of opening or extending the tail version block, writing the delta bytes, updating the used counter, setting the *delta_offset* in the provenance unit and placing the unit into a provenance block. Here *current_delta_buffer* is the *struct delta_buffer* prepared by the capture layer. The header block's *ver_tail* and *pfs_tail* pointers are updated as new blocks are linked, allowing the explorer to traverse the lists without scanning the entire file.

7.4.7 Provenance Access Interface

```
int embed_open(const char *path, struct embed_handle *out);
int embed_iter_units(struct embed_handle *h,
                     int (*cb)(const struct pfs_unit *, void *)
    ,
                     void *user);
int embed_read_delta(struct embed_handle *h,
                     const struct pfs_unit *u,
                     struct delta_buffer *out);
int embed_read_data_at(struct embed_handle *h,
                       off_t offset, size_t len,
                       uint8_t *buf);
```

Listing 7.18: Embed interface.

The provenance access interface abstracts over the block format and exposes a small API for iterating provenance units, reading deltas and logical file content. This library is used by both internal components and by the external explorer tool. Listing 7.18 shows the main functions exported by *libembed*:

- *embed_open* initialises an *embed_handle* for a container file.
- *embed_iter_units* walks all provenance units, invoking a callback for each.
- *embed_read_delta* retrieves the delta bytes associated with a particular provenance unit.
- *embed_read_data_at* reads logical content from the data section.

7.4.8 Provenance Explorer

```
function reconstruct_to_entity(handle, entity_id):
    units := list of pfs_unit
    u := find_unit_by_entity_id(handle, entity_id)
    while u has a predecessor:
        prepend u to units
        u := load predecessor unit

    base := read_data_from_disk_at_creation(u0)

    content := base
    for each unit in units:
```

```
        delta_buf := embed_read_delta(handle, unit)
        alg := delta_find_by_type(unit.delta_type)
        content := alg.apply(content, delta_buf)

    return content
```

Listing 7.19: Pseudocode for data reconstruction.

```
int explorer_reconstruct(struct embed_handle *h,
                         pfs_id_t target_eid,
                         uint8_t **out_buf, size_t *out_len)
{
    struct pfs_unit *chain = NULL;
    size_t chain_len = 0;

    if (build_pfs_chain(h, target_eid, &chain, &chain_len) !=
    0)
        return -1;

    uint8_t *content = NULL;
    size_t   content_len = 0;

    if (load_base_content(h, chain[0].ent.id, &content, &
    content_len) != 0) {
        free(chain);
        return -1;
    }

    for (size_t i = 0; i < chain_len; ++i) {
        struct delta_buffer delta = {0};
        if (embed_read_delta(h, &chain[i], &delta) != 0) {
            free(content);
            free(chain);
            return -1;
        }
        const struct delta_algorithm *alg = delta_find_by_type(
    delta.type);
        uint8_t *next = NULL;
        size_t   next_len = 0;
        if (alg->apply(content, content_len, &delta, &next, &
    next_len) != 0) {
            delta_free(&delta);
            free(content);
            free(chain);
            return -1;
        }
        delta_free(&delta);
```

```
36          free(content);
37          content = next;
38          content_len = next_len;
39      }
40
41      *out_buf = content;
42      *out_len = content_len;
43      free(chain);
44      return 0;
45  }
```

Listing 7.20: Data reconstruction implementation.

Reconstruction follows the high level process described in the Provenance Explorer section: select a target entity, collect the deltas along its ancestry, load the deltas from the version stream and replay them to materialise the content. The reconstruction algorithm to replay contents is described in Listing 7.19 and its C implementation in Listing 7.20. The function *reconstruct_to_entity* reconstructs the complete content of an entity by combining its base content with a sequence of delta units. It begins by locating the unit associated with the given entity identifier and then traverses its predecessor chain, collecting all units in chronological order. Once the earliest unit is identified, the function reads the base data from disk as it existed at the time of that unit's creation. This base serves as the initial state for reconstruction. The function then iterates through the collected units, reading each delta buffer and selecting the appropriate algorithm based on the delta type. Each algorithm applies its delta to the current content, progressively transforming it until all deltas have been processed. The final result, representing the reconstructed entity content, is then returned.

The explorer's validation mode uses the same traversal facilities to check that (i) all header pointers resolve to existing blocks, (ii) each version and provenance block forms a coherent list and (iii) the deltas associated with an entity can be replayed to a content whose hash matches the hash stored in *pfs_entity.file_hash*. When running in an interactive mode, *pfsexplorer* renders a textual account of the file's evolution, but grounded in the atomic provenance

units and delta sizes defined in this thesis.

7.4.9 Demonstration of Provenance Explorer

Having introduced the Provenance Access Interface and the Provenance Explorer, this section demonstrates their use in practice. The aim is to show how the abstract structures described earlier can be inspected, traversed, and validated using a concrete tool.

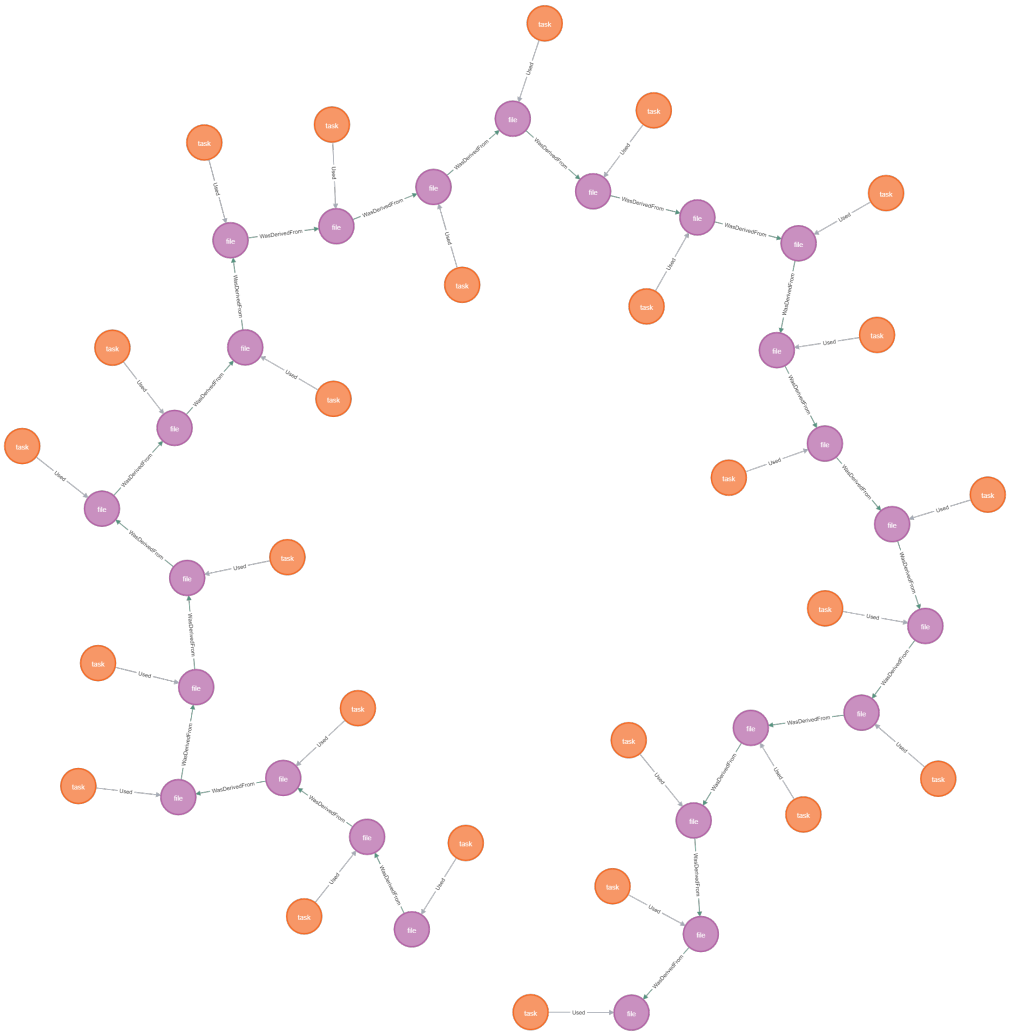


Figure 7.7: Graph view reconstructed by the provenance explorer from embedded provenance records.

Figure 7.7 - Figure 7.9 show the provenance explorer applied to a concrete compilation workflow. In this example, a C program is compiled using *gcc*, producing the output binary *a.out*. The explorer presents the embedded

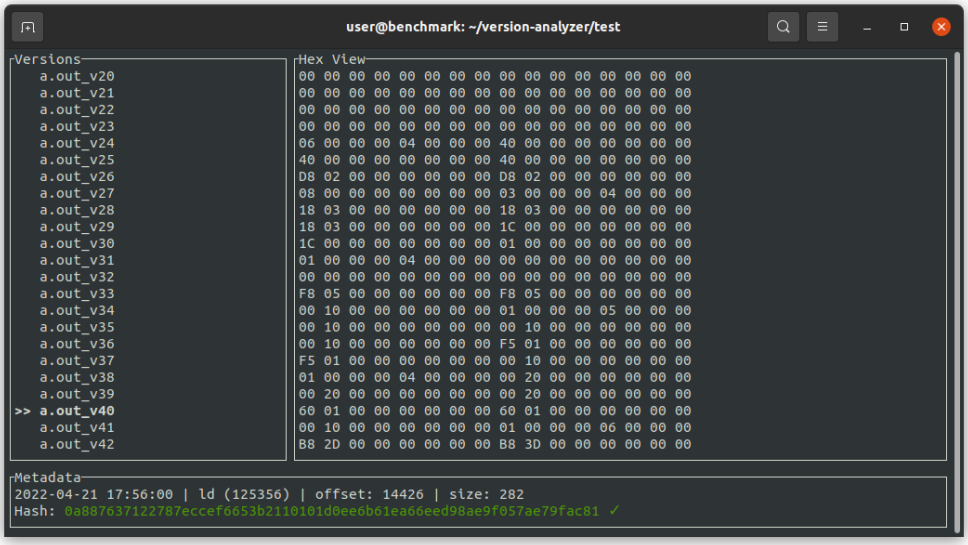


Figure 7.8: Low-level inspection view of a file with embedded provenance at version 40.

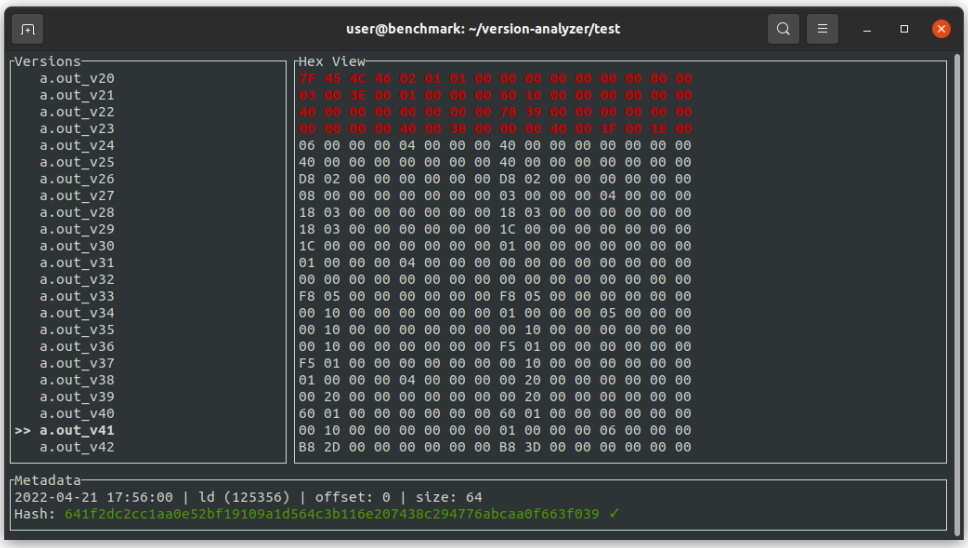


Figure 7.9: Low-level inspection view of a file with embedded provenance at version 41.

provenance captured during this process, exposing how the final executable was derived through successive compilation stages.

Figure 7.7 presents the explorer's graph-oriented view. Using only the access interface, the tool traverses the embedded provenance blocks and reconstructs the provenance graph incrementally. Files, activities, and processes are rendered as nodes, while edges represent the causal and temporal relations encoded in the atomic provenance units. This visualisation is generated lazily: nodes and edges appear as the underlying blocks are resolved, illustrating how provenance can be explored without precomputing or externalising the full graph. The figure demonstrates that fine-grained change events and their ordering are recoverable directly from the file, supporting lineage inspection and reasoning about evolution over time.

Figure 7.8 and Figure 7.9 focus on versions *a.out* at version 40 and *a.out* at version 41. These correspond to consecutive states of the output binary generated during the final linking phase of compilation. The highlighted regions in red indicate the delta between the two versions, that is, the precise byte-level changes introduced when transitioning from version 40 to version 41. This delta is not inferred externally but recovered directly from the embedded version and provenance blocks using the Provenance Access Interface.

The metadata panel associates this change with a specific execution context. In this case, the modifying activity was performed by process identifier 125356, and the responsible executable is *ld*, the system linker invoked by *gcc*. This makes the provenance of the compilation explicit. The explorer does not merely show that the file changed, but records which tool performed the transformation and when it occurred, linking the resulting binary state to the concrete compilation step that produced it.

The displayed metadata includes timestamps, byte offsets, sizes, and hashes that allow the explorer to locate, reconstruct, and validate the corresponding file state. It should be noted that the information presented in the screenshots is intentionally partial. The underlying system records a richer set of

provenance attributes, including full agent context, complete delta streams, and additional integrity and linkage metadata. Only a subset is shown here for clarity and illustrative purposes.

These figures demonstrate that the prototype is capable of capturing and exposing provenance at the level of real system activity. The compilation of a C program into an executable is treated as a sequence of provenance-bearing transformations, each embedded within the artefact itself. The provenance explorer makes this history directly inspectable, confirming that the approach supports practical, low-level workflows such as compilation while remaining self-contained within the produced file.

Taken together, these three figures demonstrate how the explorer bridges the gap between embedded representation and user-level analysis. The tool confirms that provenance embedded at the storage level can be reconstructed into meaningful structures on demand, supporting both high-level lineage exploration and low-level validation using the same underlying access interface. This serves as a practical demonstration of the feasibility and usefulness of the design presented in this chapter.

7.5 End-to-End Prototype Workflow

The preceding sections have described the individual components, data structures, algorithms and implementation mechanisms of the proposed prototype. This section brings these elements together to illustrate how they interact during a typical filesystem operation and how the resulting provenance information is subsequently accessed and interpreted.

As illustrated in Figure 7.10, the prototype operates through two complementary workflows: provenance capture and provenance exploration. The capture workflow is responsible for observing filesystem operations, identifying changes, constructing provenance records and embedding them within files. The exploration workflow retrieves this information to support lineage analy-

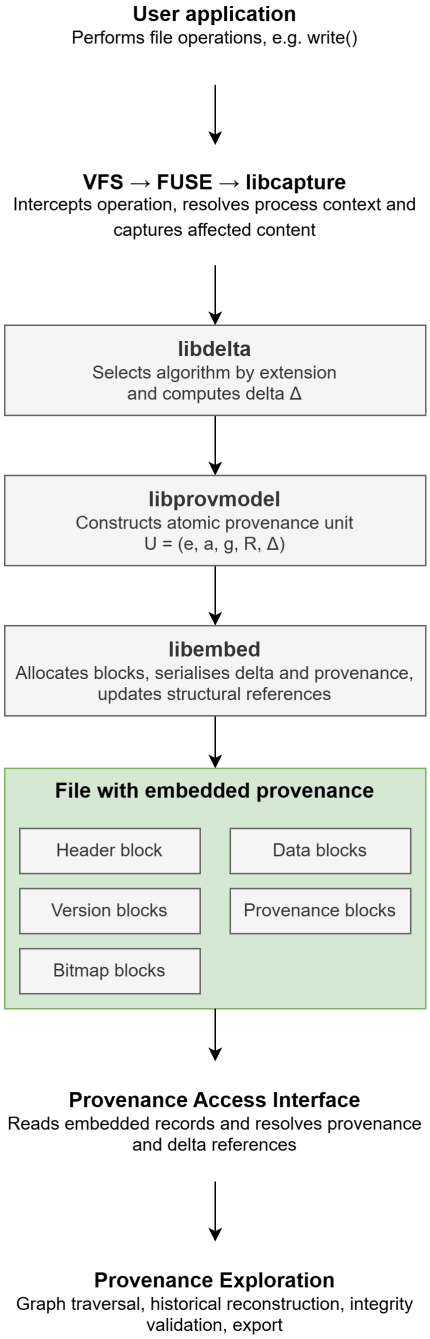


Figure 7.10: End-to-end prototype workflow.

sis, historical reconstruction and integrity verification.

7.5.1 Provenance Capture Workflow

The workflow begins when a user application performs a filesystem operation, such as creating, reading, modifying or renaming a file within the mounted provenance aware filesystem. The request passes through the Virtual File System (*VFS*) layer and is forwarded to the FUSE daemon (*pfsd*), where the corresponding callback defined in the *fuse_operations* structure is invoked.

The capture component (*libcapture*) first resolves the execution context using the mechanism illustrated in Listing 7.2. This identifies the process responsible for the operation, including its process identifier, user and group identifiers, executable path and timestamp. These attributes provide the contextual information required to associate the observed activity with an agent in the provenance model.

For operations that modify file content, such as *write()*, the handler captures the relevant previous content before forwarding the modification to the underlying filesystem. As demonstrated in Listing 7.3, the write handler uses *pread()* to obtain the bytes that are about to be overwritten and *pwrite()* to perform the requested modification.

Once the operation has been performed, the delta extraction component (*libdelta*) determines the difference between the previous and resulting content. The algorithm registry described in Listing 7.5 selects an appropriate differencing algorithm based on the file extension. If no specialised algorithm is registered, the generic byte level algorithm is used. The resulting delta describes the modification in a structured form suitable for storage and subsequent replay.

The provenance model component (*libprovmodel*) then represents the observed operation as an atomic provenance unit, defined as

$$U = (e, a, g, R, \Delta),$$

where e denotes the resulting entity, a the activity, g the responsible agent, R the applicable *PROV-DM* relationships and Δ the associated delta. The relationship is determined according to the operation mapping presented in Table 7.1. For example, a write operation is associated with *wasGeneratedBy*, while a read operation is represented using *used*.

The resulting provenance unit and its associated delta are passed to the embedding component (*libembed*). This component manages the block based storage structure described in Section 7.3.3. Delta bytes are stored in version blocks, while provenance records are stored separately in provenance blocks. Each provenance record maintains references to the corresponding delta, allowing the semantic description of an operation to be associated with the content level change it produced. Bitmap blocks support allocation, while the header maintains the structural references required to locate and navigate the embedded information.

Through this process, the file retains both its current content and the provenance information describing its evolution. Operations that do not modify content, such as reads, contribute provenance relationships without requiring the generation of content deltas.

7.5.2 Provenance Exploration and Reconstruction Workflow

The second workflow begins when the provenance explorer (*pfsexplorer*) is used to inspect a file containing embedded provenance. The explorer accesses the stored information through the provenance access interface, which abstracts the underlying block based representation.

The process begins by reading the header block to locate the relevant data, version and provenance blocks. The explorer then traverses the provenance records, resolving their references to reconstruct the relationships between entities, activities and agents.

When a user requests a historical version, the explorer identifies the corresponding entity and determines the sequence of deltas required for reconstruction. These deltas are retrieved from the version blocks using the offsets recorded in the provenance entries and applied in the appropriate order using the relevant delta algorithm.

