Flexible Tracing and Analysis of Applications' I/O Behavior

Tânia Esteves

UMinho | 2023

Universidade do Minho
Escola de Engenharia

Tânia da Conceição Araújo Esteves

Flexible Tracing and Analysis of Applications' I/O Behavior

outubro de 2023

Universidade do Minho
Escola de Engenharia

Tânia da Conceição Araújo Esteves

Flexible Tracing and Analysis of Applications' I/O Behavior

Tese de Doutoramento
Doutoramento em Informática

Trabalho efetuado sob a orientação de
João Tiago Medeiros Paulo
Rui Carlos Mendes Oliveira

outubro de 2023

Acknowledgements

A longa jornada que termina com a entrega deste documento revelou-se enriquecedora, mas também desafiadora, e não teria sido possível sem o apoio de diversas pessoas com as quais tive a oportunidade de trabalhar e conviver. A todos agradeço profundamente por me terem apoiado e acreditarem em mim.

Em primeiro lugar quero agradecer ao meu orientador, Professor João Paulo, por estar sempre disponível para me ouvir e discutir dúvidas e novas ideias, e pela incansável ajuda, motivação e excelente orientação ao longo de todo este processo que foi sem dúvida fundamental para atingir este ponto.

Agradeço também ao Professor Rui Oliveira, que coorientou este trabalho, por todas as sugestões e dicas que foram essenciais para o sucesso e qualidade deste trabalho.

Quero também expressar um especial obrigado à Cláudia Brito e ao Ricardo Macedo por toda a ajuda e conselhos e boa companhia ao longo deste percurso. Agradeço também ao Bruno Pereira, Francisco Neves, João Marco e Rui Pedro Oliveira pela colaboração em artigos resultantes deste trabalho, e ao Nuno Machado e Rogério Pontes pelas sugestões que tornaram o meu trabalho mais vigoroso. I am also grateful to Jason Haga for hosting me for a remote internship at AIST and for his insightful suggestions.

Agradeço igualmente aos demais colegas do laboratório, nomeadamente: Alberto Faria, Ana Alonso, Catarina Leones, Daniel Cruz, Diogo Couto, Diogo Ribeiro, Fábia Pereira, Fábio Coelho, Francisco Cruz, Francisco Maia, João Dias, Luís Meruje, Mariana Miranda, Miguel Peixoto, Paula Rodrigues, Ricardo Vilaça, Rita Moutinho, Rui Miguel, Rui Monteiro, Rui Pedro Oliveira, Susana Marques, Vitor Enes, e demais corpo docente