The explorer also supports integrity verification by examining structural pointers, block consistency, provenance references and stored checksums or hashes. The reconstructed provenance graph can subsequently be inspected or exported for analysis using external provenance tools.

7.5.3 Integration of the Components

Together, these workflows demonstrate how the prototype's components form a coordinated mechanism rather than a collection of independent functions. The FUSE layer provides transparent interception of filesystem operations, the delta extraction component identifies content level changes, the provenance model assigns semantic meaning to those changes, and the embedding component stores the resulting records directly within the file. The provenance access interface and explorer complete the process by making the stored information available for querying, reconstruction and verification.

This integration demonstrates how temporal awareness and embedded provenance can operate within a single prototype while preserving the modularity of the individual components.

7.6 Conclusion

This chapter has presented a working prototype that integrates temporal awareness with embedded provenance, translating the conceptual arguments of the preceding chapters into a concrete system. By treating provenance as an integral component of the file, and by representing change through atomic, delta encoded units, the design demonstrates that fine grained lineage can be

captured without relying on external provenance stores or application specific instrumentation.

The architecture shows how versioning, and embedding provenance can be combined in a coherent and extensible manner. The use of a block based container format enables provenance and data to evolve together, while preserving reconstructability and auditability from any historical point. The provenance explorer further illustrates that these records are not merely archival, but can be interrogated, validated, and replayed in a deterministic way, grounding provenance in observable file states rather than abstract graph structure.

The implementation choices, including the use of FUSE and user space components, prioritise clarity and feasibility over optimisation. As such, the prototype is not intended as a production ready system, but as evidence that the proposed design principles are practical and internally consistent. Limitations relating to performance overheads, kernel integration, and coverage of transient system state remain and are addressed in the next chapter; these arise from the scope of the implementation rather than from the underlying model.

In addressing RQ5, this chapter demonstrates that temporally aware, embedded provenance can support lineage tracing while preserving audit guarantees. The next chapter evaluates this claim, examining the correctness, performance characteristics, and practical limits of the prototype under controlled workloads.

Chapter 8

Evaluation

This chapter evaluates the prototype developed in the previous chapter, and in particular its alignment with the objectives of this thesis: to capture provenance, to embed that provenance inside the file it describes, to preserve consistency across the block based format, and to support deterministic replay and validation through the explorer. The evaluation examines correctness, reconstructability, performance, robustness and limitations. The focus is on demonstrating feasibility rather than achieving production grade metrics, as the prototype is intended to validate architectural principles rather than deliver an optimised system.

The evaluation is structured around a set of practical experiments performed against the implementation. These experiments use controlled workloads, synthetic test files and representative user level operations executed through the FUSE capture layer. The explorer tool provides the means to inspect internal structures, verify block level invariants and reconstruct content. Each experiment addresses a particular design requirement introduced earlier in the thesis: correctness of provenance recording, integrity of the embedded format, interpretability of provenance units, reconstructability of historical file states and quantifiable operational overheads.

8.1 Setup & Methodology

All of the benchmarking experiments described in this chapter were conducted on a dedicated machine configured to ensure consistency and reproducibility of results. The system used was an Intel Core i7-8850H processor, a six-core, twelve-thread CPU with a base clock of 2.6 GHz. The system was equipped with 8GB of RAM and storage was provided by a 250GB Samsung 970 NVMe drive, operating over a PCIe 3.0 $\times 4$ interface. The drive includes 512MB of dedicated LPDDR4 DRAM cache. The manufacturer rated sequential read speed is up to 3,400 MB/s, while the sequential write speed is up to 1,500 MB/s. The operating system used was Ubuntu 20.04.6 LTS. The underlying filesystem used for all benchmarking experiments is ext4 [209].

The benchmarks used in this evaluation were selected with reference to established evaluation practices in filesystem and provenance research. Previous provenance systems, including PASS [210] and PASSv2 [2], have used Linux kernel compilation to evaluate execution time overhead, storage consumption and provenance capture. Similarly, CamFlow [8] has employed *lmbench* to assess the overhead of provenance collection at the system call level. Following these approaches, *IOzone* was selected to characterise filesystem throughput under controlled I/O workloads, while Linux kernel compilation was used to evaluate provenance capture under a realistic, multi-process workload involving numerous files and complex dependencies. Together, these benchmarks allow the prototype to be evaluated at both the filesystem and application levels, while providing methodological continuity with related research.

8.1.1 Controlling Cache Effects

To ensure the validity and repeatability of filesystem benchmark results, it is necessary to clear the operating system's caches before conducting measurements, particularly for read or mixed read-write workloads. On Linux systems, this is achieved by writing the value 3 to the *drop_caches* interface [211], like

so: *sync && echo 3 |sudo tee /proc/sys/vm/drop_caches*

This command instructs the Linux kernel to drop all page caches, dentries, and inode caches. Page cache refers to the in-memory caching of file contents, while dentries and inodes cache filesystem metadata such as directory entries and file attributes. By flushing these caches, the system ensures that subsequent filesystem operations reflect genuine I/O activity on the underlying storage device, rather than being satisfied from RAM.

This consideration is important for read heavy benchmarks. Without clearing the operating system's page cache, repeated read operations may be satisfied directly from system memory, producing throughput measurements that do not accurately reflect the cost of accessing the underlying storage device. Clearing these caches reduces this effect by requiring subsequent reads to traverse the storage I/O path. However, this does not eliminate caching at the hardware level. Modern NVMe SSDs employ internal controller caches, flash translation layer metadata caches, and other optimisation mechanisms that remain active independently of the operating system. Consequently, the resulting measurements should be interpreted as filesystem performance with reduced operating system cache influence, rather than as measurements of completely uncached physical storage access.

8.2 Micro-benchmark

The evaluation does not attempt to compare this system to full provenance frameworks such as CamFlow [8], PASS [210] or SPADE [4], because these operate at different layers and capture different abstractions. The criteria here are internal: does this architecture behave as designed, and can it be implemented with quantifiable cost? For example, CamFlow uses *lmbench* to assess overhead. While *lmbench* is a well-established microbenchmark suite, its focus is on low-level operating system behaviour rather than realistic application workloads. It measures *syscall* level costs such as file operation latency (*stat*,

open, *close*, *read*, *write*), metadata overhead, and context switch or IPC effects that influence FUSE paths. Running these tests on a FUSE mount exercises the kernel-user space boundary, so results reflect FUSE overhead. However, *lmbench* does not model large sequential or random I/O patterns, sustained bandwidth under load, multi-threaded or application-level access patterns, nor does it isolate FUSE overhead from page cache effects without careful control. Consequently, while *lmbench* provides insight into micro-level performance, it does not capture the broader implications of embedding provenance within files, and any comparison based on such metrics would be misleading given the differing operational scope and objectives of these frameworks.

8.2.1 IOzone

IOzone is a benchmark tool that evaluates file system I/O performance across a wide range of operations, including sequential, random, and mixed access patterns. It generates detailed performance metrics by simulating diverse workloads and testing different record sizes, making it valuable for diagnosing bottlenecks or comparing storage systems.

We ran IOzone using the following command:

iozone -a -R -r 4k -r 64k -r 1m -r 8m -s 8G -f <file>

- *-a* enables *automatic* mode, benchmarking a range of filesystem read write operations. Full definition of these modes can be found in IOZone Manual [212].
- *-R* produces output in Excel-friendly format for easier parsing.
- *-r 4k -r 64k -r 1m -r 8m* sets multiple record sizes to test: 4 KiB, 64 KiB, 1 MB, and 8 MB. IOZone will test each of these.
- *-s 8G* sets the test file size to 8 GB, ensuring testing well beyond typical RAM size to avoid purely cache-based results.
- *-f <file>* specifies the test file path.

8.2.2 Cache Effects

Typically, the *-I* flag in IOzone will be used to bypass the operating system's buffer cache by enabling direct I/O, which corresponds to the *O_DIRECT* flag in *open* system calls. However, this approach encounters fundamental limitations when applied to any FUSE-based file system.

FUSE operates in user space but depends on a kernel module to mediate interactions between the VFS layer and the user space daemon responsible for implementing the file system's logic. Crucially, many FUSE-based file systems do not support the *O_DIRECT* flag at all. The FUSE protocol itself provides only limited and optional support for direct I/O, and even where such support is available, it is often inconsistently implemented. For example, the *passthrough.c* example technically includes support for *O_DIRECT*, but this support may not function correctly in practice. Correct operation requires that the underlying file system itself supports direct I/O and that alignment requirements, such as block size constraints, are met. Some FUSE file systems will simply ignore the *O_DIRECT* flag altogether or silently fall back to buffered I/O if they are unable to handle direct I/O natively.

As a result, using the *-I* option with IOZone against FUSE-based file systems is either impossible, because it is not supported, or else invalid, because the implementation silently falls back to buffered I/O, defeating the purpose of attempting to measure raw storage-layer performance. This limitation has significant implications for benchmarking, because without bypassing the page cache, any measurements will inevitably include the effects of read-ahead, write-behind, kernel caching policies, and system memory pressure. These factors can considerably inflate apparent performance compared to true direct-access storage measurements.

The results presented here will therefore reflect the influence of the kernel page cache. It is essential to recognise this when interpreting the results. A direct performance comparison between these FUSE-based systems and the non-FUSE system (which does support -I) would be invalid unless the non-FUSE

system is also benchmarked without -I, ensuring methodological consistency.

Record Size (KiB)	Write Gain	Read Gain	Random Read Gain	Random Write Gain
4	8.16×	16.66×	51.16×	8.52×
64	3.70×	14.36×	40.58×	17.65×
1024	4.00×	9.60×	10.05×	8.40×
8192	5.80×	4.50×	4.94×	19.84×

Table 8.1: Performance gain (buffered vs. direct I/O throughput) for selected operations.

Accordingly, we will first present benchmarks of the underlying system both with and without *-I*, to illustrate the performance implications of bypassing the page cache. All subsequent benchmarks will be conducted without *-I* to maintain parity across all evaluated systems.

Table 8.1 summarises the relative performance gain achieved by using the operating system's page cache, by presenting the ratio of buffered I/O throughput to direct I/O throughput. The ratio indicates how many times faster buffered I/O is for each operation and record size tested.

These results make clear the impact of the page cache on filesystem throughput. For the smallest record length of 4 KiB, write throughput in buffered mode is over eight times greater than in direct I/O, while read throughput shows a sixteen-fold increase. Random read operations at 4 KiB are particularly affected, with a fifty-fold increase in throughput when the page cache is enabled. Random writes similarly benefit by over eight times.

As the record sizes increases to 64 KiB and 1024 KiB, the effect of caching remains strong but diminishes somewhat for sequential operations. Write performance is roughly four times faster with the page cache, and read performance is around ten times higher. Random read and write operations continue to exhibit large differences. Random reads remain approximately ten times faster with caching, while random writes are over eight times faster.

At the largest block size of 8192 KiB, the gap for sequential operations narrows further. Buffered writes are nearly six times faster than direct writes,

while reads are about four and a half times faster. However, random writes at this size remain highly optimised by the cache, showing almost twenty-fold throughput improvement over direct I/O. Random reads also benefit considerably, with buffered I/O achieving nearly five times the performance.

The random write results exhibit variations in performance across record sizes. In particular, the 64 KiB record size achieves a buffered-to-direct I/O throughput ratio of 17.65×, substantially exceeding the corresponding ratios of 8.52× and 8.40× observed at 4 KiB and 1024 KiB, respectively. A similarly elevated ratio of 19.84× is observed at 8192 KiB.

One possible explanation is the interaction between the Linux page cache and asynchronous writeback mechanisms. Buffered random writes can be acknowledged after data has been copied into the page cache, with subsequent persistence handled by the kernel's writeback subsystem. This allows buffered I/O to benefit from write aggregation, delayed writeback and reduced immediate dependence on storage-device latency. Direct I/O bypasses the filesystem page cache for file data and is therefore more directly exposed to the characteristics of the underlying NVMe storage path.

The variation may also reflect differences in I/O request aggregation, SSD internal write management behaviour and system conditions during individual benchmark runs. In particular, the relationship between record size and throughput is not necessarily linear, since different record sizes can interact differently with the operating system's buffering and the storage device's internal scheduling mechanisms.

8.2.3 FUSE Overhead

It is a well-established fact in the literature that FUSE-based filesystems typically incur substantial overheads compared to in-kernel filesystems [213, 214]. The primary reason for this is because each filesystem call that traverses the FUSE layer requires multiple context switches between user space and kernel space.

Our initial objective is to characterise and quantify the baseline overhead that FUSE introduces in our experimental setup. This step is critical because, in the remainder of the evaluation, we aim to compare our implementations against this baseline. We isolate and account for the performance penalties that are inherent to the FUSE framework itself, so that any further overheads observed can be more accurately attributed to the specific design or implementation choices of our prototype.

Record Size (KiB)	**write**	**rewrite**	**read**	**reread**	**random read**
4	1,209,900 (80.4%)	2,838,111 (87.8%)	3,733,902 (90.3%)	3,765,295 (90.5%)	2,204,928 (91.1%)
64	1,512,859 (75.5%)	2,309,823 (63.1%)	8,833,798 (77.9%)	6,072,362 (71.6%)	8,833,534 (87.2%)
1024	1,488,995 (75.5%)	2,139,120 (59.2%)	8,266,407 (70.1%)	5,214,865 (60.6%)	8,957,369 (74.8%)
8192	1,328,469 (72.8%)	2,169,454 (60.0%)	6,287,532 (65.3%)	4,276,392 (57.5%)	6,437,434 (65.5%)

Table 8.2: Quantified FUSE overhead relative to *ext4* baseline across record sizes (KiB/s and %).

For small record sizes (4 KiB), FUSE throughput is typically around 25–30% of *ext4*. That suggests the overhead here is largely in user-kernel space transitions, context switches, and the extra copying required for FUSE operations.

As the record size increases, the absolute throughput of FUSE improves, but the relative gap with *ext4* remains severe. For 64–8192 KiB records, *ext4* routinely exceeds 10 GB/s in aggregate read-heavy workloads, while FUSE tops out around 3 GB/s, suggesting that even for very large, sequential operations the cost of FUSE context-switching and kernel-user space round-trips cannot be fully amortised.

Interestingly, writes under FUSE appear somewhat less catastrophic at large block sizes than reads (still ∼2–4 GB/s), but the relative overhead is unambiguously significant. Even the best-case scenarios show about 50–60%

lower throughput than *ext4*. Also notable is that random I/O is equally penalised. Random reads and writes under FUSE show roughly linear scaling with record size but are still significantly lower than *ext4*. For example, at 1024 KiB reclen, random read throughput is ∼12 GB/s for *ext4* vs ∼3 GB/s for FUSE, an overhead of about 75%.

8.2.4 Throughput Analysis

Filesystem	Write	Rewrite	Read	Reread
ext4	1,826,426	3,614,567	9,631,305	7,435,931
FUSE Passthrough	497,957	1,445,113	3,343,773	3,159,539
Version on Close	554,067	945,131	2,755,409	2,651,899
Version on Write	505,127	503,768	2,722,736	2,623,134

Table 8.3: Average throughput (KiB/s).

Table 8.3 summarises average throughput values (KiBytes/sec) across key workloads for the four filesystems at large block sizes (8192 KiB). This block size approximates large file transfer workloads, where bandwidth typically dominates.

8.3 Simulating File Ageing

To understand how the prototype behaves under long running, organically evolved workloads, a controlled experiment was carried out in which a real file with a rich development history was replayed through the capture layer. The aim was to observe how the block-based container grows over time, how the provenance stream develops, whether delta extraction remains effective as content diverges and whether reconstruction times stay stable as the number of embedded versions increases.

In this experiment, we draw on source files from the Linux kernel. The kernel provides a rich environment for studying file evolution due to its long development history and its large contributor base. Continuous activity over decades has produced revision patterns that range from minor line edits to extensive subsystem-level changes. The file *fs/file.c* serves as the focal point of the experiment. It contains extended functions, shared interfaces and points of interaction with several subsystems. As these subsystems evolved, the file accumulated diverse forms of modification, including identifier changes, type substitutions, logic adjustments and interface migrations. It is acknowledged that this pattern of file changes does not exactly replicate all the file saves that would have occurred as developers locally worked on coding changes, its complex reversion history does make it a suitable case for evaluating delta extraction methods within the experimental framework. Local development workflows typically entail a high frequency of granular modifications. This behaviour demonstrates the relevance of the delta size calculations presented below, as they gauge efficiency in handling such incremental updates.

A small utility was written that used git calls to reconstruct the fs/file.c at various stages of its development. The implementation of this utility is described in full in Appendix B. In brief: the utility queries the Git history for the specified path and extracts, for each matching commit, the commit identifier and its author timestamp. The commits are ordered from the earliest to the most recent, yielding a linear sequence that represents the file's evolution over time. The script then iterates over this commit sequence and reconstructs the file as it existed at each commit. For every commit, it retrieves the file contents directly from the corresponding Git tree object, rather than from the working directory. This produces an exact snapshot of the file at that point in history, independent of later modifications.

At the end, the utility produced more than 200 versions of the file, ranging from early, compact versions to modern variants that differ significantly in structure and size. These were placed into a temporary directory for ingestion.

8.3.1 Replay through the capture layer

To simulate a long-lived file, all historical versions were fed into a fresh instance of the prototype file system, using a developed utility we called the replay tool. The approach was simple. An empty container file named *file.c* was created inside a mounted provenance volume. For each snapshot in chronological order, the replay tool opened the file, rewrote its entire content with the snapshot's bytes and closed the file. From the perspective of the prototype, each step was an ordinary write, captured by the FUSE layer, passed through delta extraction and embedded into the container as a new provenance unit and a corresponding delta entry. This ensures that the prototype's own delta algorithms perform all change detection work. In practice, the write pattern looks like:

1. Open file.
2. Empty all contents.
3. Write the new snapshot's full content.
4. Flush and close.

Because the content sizes vary and some versions perform structural refactors, the resulting deltas range from tiny single-line modifications to patches spanning the majority of the file. This diversity exercises both the byte-diff algorithm and the raw mode fallback.

8.3.2 File Growth

The main question to answer for the evaluation of growth is how the size of an embedded, delta-based history file evolves over time. Table 8.4 presents a snippet of the recorded data for versions 2 through 219, illustrating the storage characteristics observed during incremental versioning. For each version, the prototype captured four key metrics:

version	**file_size**	**delta_size**	**total_size**	**delta_sum**
2	5826	380	11479	380
3	9074	2516	20553	2896
4	8962	179	29515	3075
⋮	⋮	⋮	⋮	⋮
219	38791	305	5435228	87510

Table 8.4: Version data table.

- *file_size*: The size of the data at that step, representing the actual data footprint of the version.
- *delta_size*: The size of the computed delta between the current data and its immediate predecessor, indicating the efficiency of change based storage.
- *total_size*: The cumulative size of all data up to the current version, reflecting the hypothetical storage requirement if every version were retained as a complete copy.
- *delta_sum*: The cumulative size of all deltas up to the current version, representing the aggregate cost of storing only incremental changes.

These metrics enable a comparative analysis of two storage strategies: full data retention versus delta based versioning. The *total_size* metric represents a baseline in which each version is preserved as a full copy, leading to linear growth in storage consumption as versions accumulate. The Elephant File System [173] exemplifies such design: a file system that preserves historical versions of files using copy-on-write mechanisms and configurable retention policies. In this context, *total_size* can be interpreted as an upper-bound reference that captures the cost of retaining complete versions over time. This behaviour can be observed in applications such as *vim*, which write modifications by creating a new file and replacing the old one [6], effectively materialising each version as a full copy on disk. Under such a model, storage

consumption grows proportionally with the number of saved versions, making *total_size* a useful baseline against which more space-efficient versioning schemes can be evaluated. The complete dataset, including all versions, is provided in Appendix A for reference.

The *total_size* column represents a simple baseline: retain each of the 219 versions as a full snapshot, so the storage cost is the sum of all historical file sizes. For version 219 this sum is

$$\textit{total_size} = 5{,}435{,}228 \text{ bytes.}$$

If the file had remained at its final size throughout its history, this baseline would scale as $219 \times \textit{file_size}_{219}$. In practice the file grew over time, so earlier versions were smaller and the cumulative baseline is lower than $219 \times \textit{file_size}_{219}$. Equivalently, the baseline corresponds to an average historical snapshot size of

$$\overline{s} \approx \frac{\textit{total_size}}{219} \approx 24.8 \text{ KiB},$$

which is consistent with the measured mean of 24.9 KiB across versions.

The *delta_sum* column measures how much data the prototype actually writes into the version stream to encode this history. By version 219,

$$\textit{delta_sum} = 87{,}510 \text{ bytes},$$

so the full-snapshot baseline is about 62× larger than the accumulated deltas:

$$\frac{\textit{total_size}}{\textit{delta_sum}} \approx \frac{5{,}435{,}228}{87{,}510} \approx 62.$$

In other words, the embedded delta stream is roughly sixty-fold smaller than storing every snapshot in full. To get a sense of per-version behaviour, across all 218 steps in the data:

$$\textit{file_size} \in [5{,}826, 38{,}791] \text{ bytes}, \quad \text{mean} \approx 24.9 \text{ KiB}, \quad \text{median} \approx 24.6 \text{ KiB},$$

$$\textit{delta_size} \in [148, 2{,}631] \text{ bytes}, \quad \text{mean} \approx 401 \text{ bytes}, \quad \text{median} \approx 262 \text{ bytes}.$$

A typical delta is therefore under 2% of the corresponding data size. The history contains spikes where *delta_size* is in the kilobyte range, matching larger edits or refactors, but most changes are small.

If we consider the storage cost of “current file plus full history” in the embedded scheme, it consists of roughly *file_size* for the current content plus *delta_sum* for the version stream, plus provenance units and metadata. Ignoring the latter for the moment and using version 219 as a representative point:

$$\text{content} + \text{deltas} \approx 38{,}791 + 87{,}510 \approx 126{,}301 \text{ bytes}.$$

Compared to *total_size* = 5,435,228 bytes, this is about a forty-fold reduction over a full data retention store while still supporting reconstruction of every historical version.

The median *delta_size* of 262 bytes, against a median *file_size* of around 24.6 KiB, also shows that for most commits the prototype pays only a small marginal cost for keeping an exact change history. The cumulative *delta_sum* curve bends upwards slowly because of occasional large deltas, but the overall growth remains modest.

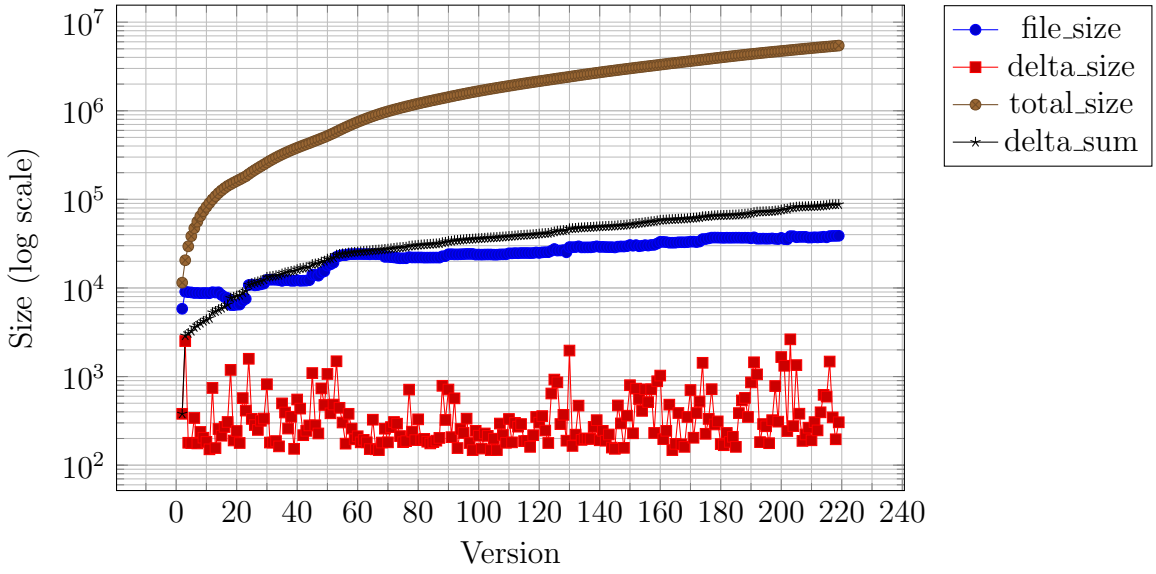


Figure 8.1: File size growth (log scale).

On the log scale, as illustrated in Figure 8.1, *total_size* forms a near-straight rising line. It increases by roughly one full data size each step, so the curve is close to linear in version count. It sits at the top of the plot because it is orders of magnitude larger than the other series.

file_size appears as a band below that line, oscillating as the file grows and shrinks. Early versions sit in the 5–10 KiB region, later versions move into the 30–40 KiB range. On a log scale this drift is visible as a gentle upward trend with local noise, reflecting the long-term growth of *file.c* and the step-like jumps around larger refactors.

delta_size appears as a noisy series much lower down. Most points cluster in the low hundreds of bytes, with occasional spikes into the kilobytes. On the log scale those spikes form narrow peaks: they correspond to more intrusive edits where the byte-diff algorithm cannot compress the change much and falls back to larger patches.

delta_sum appears as a smooth, shallow curve that remains well below *total_size* for all versions. It starts at 380 bytes at version 2 and reaches 87,510 bytes by version 219. On the log scale it rises slowly and never approaches the *total_size* line, which stays roughly two orders of magnitude above it. The vertical gap between the *total_size* and *delta_sum* curves at the right edge of the plot visually encodes the compression factor discussed above.

Overall, the graph summarises three main points:

1. The cost of keeping every historical version as a full snapshot grows with the age of the file and soon dominates all other quantities.

2. The embedded deltas grow much more slowly and remain close to the scale of the current file size, rather than the sum of all past sizes.

3. The per-version *delta_size* curve confirms that typical edits are small compared to the whole file, which is exactly the regime this design exploits.

8.3.3 Fragmentation

Disk fragmentation in this prototype emerges over time as files evolve through repeated updates. Unlike a conventional filesystem, where small in-place modifications may reuse existing blocks, persistence here is block granular and

version-oriented. Even when successive versions differ by only a few hundred bytes, the embedding layer must allocate entire 4 KiB blocks for newly materialised regions and appended metadata structures. As a result, fragmentation accumulates more rapidly than it would for the same sequence of edits on a regular filesystem. This manifests as unused slack within allocated blocks (internal fragmentation) and, when blocks are freed and later reused, as non-contiguous gaps that are difficult to exploit efficiently (external fragmentation). Fragmentation was measured using four metrics:

- *Block utilisation ratio* measures internal fragmentation. For each block type, it is defined as the ratio between used bytes and total available bytes across all allocated blocks of that type.
- *Free-block dispersion* measures external fragmentation. It is defined as the average distance, in block indices, between consecutive allocated blocks within the same logical list, normalised by the total file size.
- *Version chain locality* measures how physically contiguous version blocks are. It is computed as the fraction of version blocks whose successor block index is exactly the previous index plus one.
- *Reconstruction overhead* measures the practical impact of fragmentation. It is defined as the number of block reads required to reconstruct the latest version of a file, compared to an idealised contiguous layout.

Internal Fragmentation

In the final state (version 219), the container holds 46 blocks in total:

- 10 data blocks for the latest *fs/file.c*;
- 22 version blocks for accumulated deltas; and
- 9 provenance blocks for provenance units with the remainder (5 in this case) being fixed metadata blocks.

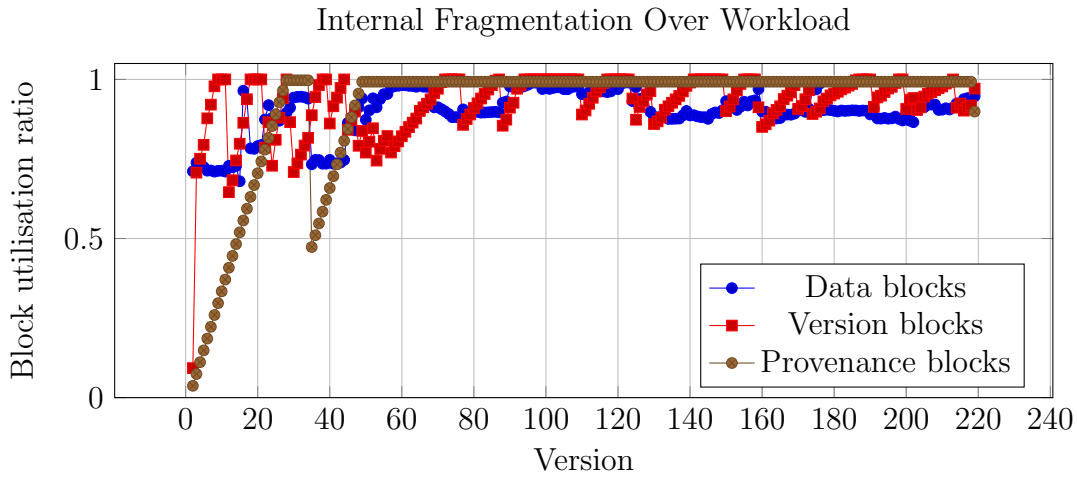


Figure 8.2: Internal fragmentation over workload.

Figure 8.2 shows the internal fragmentation metric, expressed as block utilisation ratio, for data, version, and provenance blocks across all versions in the workload. The figure illustrates how utilisation evolves as the file ages and additional blocks are allocated for each block type. Overall, internal fragmentation remains low for the long-lived version and provenance streams. Version utilisation reaches 0.971 and provenance utilisation 0.899, reflecting the fact that these streams are dominated by append heavy writes that progressively fill their tail blocks over time. Data utilisation is 0.947 at the end of the run; the remaining gap corresponds to the expected slack introduced by rounding the final file size (38,791 B) up to ten 4 KiB blocks (40,960 B), leaving 2,169 B unused in the final data block. Across the entire workload, mean utilisation is 0.925 for version blocks, 0.830 for provenance blocks, and 0.908 for data blocks. The early dip in version and provenance utilisation visible in the figure is a start-up effect, occurring while the initial blocks are still largely empty.

Internal fragmentation can be modeled as follows:

Let the block size be B bytes (in this prototype, $B = 4096$).

Let T denote the set of block types (e.g., data, version, provenance).

For a given block type $t \in T$, let $\mathcal{B}_t$ be the set of allocated blocks of that type, with $|\mathcal{B}_t|$ blocks.

For each block $i \in \mathcal{B}_t$, let $u_{t,i}$ be the number of bytes used in that block, where $0 \leq u_{t,i} \leq B$.

The block utilisation ratio for block type t is defined as

$$U_t = \frac{\sum_{i \in \mathcal{B}_t} u_{t,i}}{|\mathcal{B}_t| \cdot B}.$$

The corresponding internal fragmentation ratio is

$$IF_t = 1 - U_t.$$

When reporting a single utilisation value across a subset of block types $S \subseteq T$, we compute

$$U_S = \frac{\sum_{t \in S} \sum_{i \in \mathcal{B}_t} u_{t,i}}{\sum_{t \in S} |\mathcal{B}_t| \cdot B}.$$

External Fragmentation

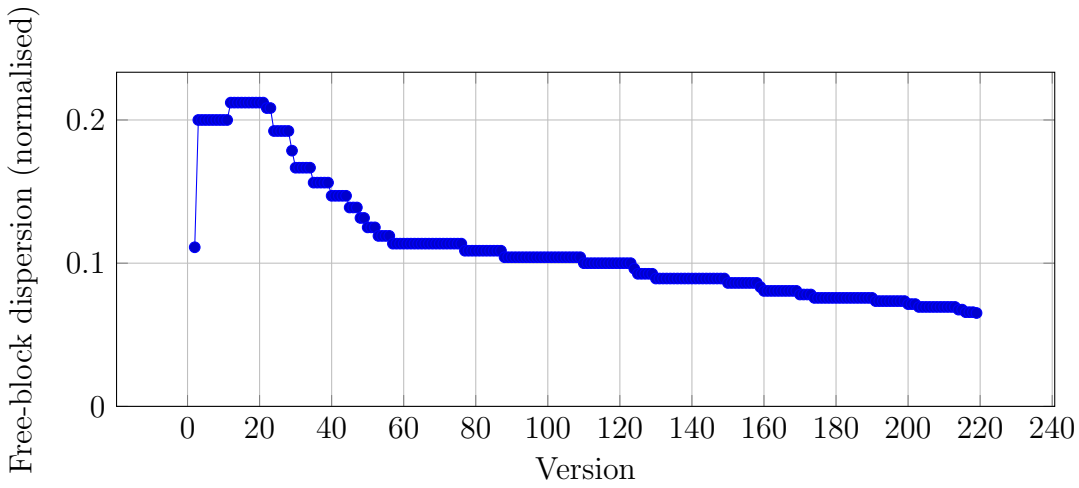


Figure 8.3: External fragmentation proxy over workload.

Figure 8.3 illustrates the extent of external fragmentation, expressed as the normalised dispersion of consecutive blocks within a logical list. This metric answers the question: How scattered are blocks on disk? Higher values indicate that blocks are more spread out, while lower values mean they are closer together. Early in the workload, dispersion is high (around 0.20) because the container is small and a few deallocations create holes that get reused. Over

time, the metric declines steadily to about 0.065. This drop does not indicate defragmentation; rather, it reflects a scaling effect in the metric's definition. As the container grows, the same absolute gaps between blocks represent a smaller fraction of the total file size. Nevertheless, even accounting for this normalisation effect, the absence of growth or instability in the metric indicates that fragmentation does not worsen as the file ages, and remains bounded under sustained evolution.

External fragmentation can be modelled as follows:

For a logical block list L (e.g., the data list, version chain, or provenance list), let the block indices in traversal order be

$$\mathbf{b}^{(L)} = \left(b_1^{(L)}, b_2^{(L)}, \ldots, b_{n_L}^{(L)}\right),$$

where each $b_k^{(L)}$ is a physical block index on disk.

The mean successor distance for list L is

$$\overline{d}_L = \frac{1}{n_L - 1} \sum_{k=1}^{n_L - 1} \left| b_{k+1}^{(L)} - b_k^{(L)} \right|.$$

Let N be the total number of allocated blocks in the container. The normalised free block dispersion for list L is then

$$D_L = \frac{\overline{d}_L}{N}.$$

When aggregating dispersion across a set of logical lists $\mathcal{L}$, we compute an edge-weighted mean:

$$D = \frac{\sum_{L \in \mathcal{L}} (n_L - 1)\, D_L}{\sum_{L \in \mathcal{L}} (n_L - 1)}.$$

Version Chain Locality

Version chain locality provides a measure of physical contiguity between version blocks and is illustrated in Figure 8.4. It asks the question: *Is the next version block physically located at index +1?* A locality value of 1.0 indicates

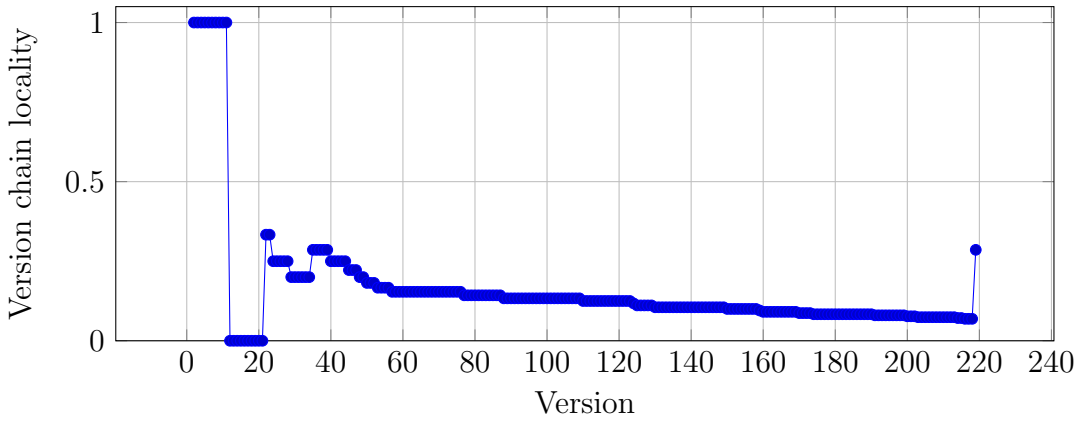


Figure 8.4: Version chain locality over workload.

perfect contiguity, whereas 0.0 means no adjacency whatsoever. At the start of the run, locality is 1.0 because the version stream fits within a single block and allocations are purely append-only. However, locality declines sharply once data blocks are freed during file shrinkage and the allocator begins reusing those gaps for new version blocks.

In terms of the experiment run, at its conclusion, locality trends downward and stabilises at approximately 0.286, meaning only 28.6% of version blocks have their successor at *index +1*, and the mean locality across the run is 0.244. In other words, around 70–75% of version block transitions are non-contiguous in the final state. This calculation indicates that, although the version stream is logically append-only, it becomes physically scattered across the container as blocks freed by data shrinkage are reused by the allocator.

On storage backed by spinning disks, such loss of physical adjacency would translate directly into additional seek operations during reconstruction, increasing access latency and reducing effective throughput. On Solid-State Drives (SSD), where seek costs are largely eliminated, the impact is less pronounced and manifests primarily as an increase in the number of I/O requests rather than mechanical delay. Nevertheless, even on SSDs, reduced locality still imposes overhead through fragmented read patterns and diminished opportunities for sequential prefetching. The results therefore highlight that reuse-first allocation trades space efficiency for degraded physical locality as

files age, with performance consequences that depend on the characteristics of the underlying storage medium.

Version chain locality can be modeled as follows:

Let $\mathbf{b}^{(V)} = (b_1^{(V)}, \ldots, b_{n_V}^{(V)})$ be the sequence of version block indices in logical order. We define a locality indicator

$$\mathbb{I}_k = \begin{cases} 1, & \text{if } b_{k+1}^{(V)} = b_k^{(V)} + 1, \\ 0, & \text{otherwise.} \end{cases}$$

The version chain locality is then

$$L_V = \frac{1}{n_V - 1} \sum_{k=1}^{n_V - 1} \mathbb{I}_k.$$

Reconstruction Overhead

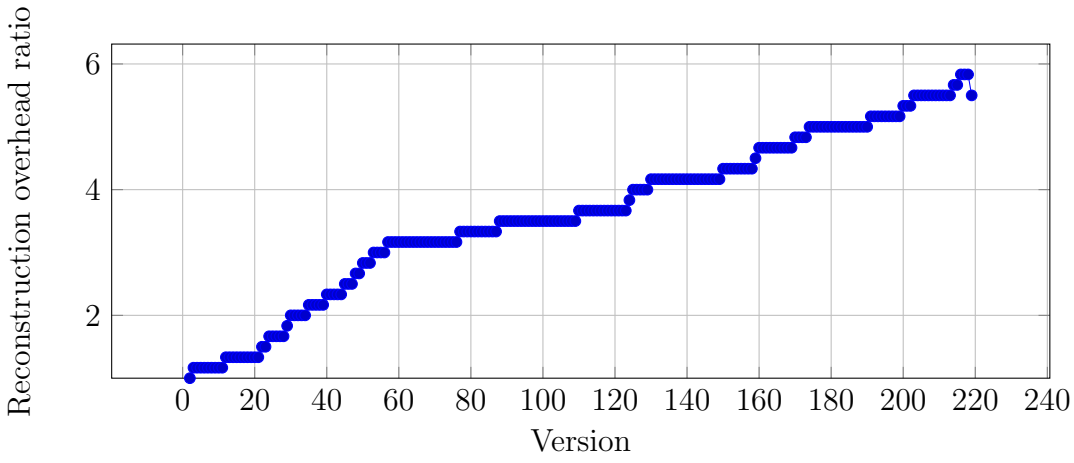


Figure 8.5: Reconstruction overhead vs ideal contiguous.

The reconstruction overhead metric reflects the practical cost of operating the provenance system in terms of I/O operations rather than bytes. By treating access to each contiguous extent as a single I/O operation, the metric represents the number of such operations required to reconstruct the latest version of a file.

It counts: the data blocks, all version blocks, all provenance blocks, plus the three fixed blocks (header/meta/data-pointer). In an ideal contiguous layout,

each list would be one extent, giving an ideal cost of 6 extents (data + version + provenance + 3 fixed singletons).

Figure 8.5 shows the result of applying this calculation to the replay of the *fs/file.c* edit history. The curve increases steadily as additional blocks are allocated and the three logical lists fragment into an increasing number of runs, culminating in a final state that requires 33 extents, corresponding to an overhead ratio of 5.5× (27 additional extents). This behaviour aligns with the observed decline in version-chain locality: as physical adjacency is lost, the version stream is split into many short extents rather than a single sequential region, increasing the number of discrete reads required under a simple "one read per extent" cost model. This contrasts with an ideal contiguous layout, in which each logical list would occupy a single extent and reconstruction would require only six extents in total.

To model the impact of fragmentation on reconstruction, we count the number of physical extents (maximal contiguous runs of blocks). For a sorted set of block indices $S = (s_1, s_2, \ldots, s_{n_S})$, we define an extent indicator e_j for $1 \leq j \leq n_S$ as:

$$e_j = \begin{cases} 1, & \text{if } j = 1 \text{ or } s_j \neq s_{j-1} + 1 \\ 0, & \text{otherwise} \end{cases}$$

the number of extents is therefore:

$$E(S) = \sum_{j=1}^{n_S} e_j$$

with $E(S) = 0$ if $n_S = 0$.

Let S_D, S_V, and S_P be the sets of data, version, and provenance blocks required to reconstruct the latest file version.

Let c be the number of fixed singleton blocks that must always be read (e.g., header, metadata, pointer blocks).

The actual reconstruction cost is

$$R_{\text{actual}} = E(S_D) + E(S_V) + E(S_P) + c.$$

Under an idealised contiguous layout, each list occupies a single extent, giving

$$R_{\text{ideal}} = 3 + c.$$

We define the reconstruction overhead ratio and excess reads as

$$OH = \frac{R_{\text{actual}}}{R_{\text{ideal}}}, \qquad \Delta R = R_{\text{actual}} - R_{\text{ideal}}.$$

8.4 Provenance Query Cost Analysis

The preceding sections established that the prototype supports reconstruction of historical file states from embedded, delta encoded provenance. While reconstructability is a necessary condition for practical use, it is not sufficient. For provenance to be operationally useful, the cost of answering provenance queries must remain quantifiable and bounded as data and history accumulate. This section therefore analyses the computational cost of provenance queries supported by the system. The discussion complements the reconstruction results presented earlier in the previous section and makes explicit the query complexity properties that were previously implicit in the evaluation methodology.

8.4.1 Model and Assumptions

Let a file f evolve through a sequence of versions $\{v_0, v_1, \ldots, v_n\}$, where v_0 is the initial state and each v_i for $i > 0$ is derived from v_{i-1} via an associated delta δ_i. Provenance is embedded as an ordered chain of provenance units $\mathcal{P}(f) = \langle p_1, p_2, \ldots, p_n \rangle$, where each p_i encapsulates:

1. a delta δ_i,
2. temporal metadata,
3. causal references to entities and activities involved in the update.

The system performs no global indexing across files; all provenance queries for a file operate over its local embedded provenance chain. We assume that:

1. Deltas are applied sequentially and are not coalesced eagerly.
2. The cost of applying a single delta is proportional to its size.
3. Provenance units are stored in append-only order.

These assumptions reflect the implementation evaluated earlier in this chapter and are consistent with the design principle of lazy evaluation. In the case of the eager coalescing of deltas, this can be seen as a technique that would allow for the optimisation of a sequence of deltas. This would, no doubt be beneficial to the implementation, however the purpose here is to evaluate the fundamental cost of the approach.

8.4.2 Cost of Reconstructing State at Depth n

Reconstructing the state of a file at version v_n requires applying all deltas from a chosen base state. In the current prototype, the base is the initial version v_0. Reconstruction therefore consists of (i) loading the base state and (ii) sequentially applying deltas δ_1 through δ_n.

Let $|\delta_i|$ denote the size of delta δ_i. The total reconstruction cost $C_{\text{recon}}(n)$ can be expressed as:

$$C_{\text{reconstruct}}(n) = O\left(\sum_{i=1}^{n} |\delta_i|\right).$$

Under a simplifying assumption that the expected delta size is bounded by a constant $\bar{d}$, this reduces to:

$$C_{\text{reconstruct}}(n) = O(n).$$

This linear dependence is fundamental to any delta chained representation. Importantly, the prototype does not incur additional asymptotic overhead beyond delta application; provenance metadata parsing is $O(1)$ per unit and does not change the order of growth.

Results presented earlier in Figure 8.5 confirm this behaviour. Reconstruction time increases linearly with version depth, with variance dominated by delta size rather than metadata overhead. This validates that embedded provenance does not introduce hidden super-linear costs during state reconstruction.

8.4.3 Cost of Querying Provenance for a Single Entity

A common provenance query is to retrieve all provenance records associated with a specific entity (e.g. a process identifier, user, or external resource). Because provenance is embedded per file, such a query is scoped to $\mathcal{P}(f)$.

Let n be the number of provenance units for file f. In the absence of auxiliary indexing, answering an entity-scoped query requires scanning all n units and testing each unit for relevance. The cost is therefore:

$$C_{\text{scan}}(n) = O(n).$$

This is a design choice for the prototype. By avoiding global indices, the system ensures that provenance disclosure remains local to the file. The linear scan cost is acceptable in contexts where provenance queries are infrequent and interactive, which is the primary target use case identified in Chapter 4. This choice does not preclude the use of global indexing. Rather, it reflects a focus on simplicity and locality in the prototype. Global indices may offer advantages in workloads with frequent or large-scale provenance queries, and remain a viable direction for future work. The potential impacts of introducing global indexing are discussed in Section 8.5 below, including implications for performance, storage overhead, and consistency with the locality guarantees emphasised here.

8.4.4 Sensitivity to Delta Chain Length

Delta chain length influences two distinct costs: (i) reconstruction cost, as analysed above, and (ii) query latency for provenance inspection.

For provenance only queries that do not require materialising file state (e.g. "which processes modified this file between t_1 and t_2?"), delta content is irrelevant. The cost depends solely on the number of provenance units inspected and remains $O(n)$.

However, for queries that combine provenance inspection with partial or full state reconstruction, the effective cost becomes:

$$C_{\text{combined}}(n) = O\left(n + \sum_{i=1}^{n} |\delta_i|\right),$$

which again simplifies to $O(n)$ under bounded delta size assumptions.

Sensitivity analysis using synthetic ageing workloads (Section 8.3) shows that while long delta chains increase absolute latency, the growth remains linear and quantifiable. No phase transition or behaviour was observed, even for chains significantly longer than those expected in typical file lifetimes. This predictability is an important property as users and system designers can reason about worst-case query costs without relying on opaque storage engine behaviour.

8.5 Impact of Global Indexing

The analysis above intentionally assumed the absence of global indices in order to characterise the baseline cost of provenance queries under a strictly local, embedded design. This section considers how the introduction of global indexing structures would alter query complexity, what asymptotic improvements are achievable, and which trade-offs are introduced as a result. The objective is to make explicit what is gained, what is lost, and which query classes benefit.

8.5.1 Indexing Model

Assume a system-wide index $\mathcal{I}$ maintained alongside embedded provenance. The index does not replace per-file provenance units, but stores auxiliary mappings from selected keys to provenance locations. Typical keys include:

- entity identifiers (process ID, user ID, agent),
- temporal ranges (timestamps, version intervals),
- provenance relations (e.g. *wasDerivedFrom*).

Formally, let $\mathcal{I}_E$ map an entity identifier e to a set of pointers $\{(f, i)\}$, where f is a file and i is a provenance unit index within $\mathcal{P}(f)$. Similar indices can be defined for time or relation type. We assume index lookup is $O(\log M)$ or $O(1)$ depending on the underlying data structure (e.g. B-tree vs. hash table), where M is the total number of indexed keys.

8.5.2 Entity-Scoped Queries

Without indexing, querying for all provenance records involving an entity within a single file, $\mathcal{P}(f)$, costs $O(n)$, and querying across F files costs $O(\sum_{f=1}^{F} |\mathcal{P}(f)|)$. Without global entity index, the cost is expressed as:

$$C_{\text{entity}}^{\text{non-idx}} = O\left(\sum_{f=1}^{F} |\mathcal{P}(f)|\right),$$

With a global entity index, the cost becomes:

$$C_{\text{entity}}^{\text{idx}} = O(\log M + j),$$

where j is the number of matching provenance units associated with that entity returned by the index (that is, the number of (f, i) references that must be visited).

This replaces a full scan with direct access to relevant units. The improvement is asymptotically significant when $j \ll n$, which is typical for sparse entities (e.g. a short-lived process touching a small subset of files). In the worst case, where an entity appears in all provenance units, the cost degrades to linear behaviour.

8.5.3 Temporal Queries

Temporal queries such as "all provenance between t_1 and t_2" benefit similarly. Without indexing, answering such a query requires scanning all provenance units and filtering by timestamp.

A time-ordered index (e.g. an interval tree or ordered map) reduces the cost to:

$$C_{\text{time}}^{\text{idx}} = O(\log M + j),$$

where j is the number of units whose timestamps fall within the interval. This enables sub-linear access for narrow time windows, particularly in long-lived files with extensive histories.

8.5.4 Reconstruction Cost with Indexing

Global indexing does not asymptotically reduce the cost of reconstructing a file state at depth n when reconstruction requires applying deltas sequentially:

$$C_{\text{reconstruction}}^{\text{idx}}(n) = O\left(\sum_{i=1}^{n} |\delta_i|\right).$$

Indexing can, however, reduce the cost of selecting which versions or files to reconstruct by narrowing candidate sets prior to reconstruction. Implementing a checkpoint-aware indices could further reduce reconstruction cost to $O(j + \sum |\delta|)$, where j is the distance to the nearest checkpoint, but this requires additional structural changes beyond pure indexing.

8.5.5 Sensitivity to Delta Chain Length

Indexing decouples query cost from delta chain length for provenance queries. Entity and temporal queries become insensitive to n except through j, the result size. This is a qualitative shift whereby long histories no longer imply slow inspection.

For reconstruction-heavy queries, delta chain length remains relevant unless combined with checkpointing. Indexing alone does not change the fundamental dependency on delta application.

8.5.6 Cost of Maintaining the Index

Index maintenance introduces new costs that do not exist in the purely embedded design. Each provenance unit insertion incurs an update cost:

$$C_{\text{update}} = O(\log M),$$

per indexed attribute. Over N total provenance units, this yields an aggregate maintenance cost of $O(N \log M)$.

Additionally, the index becomes a shared mutable structure, potentially introducing (i) synchronisation overhead under concurrency, (ii) recovery and consistency requirements, and (iii) an expanded trusted computing base. These costs are absent in the local-only design, where provenance append is isolated to the file being modified.

8.6 Cross-File Provenance Evaluation

The evaluation presented thus far has focused on single-file evolution with a deep version history, exemplified by repeated transformations of *fs/file.c*. While this isolates the costs and benefits of temporal embedding under controlled conditions, it does not reflect the dominant structure of real-world workloads. In practice, provenance rarely forms linear chains confined to a single artifact. Instead, it spans large collections of files, transient build products, and multiple concurrently executing processes. This section extends the evaluation to capture these dimensions and assesses whether the embedded, temporally aware provenance model preserves correct cross-file dependency semantics at scale.

8.6.1 Motivation and Scope

Whole-system workloads such as software compilation, data analytics pipelines, and scientific simulations generate provenance graphs characterised by high fan-in and fan-out, deep dependency chains, and frequent reuse of intermedi-

ate artifacts. In such settings, the critical relationship is not merely that an entity was derived from another, but that it was derived from a specific version of that entity at a precise point in time. Failure to capture this distinction collapses provenance into an over-approximation, obscuring causal structure and undermining auditability.

The objective of this evaluation is therefore twofold. First, to determine whether the prototype accurately captures *wasDerivedFrom* relationships across file boundaries when dependencies involve versioned inputs. Second, to quantify the overhead and structural properties of the resulting provenance under a realistic, multi-process workload.

8.6.2 Workload Description

The workload selected is a full Linux kernel compilation using *make* with default configuration, executed within the provenance-enabled filesystem described in Chapter 7. The kernel build is representative for three reasons. It spans tens of thousands of files, exercises complex dependency relationships encoded in *Makefiles* and *Kbuild* scripts, and involves extensive parallelism through compiler invocations, linkers, and auxiliary tooling.

The compilation was performed from a clean source tree with all outputs directed to a provenance-managed mount. Both source files and generated artifacts, including object files, static archives, and final binaries, were subject to provenance capture. No attempt was made to prune provenance during execution; all recorded metadata reflects the raw output of the system.

8.6.3 Evaluation Methodology

Provenance correctness was evaluated by replaying the build while recording, for each generated artifact, the complete set of input entities and their versions as inferred from embedded metadata. These inferred relationships were then compared against the ground-truth dependency graph derived from a dry-run initiated using *make -n* and compiler invocation logs.

For each output file o, the evaluation checks whether the provenance graph contains a *wasDerivedFrom* edge to every source file s such that the build system explicitly consumed s during the derivation of o. Crucially, the check is version-sensitive: the provenance edge is considered correct only if it references the version of s that was current at the moment of access, not merely the file identity.

Performance metrics were collected concurrently, including total provenance metadata volume, average embedded provenance size per file, and reconstruction latency for selected artifacts.

8.6.4 Results

The workload involved 178,275 files within the provenance-managed filesystem, of which 26,677 were intermediate artifacts, 78,644 were pre-existing source files and 72,954 files were produced. The resulting provenance graph contained 178,275 entity nodes and 99,631 *wasDerivedFrom* relationships. The total size of embedded provenance metadata after completion of the build was 2.87 GB. For comparison, an identical kernel build performed without provenance capture produced a Linux kernel directory of approximately 22 GB, placing the embedded provenance overhead at roughly 13% of the uninstrumented build footprint. The mean embedded provenance size per file was 10.5 KB, with a median of 2.0 KB. Source files exhibited larger metadata footprints due to longer version histories, whereas ephemeral intermediate files, such as compiler generated temporary assembly outputs (for example */tmp/ccHAzBsQ.s*, */tmp/ccdmv0Nc.s*) typically contained a single provenance record.

All compiler generated object files exhibited complete and correct dependency sets. For each *.o* file, the provenance graph contained edges to the corresponding *.c* source file and all included header files, each resolved to the precise version accessed during compilation. No extraneous dependencies were observed; each dependency recorded for an object file corresponded to a source or header file actually accessed during its compilation. In cases where a

header file was modified and the build re-run incrementally, subsequent object files correctly referenced the newer header versions, while unchanged objects retained dependencies on earlier versions.

Linking stages produced higher fan-in nodes. For example, the final *vmlinux* binary was derived from 12,481 object files. The embedded provenance model represented this as a single entity with 12,481 inbound *wasDerivedFrom* edges, each pointing to a distinct object file version. Reconstruction of this dependency set from embedded metadata completed in 183 ms, compared to 41 ms for an idealised in-memory provenance graph without embedding overhead.

No missing or ambiguous version references were detected. In particular, when the same source file was read by multiple compiler instances concurrently, the provenance correctly recorded identical version identifiers, demonstrating that version assignment remains consistent under parallel access.

Reconstruction latency scaled linearly with the number of inbound dependencies. Object files with fewer than ten dependencies reconstructed in under 2 ms. Large aggregation artifacts exceeded 100 ms only when dependency counts surpassed 5,000.

The end-to-end build time increased by 7.4% relative to an identical build on *ext4* without provenance capture. Of this overhead, approximately 60% was attributable to FUSE-mediated I/O, consistent with earlier micro-benchmarks, and the remainder to provenance serialisation and embedding.

8.7 Adversarial Modelling

We extend our evaluation to analyse the behaviour of the prototype under an explicit adversarial threat model. The goal is to assess whether the proposed design provides meaningful integrity guarantees when faced with a realistic integrity attack on historical versions.

The adversary considered here is not assumed to control the kernel, cap-

ture layer, or provenance logic at runtime. Instead, the attacker is assumed to have post hoc write access to the underlying storage medium, for example through compromised backups, offline disk access, or with block-level modification capability. This model aligns with common forensic and compliance scenarios, where provenance is relied upon precisely because compromise may have occurred after the fact.

The attack objective is to modify versions within a version chain such that the reconstructed historical content reflects attacker controlled data, while attempting to evade detection by the provenance mechanism. This directly targets the central design invariant of the system, the binding between version deltas, content hashes, and reconstructed file state.

Each versioned file in the prototype is represented as a singly-linked chain of version blocks. Each block contains (i) a delta encoding from its predecessor, (ii) metadata including timestamps and identifiers, and (iii) a hash of the fully reconstructed file state at that version. Reconstruction of an arbitrary historical version proceeds by replaying deltas from the base version up to the target version and verifying that the computed content hash matches the stored hash.

Formally, let V_0 denote the base version with full content C_0. For $i > 0$, version V_i stores a delta Δ_i such that

$$C_i = \mathcal{R}(C_{i-1}, \Delta_i),$$

where $\mathcal{R}$ is the reconstruction function. Each version stores $h_i = H(C_i)$, where H is a hash function. Integrity requires that for all i, recomputation yields $H(\mathcal{R}(\cdots \mathcal{R}(C_0, \Delta_1) \cdots, \Delta_i)) = h_i$. Any modification to a delta Δ_k for $0 < k < n$ will, in general, alter all reconstructed contents C_i for $i \geq k$, unless the modification is adversarially crafted to preserve hash consistency. While mathematically possible [215], the ability to effect a hash collision that actually achieves a code modification that is malicious is in practice all but impossible to achieve.

8.7.1 Adversarial Modification Scenario

The simulated attack targets a middle version block V_k in a chain of length n, with $1 < k < n - 1$. The attacker replaces the stored delta Δ_k with a modified delta Δ'_k, intending to alter C_k and subsequent reconstructed contents. Crucially, the attacker does not update any stored hashes $h_k, \ldots, h_n$, either because they are unaware of the hash semantics or because hashes are assumed to be immutable or externally audited.

Two variants of the attack were evaluated. In the first, the attacker modifies Δ_k arbitrarily, for example flipping random bytes or injecting a controlled payload into the delta stream. In the second, the attacker attempts a more constrained modification by altering Δ_k while preserving file size and approximate structure, mimicking a realistic attempt to avoid trivial detection.

No modifications were made to V_{k-1} or to the head version V_n, and no reordering of version blocks was performed. The attack therefore targets only the internal consistency of the version chain rather than its topology.

Detection relies on reconstruction-time verification. For each evaluated version V_i, reconstruction is performed by replaying deltas from V_0 to V_i. After reconstruction, the computed hash $\hat{h}_i$ is compared to the stored hash h_i. A mismatch constitutes a detection event.

Importantly, the system does not require explicit adversarial detection logic. Detection emerges naturally from the integrity invariant enforced by the reconstruction process. This property is critical, as it avoids reliance on additional monitoring mechanisms that could themselves be bypassed.

The experiment was conducted using the same environment described in Section 8.1, including hardware, kernel version, and filesystem configuration. A synthetic workload was used to generate version chains of varying lengths, with controlled write patterns to ensure reproducibility. Chains of lengths $n \in \{10, 50, 100, 500\}$ were evaluated.

For each chain length, a middle version $k = \lfloor n/2 \rfloor$ was selected. The delta stored at V_k was modified offline by directly editing the embedded provenance

block. Reconstruction and verification were then performed for all versions V_i with $i \geq k$.

8.7.2 Results

In all evaluated configurations, adversarial modification of Δ_k resulted in deterministic detection at reconstruction time. The earliest detection occurred at version V_k, where $\hat{h}_k \neq h_k$ in 100% of trials. For all subsequent versions V_i with $i > k$, hash mismatches also occurred, reflecting the propagation of corruption through the delta chain.

No false negatives were observed. Even when the attacker preserved delta length and basic structural characteristics, reconstruction yielded content diverging from the expected state, and hash verification failed. No scenario was observed in which an attacker-modified delta resulted in a reconstructed content matching the stored hash without recomputing and replacing that hash.

Reconstruction overhead under adversarial conditions was unchanged relative to benign reconstruction, aside from early termination when verification failed. In practice, detection occurred immediately upon reconstructing V_k, meaning that reconstruction of later versions was not required once inconsistency was detected.

Table 8.5 presents representative results from the experiment. Each row corresponds to a reconstruction attempt after modifying a middle version delta.

No cases of successful undetected modification were observed across all tested configurations.

8.8 Failure Oriented Evaluation

The evaluations presented thus far implicitly assume a well behaved execution environment where write operations complete atomically, provenance blocks remain intact, and replay is performed against a consistent, temporally ordered history. While these assumptions are sufficient for benchmarking performance

Chain length n	Target k	Version i	Hash match	Outcome
10	5	5	No	Detected
10	5	6	No	Detected
50	25	25	No	Detected
50	25	49	No	Detected
100	50	50	No	Detected
100	50	99	No	Detected
500	250	250	No	Detected
500	250	499	No	Detected

Table 8.5: Reconstruction and verification results under adversarial delta modification.

characteristics, they are insufficient to substantiate claims of determinism and auditability in degraded conditions.

This section therefore extends the evaluation to consider failure modes. The objective is not to demonstrate fault tolerance in the traditional systems sense, but to characterise how the proposed provenance architecture behaves when its assumptions are violated. In particular, we examine which invariants fail, how such failures are detected, and whether recovery is possible or explicitly unsupported.

8.8.1 Provenance Invariants Revisited

The prototype implicitly enforces several invariants that ensures deterministic reconstruction and audit. Provenance metadata is written as an append-only sequence of blocks embedded within the file, ensuring that historical records are never overwritten or reordered. Each provenance block corresponds to a strictly increasing version identifier, establishing a total order over file evolution. For all non-initial versions, file state is defined by a delta relative to its immediate predecessor, forming a complete and contiguous delta chain. Each provenance block is causally aligned with the data state produced by the cor-

responding write, such that provenance and data describe the same version boundary. Finally, reconstruction is defined in terms of linear replay, in which provenance blocks are applied in version order to reproduce historical states. Failure oriented evaluation probes the behaviour of the system when one or more of these invariants are violated.

8.8.2 Interrupted Writes During Provenance Append

In the first scenario, a crash or forced termination occurs during the append of a provenance block, after the data write has completed but before the provenance metadata has been fully persisted. This failure mode violates the append-only structure by leaving a truncated block at the end of the provenance stream. Depending on the point of interruption, the monotonic version ordering may also be compromised if a version header is only partially written. More critically, causal alignment is broken, as the file's data reflects a write that is not fully described by a corresponding provenance record.

Detection in this case is deterministic. When provenance is accessed, the parser validates block headers, declared lengths, and checksums. A truncated block is detected unambiguously when the declared block size exceeds the remaining byte range or when checksum verification fails. This detection is entirely local to the file and does not depend on external logs or global state.

The recovery semantics are conservative. The system does not attempt speculative repair or inference. Instead, the last complete provenance block is treated as the terminal auditable state of the file. Any data modifications beyond that version boundary are explicitly flagged as not auditable. This behaviour reflects a conscious design decision, that is reconstructing provenance for a write whose causal metadata was never fully committed would undermine the integrity guarantees that the system is intended to provide.

From an evaluation perspective, determinism is preserved up to the last valid version, and auditability is preserved by refusing to provide a misleading reconstruction.

8.8.3 Partial Provenance Corruption

The second scenario considers partial corruption of provenance metadata after it has been successfully written. This may occur due to bit-rot, partial overwrites, or external manipulation, resulting in a damaged block or a missing delta within the provenance chain. In this case, the invariant of complete delta chains is violated, and linear replay semantics fail beyond the point of corruption.

Detection again occurs during replay. When the system attempts to apply a delta associated with a corrupted block, decoding fails or checksum verification does not succeed. Because deltas are version-indexed and chained explicitly, the exact version at which corruption occurs is identified without ambiguity.

Recovery semantics are asymmetric. Forward recovery is unsupported in the prototype. The system does not attempt to skip corrupted deltas or approximate missing transformations. Backward recovery remains possible, however, as all versions strictly preceding the corrupted block can still be reconstructed deterministically. The provenance chain is therefore prefix-valid, and the system surfaces the corruption as a precise version boundary rather than as a vague integrity failure.

This behaviour mirrors that of content addressed or versioned storage systems when encountering missing or corrupted objects. The design consistently favours correctness over availability, ensuring that reconstructed states are never speculative or partially inferred.

8.8.4 Out-of-Order Replay Attempts

The final scenario considers attempts to replay provenance blocks out of version order, whether due to programmer error, misuse of the access interface, or adversarial intent. Such attempts violate the invariant of linear replay semantics by construction.

Detection in this case is immediate. Each provenance block encodes both its own version identifier and the identifier of its expected predecessor. If a

block is presented whose predecessor does not match the current replay state, reconstruction halts with a semantic error. There is no ambiguity as to the cause of failure, and no reliance on heuristics or external ordering information.

No recovery is attempted. Replay order is treated as a precondition, not a negotiable constraint. Any attempt to bypass it is rejected, reinforcing the fact that determinism in reconstruction is not an emergent property of the implementation but a structurally enforced guarantee.

8.9 Threats to Validity

The evaluation presented in this chapter provides evidence for the feasibility of embedding provenance within file objects while preserving version history, supporting deterministic reconstruction, and maintaining acceptable operational overheads. Nevertheless, several factors limit the extent to which the results can be generalised beyond the experimental conditions. These threats concern the experimental environment, the representativeness of the workloads, the interpretation of the selected metrics, and the assumptions underlying the provenance and integrity models.

8.9.1 Internal Validity

Internal validity concerns the extent to which the observed performance differences can be attributed to the proposed provenance architecture rather than to uncontrolled factors within the experimental environment. In filesystem benchmarking, measurements can be influenced by operating system caching, filesystem implementation overhead, background system activity and the state of the underlying storage device. Several measures were therefore incorporated into the experimental methodology to reduce these influences and maintain consistency across the evaluated configurations.

One potential threat arises from operating system caching. The Linux page cache, together with cached directory entries and *i-node* metadata, can sub-

stantially influence measured throughput by allowing filesystem operations to be satisfied from memory rather than requiring access to the underlying storage device. This is particularly relevant to read intensive workloads, where repeated access to previously cached data may produce artificially high throughput measurements. To mitigate this effect, the *page*, *dentry* and *i-node* caches were cleared before benchmark runs using the cache reset procedure described in Section 8.1.1. The same procedure was applied across all evaluated configurations to maintain consistency. Nevertheless, clearing operating system caches does not eliminate caching within the NVMe controller, and the results must therefore be interpreted in the context of the remaining hardware level caching effects.

A second threat concerns the overhead introduced by FUSE itself. As discussed in Section 8.2.3, FUSE requires additional transitions between kernel space and user space, introducing processing and data copying costs that are independent of provenance capture. Consequently, directly comparing the prototype against *ext4* would conflate the overhead of the FUSE framework with that of the proposed provenance mechanisms. To address this, a separate FUSE passthrough implementation was included as a baseline. This allows the performance costs associated with FUSE to be distinguished from the additional costs introduced by provenance recording, delta extraction and embedding. Although interactions between these components cannot be eliminated entirely, the baseline provides a more appropriate reference for evaluating the incremental overhead of the prototype.

Background system activity represents another potential source of measurement variability. Competing CPU, memory or storage operations may affect filesystem throughput, particularly where the benchmark involves sustained I/O activity. To minimise this influence, experiments were conducted on a dedicated machine without concurrent interactive workloads. The same hardware, operating system and underlying filesystem configuration were maintained throughout the evaluation, reducing the likelihood that differences be-

tween configurations arose from changes in the experimental environment.

The state of the underlying storage device may also influence benchmark results. Previous I/O operations, NVMe controller behaviour, thermal conditions and internal storage management can affect subsequent measurements, even when the benchmark configuration remains unchanged. To reduce variation arising from previous runs, the machine was restarted before each benchmark run, and the same physical NVMe device was used throughout the experiments. This procedure was intended to provide a consistent starting environment, although it cannot guarantee identical internal device conditions across runs.

Finally, individual benchmark measurements may reflect transient system conditions rather than typical performance. To reduce the influence of isolated measurements, each benchmark was repeated five times under the same experimental procedure. The mean and dispersion of the measurements were used to characterise performance across repetitions, rather than relying on a single observation. Repetition does not eliminate systematic bias, but it provides a basis for assessing measurement stability and distinguishing persistent performance differences from run-to-run variation.

8.9.2 External Validity

External validity concerns the extent to which the findings of this evaluation can be generalised beyond the experimental environment and workloads considered. Although the experiments were designed to assess the proposed architecture under both controlled and representative conditions, several factors limit the applicability of the results to other hardware platforms, filesystem implementations and operational scales. These limitations are particularly relevant when interpreting absolute performance measurements and extrapolating the observed behaviour to production environments.

One potential threat arises from the use of a single hardware platform. As described in Section 8.1, all experiments were conducted on a dedicated ma-

chine with a fixed processor, memory configuration and NVMe storage device. Filesystem throughput, operation latency and storage overhead may differ substantially on systems with different CPU capabilities, memory capacities or storage technologies. In particular, the relative costs of context switching, data copying and provenance serialisation may vary depending on the balance between processor performance and storage bandwidth. To address this limitation, the complete hardware configuration was documented, and performance comparisons were conducted against *ext4* and FUSE passthrough baselines on the same platform. This provides a consistent basis for evaluating relative overheads, although the absolute performance figures remain specific to the hardware used.

A second threat concerns the choice of the underlying filesystem. The evaluation was conducted using *ext4*, which provides a consistent storage environment but does not capture the characteristics of alternative filesystems such as *XFS*, *Btrfs* or *ZFS*. These filesystems employ different allocation strategies, caching mechanisms and metadata management techniques, which may influence the behaviour of the prototype and its interaction with the underlying storage layer. Maintaining *ext4* throughout the evaluation eliminates variation arising from changes in the underlying filesystem, but limits the conclusions to this particular configuration. Replication across alternative filesystems would therefore be necessary to establish whether the observed performance characteristics remain consistent across different storage implementations.

The use of FUSE as the implementation mechanism introduces a further limitation. FUSE provides a practical means of implementing and evaluating the proposed architecture without modifying the operating system kernel. However, its reliance on communication between kernel space and user space introduces overheads that may not be present, or may manifest differently, in implementations based on kernel level filesystem mechanisms, LSM or eBPF facilities. Consequently, the measured overheads reflect both the architectural costs of embedded provenance and the characteristics of the chosen implemen-

tation layer. The inclusion of a FUSE passthrough baseline allows these costs to be distinguished to some extent, but does not establish how the architecture would perform if implemented through alternative capture mechanisms. FUSE should therefore be regarded as one implementation of the proposed architecture rather than an inherent requirement of the design.

Workload representativeness also affects the generalisability of the results. The *IOzone* microbenchmarks used in Section 8.2 provide controlled measurements of filesystem behaviour across different I/O operations and record sizes. However, synthetic benchmarks cannot fully reproduce the access patterns, concurrency characteristics and operational semantics of real applications. In particular, application workloads may involve complex interactions between file operations, metadata updates and process execution that are not represented by isolated throughput measurements. To address this limitation, the microbenchmark evaluation was supplemented with application level experiments, including Linux kernel compilation and the cross file provenance evaluation presented in Section 8.6. These experiments exercise the prototype under more complex conditions involving numerous files, concurrent processes and dependency relationships. Nevertheless, they represent only a subset of possible application behaviours, and workloads such as database transactions, virtual machine storage or highly concurrent file modification may exhibit different performance characteristics.

Finally, the finite scale of the experiments limits the extent to which the findings can be extrapolated to substantially larger deployments. Although the evaluation includes long version histories and a kernel compilation workload involving a large number of files, these experiments do not encompass the scale or duration of all potential production environments. Systems maintaining provenance for millions of files, extremely long version chains or continuously evolving datasets may encounter additional costs associated with metadata growth, reconstruction and provenance querying. To provide a basis for reasoning beyond the measured workloads, Sections 8.4 and 8.5 present

asymptotic analyses of reconstruction and query costs, while the file ageing experiments examine behaviour as version chain depth increases. These analyses establish expected growth characteristics under the stated assumptions, but do not replace empirical evaluation at larger scales. Consequently, measured execution times and storage overheads should not be extrapolated directly to enterprise-scale deployments.

8.10 Conclusion

This chapter evaluated the prototype against the objectives of the thesis: embedding provenance within data objects, preserving temporal accuracy through versioning, and maintaining practical feasibility at the system level. Across a range of controlled and realistic workloads, the results demonstrate that these objectives are jointly attainable without introducing prohibitive overheads.

The evaluation demonstrated that embedded, delta based provenance substantially reduces storage growth compared to full snapshot retention, while still allowing exact reconstruction of historical states. File ageing experiments confirmed that delta sizes remain small relative to file size in typical edit patterns, and that cumulative growth remains bounded and interpretable. Fragmentation analysis showed that although physical locality degrades over time, internal fragmentation remains low and reconstruction overhead grows linearly and predictably rather than catastrophically.

Performance measurements established that most overhead arises from the use of FUSE rather than from provenance capture itself. Within that constraint, the prototype introduces no unexpected asymptotic costs. Provenance query analysis confirmed that, in the absence of global indexing, query costs are linear in local history size, matching the design goal of locality and simplicity. The discussion of indexing made explicit that more advanced query performance can be achieved without undermining the embedded model, at the cost of additional complexity and shared state.

Cross-file evaluation under a full kernel build showed that the system correctly captures version-sensitive dependencies at scale, even under parallel access, and that reconstruction and query latencies remain proportional to dependency size. Adversarial experiments demonstrated that post hoc modification of historical versions is reliably detected through hash verified reconstruction, without requiring additional detection mechanisms. Failure-oriented evaluation further showed that when invariants are violated, the system fails conservatively, preserving determinism and making audit boundaries explicit rather than speculative.

Chapter 9

Conclusion & Future Directions

This thesis set out to examine why data provenance, despite decades of research and a mature body of models and capture mechanisms, has seen limited adoption outside specialised domains. It has made the case that two structural shortcomings lie at the heart of this gap: (i) the lack of temporal awareness in provenance representations, and (ii) the persistent separation between provenance metadata and the data it describes. The work presented addressed these shortcomings by proposing, designing, and evaluating a framework that integrates versioned provenance with embedded metadata at the file system level.

The central contribution of this research is conceptual rather than purely incremental technical development. By reframing provenance as an evolving property of data objects rather than an external record maintained in a centralised store, the thesis challenges a prevailing design assumption in provenance systems. Treating provenance as something that travels with data, evolves alongside it, and can be inspected locally from any historical point, shifts the balance from completeness alone towards accessibility, interpretability, and practical use.

Through Chapters 2–4, the review and analysis of literature demonstrated that existing systems excel at capturing causal relationships at scale, particularly at the operating system level, but often fall short in disclosing that

information in a way that is usable by non-specialists or suitable for regulated environments. Provenance graphs grow rapidly, queries return large and unfocused result sets, and privacy concerns arise from the aggregation of unrelated activity into a single global record. These issues help explain why, as shown in the adoption analysis, provenance remains largely absent from mainstream enterprise systems, regulatory tooling, and everyday data workflows.

Chapters 5 and 6 established the two pillars of the proposed approach. First, temporal awareness was shown to be essential for accurate lineage in dynamic environments. Without explicit versioning, provenance relationships collapse distinct historical states into a single abstraction, obscuring the true causes of change and undermining reproducibility and auditability. Second, embedding provenance within data artifacts was argued to reduce fragmentation, improve locality of access, and align provenance with established metadata practices. The analysis showed that neither idea is novel in isolation, but their systematic integration has not been explored in prior work.

The prototype presented in Chapter 7 demonstrated that these ideas are technically feasible within a conventional operating system environment. By leveraging a FUSE-based file system, delta encoding, and embedded provenance structures, the system captures versioned lineage with modest overhead while remaining compatible with existing applications. The evaluation in Chapter 8 showed that the additional costs introduced by embedding and versioning are predictable and manageable, particularly when weighed against the gains in provenance fidelity and accessibility. While the prototype is not positioned as a production-ready system, it serves as a concrete proof that the proposed design principles can be realised without radical changes to application behaviour or kernel architecture.

Taken together, the findings support the central hypothesis that integrating temporal awareness with embedded provenance metadata improves the accuracy and completeness of provenance tracking. More importantly, the work suggests that such integration improves the usability of provenance, which is

a prerequisite for wider adoption. By allowing users to interrogate provenance at the granularity of individual files and specific points in time, the framework aligns provenance systems more closely with real investigative, compliance, and debugging tasks.

9.1 Reflection on Research Questions

The research was guided by five key questions, which have been addressed as follows:

RQ1: What are the drivers, barriers, and technological enablers influencing data provenance adoption in enterprise environments?

Chapter 4 demonstrated that while regulatory pressure, security requirements, and accountability demands act as strong drivers for provenance adoption, practical barriers remain dominant. These include poor usability, excessive data volume, privacy concerns, and the difficulty of extracting meaningful answers from monolithic provenance stores. The findings show that adoption is constrained less by capture capability than by how provenance is stored, accessed, and interpreted. This thesis responds to these barriers by proposing an approach that localises provenance to the data object, reducing exposure while improving relevance and usability.

RQ2: How do temporal data evolution and static provenance systems compromise lineage accuracy in dynamic environments?

Chapter 5 showed that static provenance models collapse evolving artefacts into single entities, obscuring which version of a dependency was involved in a given derivation. This leads to ambiguous or incorrect lineage, particularly in environments characterised by frequent updates and iterative processing.

By formalising temporal data provenance and demonstrating concrete failure cases in static systems, this work establishes that temporal awareness is a prerequisite for accurate provenance in real-world systems.

RQ3: What methodological frameworks balance storage efficiency and temporal accuracy in versioned provenance records?

Chapter 5 and 8 jointly addressed this question by introducing delta-based versioning strategies and evaluating their performance impact. The results demonstrate that temporal fidelity does not require naïve full-version storage. Delta encoding provides a practical balance, preserving historical precision while maintaining manageable storage growth and acceptable reconstruction overhead. This validates temporal provenance as a scalable design choice rather than a prohibitive one.

RQ4: What are the feasibility and implications of embedding provenance metadata directly within data artifacts versus centralised repositories?

Chapter 6 examined multiple techniques for embedding provenance, evaluating their trade-offs in terms of portability, compatibility, and integrity. The analysis showed that embedded provenance improves accessibility, reduces unintended disclosure, and aligns with established metadata practices. The prototype implementation in Chapter 7 confirms feasibility, demonstrating that embedding can be achieved without modifying application behaviour or relying on specialised provenance infrastructure.

RQ5: How do temporally aware, embedded provenance systems enable scalable lineage tracing while preserving auditability from arbitrary historical points?

Chapter 7 took the key outcomes from Chapter 4–6 and synthesised these into a unified system design, while Chapter 8 validated its behaviour under realistic workloads. The results show that temporal queries, historical reconstruction, and point-in-time audits are all supported without requiring global graph traversal or centralised analysis. This directly addresses the core hypothesis, demonstrating that temporal awareness and embedded provenance are complementary and mutually reinforcing.

9.2 Limitations

The prototype developed in this thesis operates primarily at the file system boundary. This design enables reliable capture of file level interactions while remaining application agnostic, but it necessarily constrains the scope of observable behaviour. Although limited process related context is available via the FUSE execution context, such as process identifiers, the system does not capture the full spectrum of process semantics or in-memory data transformations. This limitation arises directly from the decision to implement the prototype using FUSE. FUSE was selected to prioritise rapid experimentation, portability, and clarity of design over exhaustive coverage. The conceptual contribution of the thesis is not tied to FUSE itself, and the approach can be retrofitted into kernel-level provenance systems, such as CamFlow, where richer process and execution context is already available.

The reliance on FUSE also has performance implications. Operating in user space introduces additional context switches and data copies that would not be present in a native kernel implementation. While the evaluation shows that overhead remains bounded under the tested workloads, the prototype is not intended to represent the performance ceiling of the proposed approach. Rather, it establishes functional feasibility. A production quality implementation spanning kernel space and user space services would be expected to achieve lower overheads and finer grained interception, particularly for high frequency

operations and large scale workloads. These engineering trade-offs were consciously accepted to keep the focus on validating the provenance model and embedding strategy rather than optimising a specific implementation.

Another limitation lies in the abstraction level of the recorded provenance. The prototype captures low-level system events and versioned file transformations, but it does not attempt to infer or encode higher-level semantic intent. This choice aligns with the emphasis on the thesis being on system-level provenance as a neutral substrate. Embedding application specific meaning at this stage would risk conflating concerns and reducing generality. While this limits immediate interpretability for some domains, it preserves the ability for domain specific layers to be added without constraining the underlying representation.

Finally, the work does not fully address governance questions surrounding the long term management of embedded provenance. Embedding provenance tightly couples history to data, which strengthens auditability but complicates scenarios where provenance must be pruned, redacted, or selectively disclosed. These concerns intersect with legal, organisational, and ethical considerations that extend beyond system design. They are acknowledged as essential for real world deployment, but they are not resolved here in order to maintain focus on the technical feasibility and structural implications of the proposed framework.

9.3 Future Directions

Several avenues for future work follow naturally from the boundaries established in this thesis. A clear next step is implementation within the kernel or as an extension to existing kernel-level provenance systems. Integrating the proposed temporal and embedded provenance model into frameworks such as CamFlow would allow the approach to benefit from richer process context, stronger integrity guarantees, and reduced overhead. Such an implementation would also enable direct comparison between centralised graph-based prove-

nance and embedded, versioned provenance within the same capture infrastructure.

Extending the framework to distributed environments presents another important direction. Embedded provenance raises questions about how histories should be merged, reconciled, or validated when data is replicated across nodes or administrative domains. Exploring these issues would require engagement with distributed systems concepts such as eventual consistency, content addressing, and secure log replication, and would test the robustness of embedded provenance under realistic deployment conditions.

There is also scope to enrich the semantic layer above the embedded representation. While the current framework deliberately avoids application-level interpretation, future work could explore mechanisms for attaching optional, domain specific annotations to particular versions or transformations. This would preserve the separation between capture and interpretation while enabling richer explanations in contexts such as scientific workflows, compliance reporting, or incident investigation.

Tooling and analysis remain an open area. Embedded provenance alters the access pattern for provenance queries, favouring local inspection over global traversal. Developing query interfaces, indexing strategies, and visualisation tools that exploit this locality would be essential for usability at scale. In particular, techniques for composing provenance across multiple artifacts without reconstructing a monolithic global graph merit further study. This local-first approach can be complemented by global indexing mechanisms, as discussed in Chapter 8, to support discovery, cross-artifact linkage, and efficient entry points into distributed provenance structures.

Finally, governance and policy considerations warrant focused investigation. Mechanisms for selective disclosure, cryptographic integrity protection, and controlled modification of embedded provenance will be central to adoption in regulated and adversarial environments. Addressing these challenges would move the framework from a proof of concept towards a deployable foun-

dation for accountable data systems.

9.3.1 LLM-Assisted Provenance Analysis and Security

An interesting direction for future research is the integration of artificial intelligence (AI), particularly large language models (LLMs), into the proposed provenance framework to support automated security analysis and threat detection. While this research has demonstrated the feasibility of capturing temporal data evolution and embedding provenance directly within files, the resulting provenance records can be used for identifying suspicious activities. Contemporary cybersecurity applications increasingly employ AI and machine learning to analyse system logs [216], detect anomalous behaviour [217] and correlate security events [218]. Extending these capabilities to provenance information could enable the identification of unauthorised file modifications, unexpected dependency changes, provenance tampering, and possible software supply chain attacks. Unlike conventional log-based approaches, the proposed framework captures both the historical evolution of data and the causal relationships between transformations, potentially providing AI models with richer contextual information for distinguishing legitimate operations from adversarial behaviour.

A further possibility is to employ LLMs to bootstrap the provenance analysis process, reducing the need for extensive manually labelled training datasets or predefined detection rules. For example, an LLM could initially examine provenance records, filesystem events, historical deltas, and associated log data to identify potentially suspicious sequences of operations and generate preliminary security annotations. These annotations could subsequently be validated by security analysts and used to develop more specialised detection models. Such an approach could also support natural language interrogation of provenance histories, allowing users to investigate how a particular file was modified, which processes contributed to its current state, or whether an observed change indicates a possible security weakness. However, the reliability of AI generated

assessments, including false positives, hallucinations, and the privacy implications of exposing sensitive metadata to external models, would require careful investigation. Future work could therefore evaluate whether combining verifiable, temporally aware provenance with AI assisted reasoning improves the accuracy, and practicality of security analysis, while preserving the integrity and accountability guarantees central to this research.

9.4 Final Remarks

Data provenance is often discussed in terms of scale, completeness, and formal models. This thesis argues that usability, temporal precision, and proximity to data are equally important. By embedding versioned provenance within data artifacts themselves, the work takes a step towards making provenance a routine and inspectable part of everyday data use rather than a specialised forensic capability. If provenance is to fulfil its promise as a foundation for trust, accountability, and understanding in digital systems, it must be designed not only to record what happened, but to be consulted, interpreted, and acted upon when it matters.

Appendix A

Supplementary Results:

Retention vs. Delta Versioning

version	file_size	delta_size	total_size	delta_sum
2	5826	380	11479	380
3	9074	2516	20553	2896
4	8962	179	29515	3075
5	8903	178	38418	3253
6	8778	341	47196	3594
7	8787	176	55983	3770
8	8736	237	64719	4007
9	8767	204	73486	4211
10	8769	183	82255	4394
11	8723	151	90978	4545
12	8952	744	99930	5289
13	8886	156	108816	5445
14	8937	256	117753	5701
15	8335	215	126088	5916
16	7903	270	133991	6186
17	7694	307	141685	6493

18	6420	1189	148105	7682
19	6400	190	154505	7872
20	6468	243	160973	8115
21	6495	178	167468	8293
22	7164	571	174632	8864
23	7533	413	182165	9277
24	10792	1588	192957	10865
25	10912	333	203869	11198
26	10702	284	214571	11482
27	10796	250	225367	11732
28	10989	317	236356	12049
29	11200	336	247556	12385
30	12356	819	259912	13204
31	12381	180	272293	13384
32	12385	180	284678	13564
33	12361	187	297039	13751
34	12278	163	309317	13914
35	12019	498	321336	14412
36	12242	393	333578	14805
37	12236	259	345814	15064
38	12045	352	357859	15416
39	12045	153	369904	15569
40	12242	551	382146	16120
41	12045	435	394191	16555
42	12071	218	406262	16773
43	12104	244	418366	17017
44	12235	281	430601	17298
45	14151	1095	444752	18393
46	14041	282	458793	18675

47	13751	228	472544	18903
48	14961	735	487505	19638
49	15407	478	502912	20116
50	17893	1070	520805	21186
51	18460	385	539265	21571
52	19250	483	558515	22054
53	22443	1488	580958	23542
54	23045	444	604003	23986
55	23467	303	627470	24289
56	23481	175	650951	24464
57	23959	380	674910	24844
58	24063	259	698973	25103
59	24090	196	723063	25299
60	24140	212	747203	25511
61	24138	181	771341	25692
62	24098	193	795439	25885
63	24096	181	819535	26066
64	23992	152	843527	26218
65	24123	326	867650	26544
66	24019	151	891669	26695
67	24009	148	915678	26843
68	24010	180	939688	27023
69	22444	258	962132	27281
70	22448	181	984580	27462
71	22277	269	1006857	27731
72	22113	307	1028970	28038
73	21843	294	1050813	28332
74	21630	213	1072443	28545
75	21644	182	1094087	28727

76	21643	187	1115730	28914
77	22266	712	1137996	29626
78	22083	239	1160079	29865
79	22082	192	1182161	30057
80	22091	329	1204252	30386
81	22090	192	1226342	30578
82	21990	198	1248332	30776
83	21997	184	1270329	30960
84	22020	176	1292349	31136
85	22027	182	1314376	31318
86	22034	189	1336410	31507
87	22006	203	1358416	31710
88	22775	787	1381191	32497
89	22905	324	1404096	32821
90	24013	714	1428109	33535
91	24053	207	1452162	33742
92	23947	568	1476109	34310
93	23947	156	1500056	34466
94	24027	256	1524083	34722
95	24099	231	1548182	34953
96	24199	335	1572381	35288
97	24206	178	1596587	35466
98	24190	148	1620777	35614
99	23797	244	1644574	35858
100	23833	215	1668407	36073
101	23764	154	1692171	36227
102	23836	228	1716007	36455
103	23872	214	1739879	36669
104	23865	148	1763744	36817

105	23737	199	1787481	37016
106	23730	148	1811211	37164
107	23913	295	1835124	37459
108	23963	219	1859087	37678
109	23961	179	1883048	37857
110	24591	332	1907639	38189
111	24589	181	1932228	38370
112	24730	300	1956958	38670
113	24660	272	1981618	38942
114	24801	291	2006419	39233
115	24798	187	2031217	39420
116	24801	188	2056018	39608
117	24736	161	2080754	39769
118	24801	218	2105555	39987
119	25159	350	2130714	40337
120	24867	302	2155581	40639
121	25225	358	2180806	40997
122	25340	246	2206146	41243
123	25334	178	2231480	41421
124	26252	644	2257732	42065
125	27517	925	2285249	42990
126	26422	861	2311671	43851
127	26502	291	2338173	44142
128	26746	370	2364919	44512
129	25269	189	2390188	44701
130	28929	1966	2419117	46667
131	28864	165	2447981	46832
132	28924	221	2476905	47053
133	29498	470	2506403	47523

134	28703	193	2535106	47716
135	28708	200	2563814	47916
136	28731	199	2592545	48115
137	28745	198	2621290	48313
138	29056	270	2650346	48583
139	29506	322	2679852	48905
140	29052	191	2708904	49096
141	29088	244	2737992	49340
142	28959	189	2766951	49529
143	28887	222	2795838	49751
144	28883	158	2824721	49909
145	28667	153	2853388	50062
146	29241	472	2882629	50534
147	29406	296	2912035	50830
148	29346	157	2941381	50987
149	29599	362	2970980	51349
150	30525	799	3001505	52148
151	29960	229	3031465	52377
152	30436	731	3061901	53108
153	29580	559	3091481	53667
154	29941	410	3121422	54077
155	30417	718	3151839	54795
156	30076	515	3181915	55310
157	30552	718	3212467	56028
158	30649	231	3243116	56259
159	31709	885	3274825	57144
160	33133	1026	3307958	58170
161	33113	197	3341071	58367
162	32886	244	3373957	58611

163	32414	482	3406371	59093
164	32391	148	3438762	59241
165	32431	174	3471193	59415
166	32779	388	3503972	59803
167	32823	172	3536795	59975
168	32787	161	3569582	60136
169	32921	350	3602503	60486
170	33301	704	3635804	61190
171	33294	204	3669098	61394
172	32958	388	3702056	61782
173	33429	523	3735485	62305
174	35596	1430	3771081	63735
175	35666	226	3806747	63961
176	36302	333	3843049	64294
177	36971	720	3880020	65014
178	37045	288	3917065	65302
179	36993	312	3954058	65614
180	36885	173	3990943	65787
181	36799	168	4027742	65955
182	36915	231	4064657	66186
183	36853	183	4101510	66369
184	36893	209	4138403	66578
185	36937	161	4175340	66739
186	36930	387	4212270	67126
187	36857	538	4249127	67664
188	36907	574	4286034	68238
189	36871	350	4322905	68588
190	36411	859	4359316	69447
191	37248	1449	4396564	70896

192	35974	1059	4432538	71955
193	35960	182	4468498	72137
194	36053	291	4504551	72428
195	35936	275	4540487	72703
196	35982	177	4576469	72880
197	36228	322	4612697	73202
198	35855	782	4648552	73984
199	35648	310	4684200	74294
200	36729	1660	4720929	75954
201	35632	1324	4756561	77278
202	35421	243	4791982	77521
203	38427	2631	4830409	80152
204	38375	276	4868784	80428
205	37385	1347	4906169	81775
206	37703	381	4943872	82156
207	37713	188	4981585	82344
208	37732	222	5019317	82566
209	37060	265	5056377	82831
210	37089	191	5093466	83022
211	37334	306	5130800	83328
212	37198	247	5167998	83575
213	37435	395	5205433	83970
214	37844	622	5243277	84592
215	37299	593	5280576	85185
216	38475	1478	5319051	86663
217	38685	346	5357736	87009
218	38701	196	5396437	87205
219	38791	305	5435228	87510

Appendix B

Implementation of Git-Based File Reconstruction Utility

```
#!/bin/bash

set -euo pipefail

REPO_PATH="$1"
FILE_PATH="$2"
DEST_DIR="$3"

if [ ! -d "$REPO_PATH/.git" ]; then
    echo "Error: '$REPO_PATH' is not a Git repository."
    exit 1
fi

if [ ! -d "$DEST_DIR" ]; then
    echo "Creating destination directory: $DEST_DIR"
    mkdir -p "$DEST_DIR"
fi

# Get relative path of the file from the repo root
cd "$REPO_PATH"
REL_PATH=$(realpath --relative-to="$REPO_PATH" "$FILE_PATH")

# Get all commits that touched the file, oldest first
COMMITS=$(git log --reverse --format=%H -- "$REL_PATH")

if [ -z "$COMMITS" ]; then
    echo "No commits found for file: $REL_PATH"
    exit 1
```

```
fi

VERSION=1
for COMMIT in $COMMITS; do
    echo "Processing commit $COMMIT"
    OUT_FILE="$(basename "$REL_PATH")"
    git show "${COMMIT}:${REL_PATH}" > "$DEST_DIR/$OUT_FILE"
    echo "Exported version $VERSION to $DEST_DIR/$OUT_FILE"
    ((VERSION++))
done
```

References

[1] "Open Provenance Model (OPM)," http://openprovenance.org/opm/, accessed: 2023-06-21.

[2] K.-K. Muniswamy-Reddy, J. Barillari, U. Braun, D. A. Holland, D. Maclean, M. I. Seltzer, and S. D. Holland, "Layering in provenance-aware storage systems," 2008.

[3] R. P. Spillane, R. Sears, C. Yalamanchili, S. Gaikwad, M. Chinni, and E. Zadok, "Story book: An efficient extensible provenance framework." in *Workshop on the Theory and Practice of Provenance*, 2009.

[4] A. Gehani and D. Tariq, "Spade: Support for provenance auditing in distributed environments," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2012, pp. 101–120.

[5] C. H. Suen, R. K. Ko, Y. S. Tan, P. Jagadpramana, and B. S. Lee, "S2logger: End-to-end data tracking mechanism for cloud data provenance," in *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2013, pp. 594–602.

[6] R. K. Ko and M. A. Will, "Progger: An efficient, tamper-evident kernel-space logger for cloud data provenance tracking," in *2014 IEEE 7th International Conference on Cloud Computing*. IEEE, 2014, pp. 881–889.

[7] A. Bates, D. J. Tian, K. R. Butler, and T. Moyer, "Trustworthy whole-system provenance for the Linux kernel," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 319–334.

[8] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, "Practical whole-system provenance capture," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, pp. 405–418.

[9] T. J. C. Corrick, "Progger 3: A low-overhead, tamper-proof provenance system," 2021.

[10] Exiv2 Project, "The metadata in tiff files," https://dev.exiv2.org/projects/exiv2/wiki/The_Metadata_in_TIFF_files, accessed: 18 September 2026.

[11] B. Pan, N. Stakhanova, and S. Ray, "Data provenance in security and privacy," *ACM Computing Surveys*, vol. 55, no. 14s, pp. 1–35, 2023.

[12] O. I. Abiodun, M. Alawida, A. E. Omolara, and A. Alabdulatif, "Data provenance for cloud forensic investigations, security, challenges, solutions and future perspectives: A survey," *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 10, pp. 10 217–10 245, 2022.

[13] F. Guitton, A. Oehmichen, É. Bossé, and Y. Guo, "Honest computing: achieving demonstrable data lineage and provenance for driving data and process-sensitive policies," *Data & Policy*, vol. 6, p. e84, 2024.

[14] Y. L. Simmhan, B. Plale, and D. Gannon, "A survey of data provenance techniques," *Computer Science Department, Indiana University, Bloomington IN*, vol. 47405, p. 69, 2005.

[15] M. Johns, T. Meurers, F. N. Wirth, A. C. Haber, A. Müller, M. Halilovic, F. Balzer, and F. Prasser, "Data provenance in biomedical research: scoping review," *Journal of medical Internet research*, vol. 25, p. e42289, 2023.

[16] T. Moyer, K. Chadha, R. Cunningham, N. Schear, W. Smith, A. Bates, K. Butler, F. Capobianco, T. Jaeger, and P. Cable, "Leveraging data provenance to enhance cyber resilience," in *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 2016, pp. 107–114.

[17] K. Nithya, M. Selvan, V. Gnanavel, P. Solainayagi, and K. Venkatesh, "Leveraging bayesian networks for enhanced transparency in financial data provenance tracking," in *2024 International Conference on Cybernation and Computation (CYBERCOM)*. IEEE, 2024, pp. 671–676.

[18] K. Werder, B. Ramesh, and R. Zhang, "Establishing data provenance for responsible artificial intelligence systems," *ACM Transactions on Management Information Systems (TMIS)*, vol. 13, no. 2, pp. 1–23, 2022.

[19] J. Soldatos, A. Despotopoulou, N. Kefalakis, and B. Ipektsidis, "Blockchain based data provenance for trusted artificial intelligence,"

Trusted Artificial Intelligence in Manufacturing: A Review of the Emerging Wave of Ethical and Human Centric AI Technologies for Smart Production, pp. 1–29, 2021.

[20] J. Singh, J. Cobbe, and C. Norval, "Decision provenance: Harnessing data flow for accountable systems," *IEEE Access*, vol. 7, pp. 6562–6574, 2018.

[21] J. Freire, D. Koop, E. Santos, and C. T. Silva, "Provenance for computational tasks: A survey," *Computing in science & engineering*, vol. 10, no. 3, pp. 11–21, 2008.

[22] P. Datta, I. Polinsky, M. A. Inam, A. Bates, and W. Enck, "{ALASTOR}: Reconstructing the provenance of serverless intrusions," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2443–2460.

[23] J. Zeng, C. Zhang, and Z. Liang, "Palantír: Optimizing attack provenance with hardware-enhanced system observability," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 3135–3149.

[24] M. Zipperle, F. Gottwalt, E. Chang, and T. Dillon, "Provenance-based intrusion detection systems: A survey," *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–36, 2022.

[25] S. B. Davidson and J. Freire, "Provenance and scientific workflows: challenges and opportunities," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 1345–1350.

[26] V. Cuevas-Vicenttín, S. Dey, S. Köhler, S. Riddle, and B. Ludäscher, "Scientific workflows and provenance: Introduction and research opportunities," *Datenbank-Spektrum*, vol. 12, no. 3, pp. 193–203, 2012.

[27] R. Souza, L. Azevedo, V. Lourenço, E. Soares, R. Thiago, R. Brandão, D. Civitarese, E. Brazil, M. Moreno, P. Valduriez *et al.*, "Provenance data in the machine learning lifecycle in computational science and engineering," in *2019 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*. IEEE, 2019, pp. 1–10.

[28] S. Samuel, F. Löffler, and B. König-Ries, "Machine learning pipelines: provenance, reproducibility and fair data principles," in *International Provenance and Annotation Workshop*. Springer, 2020, pp. 226–230.

[29] T. Pasquier, D. Eyers, and M. Seltzer, "From here to Provtopia," in *VLDB Workshop on Data Management and Analytics for Medicine and Healthcare*. Springer, 2019, pp. 54–67.

[30] S. Bowers, T. M. McPhillips, and B. Ludäscher, "Provenance in collection-oriented scientific workflows," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, pp. 519–529, 2008.

[31] W. C. Tan, "Provenance in databases: Past, current, and future." *IEEE Data Eng. Bull.*, vol. 30, no. 4, pp. 3–12, 2007.

[32] D. Tang, R. Zhao, Y. Lin, T. Zhang, and P. Zhang, "Modeling the data provenance of relational databases supporting full-featured sql and procedural languages," *Applied Sciences*, vol. 13, no. 1, p. 64, 2022.

[33] D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, "Hi-fi: collecting high-fidelity whole-system provenance," in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012, pp. 259–268.

[34] K. Eckert, "Provenance and annotations for linked data," in *Proceedings of the International Conference on Dublin Core and Metadata Applications*. Dublin Core Metadata Initiative, 2013.

[35] D. P. Groth and K. Streefkerk, "Provenance and annotation for visual exploration systems," *IEEE transactions on visualization and computer graphics*, vol. 12, no. 6, pp. 1500–1510, 2006.

[36] K. Cheung and J. Hunter, "Provenance explorer–customized provenance views using semantic inferencing," in *International Semantic Web Conference*. Springer, 2006, pp. 215–227.

[37] P. Buneman, S. Khanna, and T. Wang-Chiew, "Why and where: A characterization of data provenance," in *Database Theory—ICDT 2001: 8th International Conference London, UK, January 4–6, 2001 Proceedings 8*. Springer, 2001, pp. 316–330.

[38] P. Groth and L. Moreau, "Prov-overview," *W3C Working Group Note*, vol. 1135, pp. 881–906, 2013.

[39] M. Greenwood, C. Goble, R. Stevens, J. Zhao, M. Addis, D. Marvin, L. Moreau, and T. Oinn, "Provenance of e-science experiments-experience from bioinformatics," 2003.

[40] Q. Pham, T. Malik, and I. Foster, "Using provenance for repeatability," in *5th USENIX Workshop on the Theory and Practice of Provenance (TaPP 13)*, 2013.

[41] Y. Xie, D. Feng, Z. Tan, and J. Zhou, "Unifying intrusion detection and forensic analysis via provenance awareness," *Future Generation Computer Systems*, vol. 61, pp. 26–36, 2016.

[42] H. J. Pandit, D. O'Sullivan, and D. Lewis, "Queryable provenance metadata for gdpr compliance," *Procedia Computer Science*, vol. 137, pp. 262–268, 2018.

[43] B. E. Ujcich, A. Bates, and W. H. Sanders, "A provenance model for the european union general data protection regulation," in *International Provenance and Annotation Workshop*. Springer, 2018, pp. 45–57.

[44] F. T. Jaigirdar, C. Rudolph, and C. Bain, "Risk and compliance in iot-health data propagation: A security-aware provenance based approach," in *2021 IEEE International Conference on Digital Health (ICDH)*. IEEE, 2021, pp. 27–37.

[45] D. Stodder, "Reducing time to insight and maximizing the benefits of real-time data," TDWI Research, Best Practices Report, 2024.

[46] EDM Association, "Global data management benchmark report," EDM Association, Benchmark Report, May 2026.

[47] R. Hu, Z. Yan, W. Ding, and L. T. Yang, "A survey on data provenance in iot," *World Wide Web*, vol. 23, no. 2, pp. 1441–1463, 2020.

[48] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research," *Journal of management information systems*, vol. 24, no. 3, pp. 45–77, 2007.

[49] K. Gierend, J. A. Wodke, S. Genehr, R. Gött, R. Henkel, F. Krüger, M. Mandalka, L. Michaelis, A. Scheuerlein, and M. Schröder, "Tapp: Defining standard provenance information for clinical research data and workflows-obstacles and opportunities," in *Companion Proceedings of the ACM Web Conference 2023*, 2023, pp. 1551–1554.

[50] S. Ram and J. Liu, "A new perspective on semantics of data provenance," 2009.

[51] L. Moreau, N. Hogan, and N. O'Donnell, "Implementing an environmental management system using provenance-by-design," in *Companion Proceedings of the ACM Web Conference 2023*, 2023, pp. 1531–1541.

[52] M. Ahmed, A. R. Dar, M. Helfert, A. Khan, and J. Kim, "Data provenance in healthcare: approaches, challenges, and future directions," *Sensors*, vol. 23, no. 14, p. 6495, 2023.

[53] M. J. Sembay, D. D. J. de Macedo, L. P. Júnior, R. M. M. Braga, and A. Sarasa-Cabezuelo, "Provenance data management in health information systems: a systematic literature review," *Journal of Personalized Medicine*, vol. 13, no. 6, p. 991, 2023.

[54] E. Gultekin and M. S. Aktas, "Systematic literature review on data provenance in internet of things," in *International Conference on Computational Science and Its Applications*. Springer, 2022, pp. 31–46.

[55] T. Auge, S. Genehr, F. Krüger, and M. Schröder, "Towards dimensions and granularity in a unified workflow and data provenance framework," *arXiv preprint arXiv:2504.11278*, 2025.

[56] R. B. Webjornsen, "Discovering data lineage in data warehouse," 2005.

[57] K. T. P. Järv and T. Tammet, "Computing data lineage and business," in *Knowledge Discovery, Knowledge Engineering and Knowledge Management: 8th International Joint Conference, IC3K 2016, Porto, Portugal, November 9–11, 2016, Revised Selected Papers*, vol. 914. Springer, 2018, p. 101.

[58] M. Herschel, R. Diestelkämper, and H. Ben Lahmar, "A survey on provenance: What for? what form? what from?" *The VLDB Journal*, vol. 26, no. 6, pp. 881–906, 2017.

[59] Q. Zhang, Y. Cao, Q. Wang, D. Vu, P. Thavasimani, T. McPhillips, P. Missier, P. Slaughter, C. Jones, and M. B. Jones, "Revealing the detailed lineage of script outputs using hybrid provenance," *International Journal of Digital Curation*, vol. 12, no. 2, pp. 390–408, 2017.

[60] I. Foster, J. Vockler, M. Wilde, and Y. Zhao, "Chimera: A virtual data system for representing, querying, and automating data derivation," in *Proceedings 14th International Conference on Scientific and Statistical Database Management*. IEEE, 2002, pp. 37–46.

[61] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: an extensible system for design and execution of scientific workflows," in *Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004.* IEEE, 2004, pp. 423–424.

[62] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, and P. Fisher, "The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud," *Nucleic acids research*, vol. 41, no. W1, pp. W557–W561, 2013.

[63] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, and J. Myers, "The open provenance model core specification (v1. 1)," *Future generation computer systems*, vol. 27, no. 6, pp. 743–756, 2011.

[64] J. Cheney, S. Chong, N. Foster, M. Seltzer, and S. Vansummeren, "Provenance: a future history," in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, 2009, pp. 957–964.

[65] N. Kwasnikowska, L. Moreau, and J. V. D. Bussche, "A formal account of the open provenance model," *ACM Transactions on the Web (TWEB)*, vol. 9, no. 2, pp. 1–44, 2015.

[66] L. Moreau and P. Groth, "Prov-overview: An overview of the prov family of documents," World Wide Web Consortium (W3C), W3C Note, Apr. 2013, w3C Note 30 April 2013. [Online]. Available: https://www.w3.org/TR/prov-overview/

[67] K. Belhajjame, R. B'Far, J. Cheney, S. Coppens, S. Cresswell, Y. Gil, P. Groth, G. Klyne, T. Lebo, and J. McCusker, "Prov-dm: The prov data model," *W3C Recommendation*, vol. 14, pp. 15–16, 2013.

[68] L. Moreau and P. Missier, "PROV-N: The Provenance Notation," World Wide Web Consortium (W3C) Recommendation, Apr. 2013. [Online]. Available: http://www.w3.org/TR/prov-n/

[69] H. Hua, C. Tilmes, S. Zednik, and L. Moreau, "PROV-XML: The PROV XML Schema," World Wide Web Consortium (W3C) Note, Apr. 2013. [Online]. Available: http://www.w3.org/TR/prov-xml/

[70] T. D. Huynh, M. O. Jewell, A. Sezavar Keshavarz, D. T. Michaelides, H. Yang, and L. Moreau, "The prov-json serialization," 2013.

[71] T. Lebo, S. Sahoo, D. McGuinness, K. Belhajjame, J. Cheney, D. Corsar, D. Garijo, S. Soiland-Reyes, S. Zednik, and J. Zhao, "PROV-O: The PROV Ontology," World Wide Web Consortium (W3C) Recommendation, Apr. 2013. [Online]. Available: http://www.w3.org/TR/2013/REC-prov-o-20130430/

[72] L. Moreau, O. Hartig, Y. Simmhan, J. Myers, T. Lebo, K. Belhajjame, and S. Miles, "Prov-aq: provenance access and query," 2013.

[73] R. Han, S. Byna, H. Tang, B. Dong, and M. Zheng, "Prov-io: An i/o-centric provenance framework for scientific data on hpc systems," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, 2022, pp. 213–226.

[74] Y. Cao, C. Jones, V. Cuevas-Vicenttín, M. B. Jones, B. Ludäscher, T. McPhillips, P. Missier, C. Schwalm, P. Slaughter, and D. Vieglais, "Dataone: a data federation with provenance support," in *International Provenance and Annotation Workshop*. Springer, 2016, pp. 230–234.

[75] P. Missier, S. Dey, K. Belhajjame, V. Cuevas-Vicenttín, and B. Ludäscher, "{D-PROV}: Extending the {PROV} provenance model with {Workflow} structure," in *5th USENIX Workshop on the Theory and Practice of Provenance (TaPP 13)*, 2013.

[76] F. A. Bhuyan, S. Lu, D. Ruan, and J. Zhang, "Scalable provenance storage and querying using pig latin for big data workflows," in *2017 IEEE International Conference on Services Computing (SCC)*. IEEE, 2017, pp. 459–466.

[77] A. Chapman, L. Lauro, P. Missier, and R. Torlone, "Supporting better insights of data science pipelines with fine-grained provenance," *ACM Transactions on Database Systems*, vol. 49, no. 2, pp. 1–42, 2024.

[78] J. G. Levine, J. B. Grizzard, P. W. Hutto, and H. L. Owen, "A methodology to characterize kernel level rootkit exploits that overwrite the system call table," in *IEEE SoutheastCon, 2004. Proceedings.* IEEE, 2004, pp. 25–31.

[79] S. Smalley, C. Vance, and W. Salamon, "Implementing SELinux as a Linux security module," *NAI Labs Report*, vol. 1, no. 43, p. 139, 2001.

[80] Canonical Ltd., "Apparmor security project," https://gitlab.com/apparmor/apparmor, 2024, accessed: 2024-04-28.

[81] B. Gregg, "Bpf internals," Slides presented at USENIX LISA 2021, 2021, available online at https://www.brendangregg.com/Slides/LISA2021_BPF_Internals.pdf.

[82] S. W. Zhong, J. Liu, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Revealing the unstable foundations of ebpf-based kernel extensions," in

Proceedings of the Twentieth European Conference on Computer Systems, 2025, pp. 21–41.

[83] P. Macko and M. Chiarini, "Collecting provenance via the xen hypervisor," in *3rd USENIX Workshop on the Theory and Practice of Provenance (TaPP 11)*, 2011.

[84] S. Ma, X. Zhang, and D. Xu, "Protracer: Towards practical provenance tracing by alternating between logging and tainting," in *23rd Annual Network And Distributed System Security Symposium (NDSS 2016)*. Internet Soc, 2016.

[85] Y. Cui and J. Widom, "Practical lineage tracing in data warehouses," in *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*. IEEE, 2000, pp. 367–378.

[86] I. Neo4j, "Neo4j graph database & analytics," 2025, accessed: 2023-05-01. [Online]. Available: https://neo4j.com/

[87] D. Sunuwar and M. Singh, "Comparative analysis of relational and graph databases for data provenance: Performance, queries, and security considerations," in *2023 World Conference on Communication & Computing (WCONF)*. IEEE, 2023, pp. 1–7.

[88] R. K. Ko, P. Jagadpramana, and B. S. Lee, "Flogger: A file-centric logger for monitoring file access and transfers within cloud computing environments," in *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2011, pp. 765–771.

[89] O. Setayeshfar, C. Adkins, M. Jones, K. H. Lee, and P. Doshi, "Graalf: Supporting graphical analysis of audit logs for forensics." *Softw. Impacts*, vol. 8, p. 100068, 2021.

[90] Y. S. Tan, "Reconstructing data provenance from log files," Ph.D. dissertation, The University of Waikato, 2017.

[91] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, and A. Wipat, "Taverna: a tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, no. 17, pp. 3045–3054, 2004.

[92] T. De Nies, S. Magliacane, R. Verborgh, S. Coppens, P. Groth, E. Mannens, and R. Van de Walle, "Git2prov: Exposing version control system content as w3c prov." in *ISWC (Posters & Demos)*, 2013, pp. 125–128.

[93] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer, "Provenance-aware storage systems." in *Usenix annual technical conference, general track*, 2006, pp. 43–56.

[94] K.-K. Muniswamy-Reddy, P. Macko, and M. Seltzer, "Provenance for the cloud," in *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, ser. FAST'10. USA: USENIX Association, 2010, p. 15–14.

[95] M. A. Halcrow, "ecryptfs: An enterprise-class encrypted filesystem for Linux," in *Proceedings of the 2005 Linux Symposium*, vol. 1, 2005, pp. 201–218.

[96] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener, "The lorel query language for semistructured data," *International journal on digital libraries*, vol. 1, no. 1, pp. 68–88, 1997.

[97] J. Katcher, "Postmark: A new file system benchmark," *TR3022*, 1997.

[98] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, "Gapped blast and psi-blast: a new generation of protein database search programs," *Nucleic acids research*, vol. 25, no. 17, pp. 3389–3402, 1997.

[99] R. Sears and E. Brewer, "Stasis: Flexible transactional storage," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 29–44.

[100] R. P. Spillane, S. Gaikwad, M. Chinni, E. Zadok, and C. P. Wright, "Enabling transactional file access via lightweight kernel extensions." in *FAST*, vol. 9, 2009, pp. 29–42.

[101] R. Sears, M. Callaghan, and E. Brewer, "Rose: Compressed, log-structured replication," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 526–537, 2008.

[102] "Tpc benchmark c standard specification, revision 5.11," Transaction Processing Performance Council (TPC), Tech. Rep., February 2010. [Online]. Available: http://www.tpc.org/tpcc/

[103] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, 2001.

[104] G. H. Kim and E. H. Spafford, "The design and implementation of tripwire: A file system integrity checker," in *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, 1994, pp. 18–29.

[105] R. K. Ko, P. Jagadpramana, M. Mowbray, S. Pearson, M. Kirchberg, Q. Liang, and B. S. Lee, "Trustcloud: A framework for accountability and trust in cloud computing," in *2011 IEEE World Congress on Services*. IEEE, 2011, pp. 584–588.

[106] M. Stonebraker and L. A. Rowe, "The design of postgres," *ACM Sigmod Record*, vol. 15, no. 2, pp. 340–355, 1986.

[107] P. A. Boncz, M. Zukowski, and N. Nes, "Monetdb/x100: Hyper-pipelining query execution." in *Cidr*, vol. 5, 2005, pp. 225–237.

[108] "Novel information gathering and harvesting techniques for intelligence in global autonomous language exploitation," http://www.speech.sri.com/projects/GALE/, SRI International, Speech Technology and Research Laboratory, n.d., dARPA GALE project page. Accessed: 2021-04-27.

[109] "The llvm compiler infrastructure project," https://llvm.org/, 2023, accessed: 2023-03-12.

[110] The Apache Software Foundation, "Apache lucene: A high-performance, full-featured text search engine library," https://lucene.apache.org/, 2025, accessed: 2023-05-01.

[111] "Graphviz — graph visualization software," https://graphviz.org/, 2025, accessed: 2023-05-01.

[112] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "Openstack: toward an open-source solution for cloud computing," *International Journal of Computer Applications*, vol. 55, no. 3, 2012.

[113] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 545–554.

[114] Y. Etsion, D. Tsafrir, S. Kirkpatrick, and D. G. Feitelson, "Fine grained kernel logging with klogger: Experience and insights," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, 2007, pp. 259–272.

[115] T. Pasquier and X. M. Han, "Camflow/camflow-query: v0.1.0," May 2017. [Online]. Available: https://doi.org/10.5281/zenodo.571433

[116] T. Zanussi, K. Yaghmour, R. Wisniewski, R. Moore, and M. Dagenais, "relayfs: An efficient unified approach for transmitting data from kernel to user space," in *Linux Symposium*, vol. 494, 2003.

[117] S. Hoffman, *Apache Flume: distributed log collection for Hadoop*. Packt Publishing Ltd, 2013.

[118] O. Standard, "MQTT version 3.1.1," *URL http://docs. oasis-open. org/mqtt/mqtt/v3*, vol. 1, p. 29, 2014.

[119] L. W. McVoy and C. Staelin, "Lmbench: Portable tools for performance analysis." in *USENIX annual technical conference*. San Diego, CA, USA, 1996, pp. 279–294.

[120] T. Pasquier, X. Han, T. Moyer, A. Bates, O. Hermant, D. Eyers, J. Bacon, and M. Seltzer, "Runtime analysis of whole-system provenance," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 1601–1616.

[121] S. Y. Lim, B. Stelea, X. Han, and T. Pasquier, "Secure namespaced kernel audit for containers," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 518–532.

[122] "cilium/cilium: ebpf-based networking, security, and observability," https://github.com/cilium/cilium, 2024, accessed: 2024-04-28.

[123] "Falco — event sources / drivers," https://falco.org/docs/event-sources/drivers/, 2024, accessed: 2024-04-28.

[124] "Tracee: ebpf-based runtime security and forensics tool," https://github.com/aquasecurity/tracee, 2024, accessed: 2024-04-28.

[125] F. Curbera, Y. Doganata, A. Martens, N. K. Mukhi, and A. Slominski, "Business provenance–a technology to increase traceability of end-to-end operations," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 2008, pp. 100–119.

[126] T. Pasquier, J. Singh, J. Powles, D. Eyers, M. Seltzer, and J. Bacon, "Data provenance to audit compliance with privacy policy in the internet of things," *Personal and Ubiquitous Computing*, vol. 22, pp. 333–344, 2018.

[127] J. Tyndall, "Aacods checklist. 2010," *URL: https://dspace. flinders. edu. au/xmlui/bitstream/handle/2328/3326/AACODS_Checklist. pdf [accessed 2026-09-12]*, 2021.

[128] S. Shah, C. A. Soules, G. R. Ganger, and B. D. Noble, "Using provenance to aid in personal file search." in *USENIX Annual Technical Conference*, 2007, pp. 171–184.

[129] D. P. Campagna, A. S. da Silva, and V. Braganholo, "Achieving gdpr compliance through provenance: An extended model," in *Simpósio Brasileiro de Banco de Dados (SBBD)*. SBC, 2020, pp. 13–24.

[130] M. Talha, M. Ahmed, and A. Khan, "Towards semantic blockchain for providing provenance for gdpr compliance," in *Towards semantic blockchain for providing provenance for GDPR compliance*.

[131] M. Davari and E. Bertino, "Access control model extensions to support data privacy protection based on gdpr," in *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 2019, pp. 4017–4024.

[132] F. Zafar, A. Khan, S. Suhail, I. Ahmed, K. Hameed, H. M. Khan, F. Jabeen, and A. Anjum, "Trustworthy data: A survey, taxonomy and future trends of secure provenance schemes," *Journal of network and computer applications*, vol. 94, pp. 50–68, 2017.

[133] S. Sultana, M. Shehab, and E. Bertino, "Secure provenance transmission for streaming data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 8, pp. 1890–1903, 2012.

[134] R. Lu, X. Lin, X. Liang, and X. Shen, "Secure provenance: the essential of bread and butter of data forensics in cloud computing," in *Proceedings of the 5th ACM symposium on information, computer and communications security*, 2010, pp. 282–292.

[135] J. Reuben, L. A. Martucci, S. Fischer-Hübner, H. S. Packer, H. Hedbom, and L. Moreau, "Privacy impact assessment template for provenance," in *2016 11th International Conference on Availability, Reliability and Security (ARES)*. IEEE, 2016, pp. 653–660.

[136] R. Hasan, R. Sion, and M. Winslett, "Introducing secure provenance: problems and challenges," in *Proceedings of the 2007 ACM workshop on Storage security and survivability*, 2007, pp. 13–18.

[137] ——, "Preventing history forgery with secure provenance," *ACM Transactions on Storage (TOS)*, vol. 5, no. 4, pp. 1–43, 2009.

[138] F. T. Jaigirdar, C. Rudolph, and C. Bain, "Prov-iot: a security-aware iot provenance model," in *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2020, pp. 1360–1367.

[139] J. Tharaud, S. Wohlgemuth, I. Echizen, N. Sonehara, G. Müller, and P. Lafourcade, "Privacy by data provenance with digital watermarking-a proof-of-concept implementation for medical services with electronic health records," in *2010 Sixth International Conference on Intelligent Information Hiding and Multimedia Signal Processing*. IEEE, 2010, pp. 510–513.

[140] M. J. Velez De Villa Rojas, "Provenance information in ehealth," Master's thesis, Universitat Politècnica de Catalunya, 2024.

[141] P. Missier, K. Belhajjame, and J. Cheney, "The w3c prov family of specifications for modelling provenance metadata," in *Proceedings of the 16th international conference on extending database technology*, 2013, pp. 773–776.

[142] S. Jain and H. M. Krumholz, "Patient privacy and data provenance in pulmonary and critical care research using big data," *Annals of the American Thoracic Society*, vol. 21, no. 4, pp. 538–540, 2024.

[143] M. R. Mehra, S. S. Desai, S. Kuy, T. D. Henry, and A. N. Patel, "Cardiovascular disease, drug therapy, and mortality in covid-19," *New England Journal of Medicine*, vol. 382, no. 25, p. e102, 2020.

[144] M. R. Mehra, F. Ruschitzka, and A. N. Patel, "Retraction—hydroxychloroquine or chloroquine with or without a macrolide for treatment of covid-19: a multinational registry analysis," *The lancet*, vol. 395, no. 10240, p. 1820, 2020.

[145] S.-O. Act, "Sarbanes-oxley act," *Washington DC*, 2002.

[146] H. P. Kothandapani, "Optimizing financial data governance for improved risk management and regulatory reporting in data lakes," *International Journal of Applied Machine Learning and Computational Intelligence*, vol. 12, no. 4, pp. 41–63, 2022.

[147] W.-T. Tsai, X. Wei, Y. Chen, R. Paul, J.-Y. Chung, and D. Zhang, "Data provenance in soa: security, reliability, and integrity," *Service Oriented Computing and Applications*, vol. 1, pp. 223–247, 2007.

[148] D. Appelbaum, "Securing big data provenance for auditors: The big data provenance black box as reliable evidence," *Journal of emerging technologies in accounting*, vol. 13, no. 1, pp. 17–36, 2016.

[149] S. Akoush, R. Sohan, and A. Hopper, "{HadoopProv}: Towards provenance as a first class citizen in {MapReduce}," in *5th USENIX Workshop on the Theory and Practice of Provenance (TaPP 13)*, 2013.

[150] E. Šimková, "Impact of the sarbanes-oxley act on the system of internal controls and is audit," *Systémová Integrace*, vol. 3, pp. 92–111, 2005.

[151] M. Bahrenburg, *Automated processes for Sarbanes-Oxley risk management in a UNIX environment*. Rochester Institute of Technology, 2008.

[152] "Slsa: Supply-chain levels for software artifacts," https://slsa.dev/, The Linux Foundation, 2025, accessed: 2025-01-18.

[153] "The update framework (tuf)," https://theupdateframework.io/, 2025, accessed: 2025-01-18. [Online]. Available: https://theupdateframework.io/

[154] Sigstore contributors, "sigstore/cosign: Code signing and transparency," https://github.com/sigstore/cosign, 2025, accessed: 2025-01-18. [Online]. Available: https://github.com/sigstore/cosign

[155] "in-toto: A framework to secure the integrity of software supply chains," https://in-toto.io/, 2025, accessed: 2025-01-18. [Online]. Available: https://in-toto.io/

[156] "Google cloud build provenance," https://cloud.google.com/build/docs/securing-builds/generate-validate-build-provenance, 2025, accessed: 2025-01-18. [Online]. Available: https://cloud.google.com/build/docs/securing-builds/generate-validate-build-provenance

[157] "Google cloud: Binary authorization," https://cloud.google.com/binary-authorization/docs, 2025, accessed: 2025-01-18. [Online]. Available: https://cloud.google.com/binary-authorization/docs

[158] "Aws well-architected framework: Devops guidance," https://docs.aws.amazon.com/wellarchitected/latest/devops-guidance/devops-guidance.html, 2023, publication date: September 20, 2023; Accessed: 2025-01-18. [Online]. Available: https://docs.aws.amazon.com/wellarchitected/latest/devops-guidance/devops-guidance.html

[159] T. Stalnaker, "Sboms as a solution in the software supply chain."

[160] T. Bi, B. Xia, Z. Xing, Q. Lu, and L. Zhu, "On the way to sboms: Investigating design issues and solutions in practice," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 6, pp. 1–25, 2024.

[161] R. Hiesgen, M. Nawrocki, T. C. Schmidt, and M. Wählisch, "The race to the vulnerable: Measuring the log4j shell incident," *arXiv preprint arXiv:2205.02544*, 2022.

[162] J. Martínez and J. M. Durán, "Software supply chain attacks, a threat to global cybersecurity: Solarwinds' case study," *International Journal of Safety and Security Engineering*, vol. 11, no. 5, pp. 537–545, 2021.

[163] M. Willett, "Lessons of the solarwinds hack," in *Survival April–May 2021: Facing Russia*. Routledge, 2023, pp. 7–25.

[164] "Coalition for content provenance and authenticity (c2pa)," https://c2pa.org/, 2025, accessed: 2025-01-18. [Online]. Available: https://c2pa.org/

[165] Coalition for Content Provenance and Authenticity, "Meta joins the coalition for content provenance and authenticity (c2pa)," https://spec.c2pa.org/post/meta_pr/, 2024, accessed: 2025-01-18.

[166] Reuters News Agency. (2023) Reuters authenticity proof of concept. Pilot with Canon and Starling Lab on C2PA-based provenance for photojournalism. [Online]. Available: https://reutersagency.com/authenticity-poc

[167] Adobe Content Authenticity Initiative. (2024) In the age of ai: Growing content credentials momentum across social media platforms and ai companies. Details on Adobe's integration of Content Credentials and industry adoption. [Online]. Available: https://blog.adobe.com/en/publish/2024/09/18/authenticity-age-ai-growing-content-credentials-momentum-across-social-media-platforms-ai-companies-rising-consumer-awareness

[168] Q. Wang, W. U. Hassan, D. Li, K. Jee, X. Yu, K. Zou, J. Rhee, Z. Chen, W. Cheng, and C. A. Gunter, "You are what you do: Hunting stealthy malware via data provenance analysis." in *NDSS*, 2020.

[169] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "Nodoze: Combatting threat alert fatigue with automated provenance triage," in *network and distributed systems security symposium*, 2019.

[170] R. Buchta, G. Gkoktsis, F. Heine, and C. Kleiner, "Advanced persistent threat attack detection systems: A review of approaches, challenges, and trends," *Digital Threats: Research and Practice*, vol. 5, no. 4, pp. 1–37, 2024.

[171] C. M. Pilato, B. Collins-Sussman, and B. W. Fitzpatrick, *Version control with subversion: next generation open source version control.* " O'Reilly Media, Inc.", 2008.

[172] J. Dietrich, T. White, B. Hassanshahi, and P. Krishnan, "Levels of binary equivalence for the comparison of binaries from alternative builds," in *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2025, pp. 576–587.

[173] D. J. Santry, M. J. Feeley, N. C. Hutchinson, and A. C. Veitch, "Elephant: The file system that never forgets," in *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*. IEEE, 1999, pp. 2–7.

[174] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and performance in a distributed file system," *ACM Trans. Comput. Syst.*, vol. 6, no. 1, p. 51–81, Feb. 1988. [Online]. Available: https://doi.org/10.1145/35037.35059

[175] R. Pike, D. Presotto, K. Thompson, and H. Trickey, "Plan 9 from bell labs," in *Proceedings of the summer 1990 UKUUG Conference*. London, UK, 1990, pp. 1–9.

[176] J. MacDonald, "File system support for delta compression," Ph.D. dissertation, Masters thesis. Department of Electrical Engineering and Computer Science, 2000.

[177] R. C. Burns and D. D. Long, "Efficient distributed backup with delta compression," in *Proceedings of the fifth workshop on I/O in parallel and distributed systems*, 1997, pp. 27–36.

[178] W. Xia, C. Li, H. Jiang, D. Feng, Y. Hua, L. Qin, and Y. Zhang, "Edelta: A {Word-Enlarging} based fast delta compression approach," in *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*, 2015.

[179] W. Xia, H. Jiang, D. Feng, L. Tian, M. Fu, and Y. Zhou, "Ddelta: A deduplication-inspired fast delta compression approach," *Performance Evaluation*, vol. 79, pp. 258–272, 2014.

[180] H. Tan, W. Xia, X. Zou, C. Deng, Q. Liao, and Z. Gu, "The design of fast delta encoding for delta compression based storage systems," *ACM Transactions on Storage*, vol. 20, no. 4, pp. 1–30, 2024.

[181] J. MacDonald, "xdelta: open-source binary diff / delta compression tools," https://github.com/jmacd/xdelta, Jan 2023, version 3.x, Apache-2.0 license.

[182] C. Percival, "Naive differences of executable code," *Draft Paper, http://www. daemonology. net/bsdiff*, 2003.

[183] J. J. Hunt, K.-P. Vo, and W. F. Tichy, "Delta algorithms: An empirical analysis," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 7, no. 2, pp. 192–214, 1998.

[184] A. Tridgell and P. Mackerras, "The rsync algorithm," 1996.

[185] M. Roitzsch and M. Pohlack, "Principles for the prediction of video decoding times applied to mpeg-1/2 and mpeg-4 part 2 video," in *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. IEEE, 2006, pp. 271–280.

[186] The PostgreSQL Global Development Group, *PostgreSQL Documentation: Write-Ahead Logging (WAL)*, 2024, accessed: 2024-04-28. [Online]. Available: https://www.postgresql.org/docs/current/wal-intro.html

[187] W. F. Tichy, "Rcs—a system for version control," *Software: Practice and Experience*, vol. 15, no. 7, pp. 637–654, 1985.

[188] R. Gliklich, M. Leavy, and N. Dreyer, "Tools and technologies for registry interoperability, registries for evaluating patient outcomes: A user's guide, addendum 2.(prepared by l&m policy research, llc under contract no. 290-2014-00004-c.) ahrq publication no. 19 (20)-ehc017-ef," *Rockville, MD: Agency for Healthcare Research and Quality*, 2019.

[189] M. A. Santana, P. L. Guimarães, E. S. Almeida, and T. Eklin, "The importance of metrological metadata in the environmental monitoring," in *Journal of Physics: Conference Series*, vol. 733, no. 1. IOP Publishing, 2016, p. 012033.

[190] *Information and Documentation — The Dublin Core Metadata Element Set — Part 1: Core Elements*, International Organization for Standardization Std. ISO 15 836-1:2017, 2017. [Online]. Available: https://www.iso.org/standard/71339.html

[191] P. Harvey, "ExifTool: Read and write meta information in files," https://exiftool.org/, 2010.

[192] W. D. Pence, L. Chiappetti, C. G. Page, R. A. Shaw, and E. Stobie, "Definition of the flexible image transport system (fits), version 3.0," *Astronomy & Astrophysics*, vol. 524, p. A42, 2010.

[193] D. C. Wells and E. W. Greisen, "Fits-a flexible image transport system," in *Image processing in astronomy*, 1979, p. 445.

[194] N. Ritter, M. Ruth, B. B. Grissom, G. Galang, J. Haller, G. Stephenson, S. Covington, T. Nagy, J. Moyers, and J. Stickley, "GeoTIFF format specification GeoTIFF revision 1.0," *SPOT Image Corp*, vol. 1, pp. 154–172, 2000.

[195] X. Liang, S. Shetty, D. Tosh, C. Kamhoua, K. Kwiat, and L. Njilla, "Provchain: A blockchain-based data provenance architecture in cloud environment with enhanced privacy and availability," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2017, pp. 468–477.

[196] Linux man-pages project, *xattr(7) — Linux Manual Page*, 2026, section: Filesystem differences. [Online]. Available: https://man7.org/linux/man-pages/man7/xattr.7.html

[197] Linux Kernel Documentation, *ext4 Extended Attributes*. [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/ext4/attributes.html

[198] Linux man-pages project, *attr(1) — Extended Attributes on XFS Filesystem Objects*. [Online]. Available: https://man7.org/linux/man-pages/man1/attr.1.html

[199] Microsoft, "Alternate data streams," Microsoft Learn, Windows Protocols, accessed: 2026-09-18. [Online]. Available: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-fscc/e2b19412-a925-4360-b009-86e3b8a020c8

[200] M. Russinovich, "Streams," Microsoft Sysinternals, Jul. 2016, accessed: 2026-09-18. [Online]. Available: https://learn.microsoft.com/en-us/sysinternals/downloads/streams

[201] Microsoft, "Alternate data backup stream structure," Microsoft Learn, Windows Protocols, accessed: 2026-09-18. [Online]. Available:

https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-bkup/3a04277f-0ccf-422f-bc26-fc7222205bdd

[202] Linux Kernel Documentation, "Ext4: Index nodes," Linux Kernel Documentation, accessed: 2026-09-18. [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/ext4/inodes.html

[203] Linux man-pages Project, *ext4(5): The Fourth Extended File System*, filesystem features and on-disk format compatibility. [Online]. Available: https://man7.org/linux/man-pages/man5/ext4.5.html

[204] XFS Project, *XFS Algorithms and Data Structures*, XFS Project, filesystem structure and on-disk inode documentation. [Online]. Available: https://www.kernel.org/pub/linux/utils/fs/xfs/docs/xfs_filesystem_structure.pdf

[205] C. Staelin and H. Garcia-Molina, "File system design using large memories," in *Proceedings of the 5th Jerusalem Conference on Information Technology, 1990.'Next Decade in Information Technology'.* IEEE, 1990, pp. 11–21.

[206] L. C. Gleim, L. Müller, F. Brillowski, and S. Decker, "Capturing provenance information in the file system." in *SEMANTiCS (Posters & Demos)*, 2021.

[207] IDRIX, *VeraCrypt*, IDRIX, 2025, https://veracrypt.io/en/Home.html.

[208] V. Gough, "Encfs-encrypted filesystem module for Linux," *http://arg0.net/users/vgough/encfs.html*, 2004.

[209] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," in *Proceedings of the Linux symposium*, vol. 2. Citeseer, 2007, pp. 21–33.

[210] M. Seltzer, K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and J. Ledlie, "Provenance-aware storage systems," 2005.

[211] L. K. Contributors, "Linux kernel documentation: sysctl/vm.txt," https://www.kernel.org/doc/Documentation/sysctl/vm.txt, 2023, accessed: 2023-02-12.

[212] The IOzone Team, *IOzone Filesystem Benchmark*, http://www.iozone.org, 1998, available at https://www.iozone.org/docs/IOzone_msword_98.pdf.

[213] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To {FUSE} or not to {FUSE}: Performance of {User-Space} file systems," in *15th USENIX*

Conference on File and Storage Technologies (FAST 17), 2017, pp. 59–72.

[214] B. K. R. Vangoor, P. Agarwal, M. Mathew, A. Ramachandran, S. Sivaraman, V. Tarasov, and E. Zadok, "Performance and resource utilization of fuse user-space file systems," *ACM Transactions on Storage (TOS)*, vol. 15, no. 2, pp. 1–49, 2019.

[215] I. Mironov *et al.*, "Hash functions: Theory, attacks, and applications," *Microsoft Research, Silicon Valley Campus*, vol. 10, 2005.

[216] V. Beck, M. Landauer, M. Wurzenberger, F. Skopik, and A. Rauber, "System log parsing with large language models: A review," *arXiv preprint arXiv:2504.04877*, 2025.

[217] P. K. Danso, L. McGevna, J. Chow, and J. Luo, "Llm-based anomaly detection for digital substation cybersecurity," in *2025 IEEE 22nd International Conference on Smart Communities: Improving Quality of Life using AI, Robotics and IoT (HONET)*. IEEE, 2025, pp. 128–133.

[218] E. Eilertsen, V. Mavroeidis, and G. Grov, "Towards agentic investigation of security alerts," in *2025 IEEE International Conference on Big Data (BigData)*. IEEE, 2025, pp. 7793–7802.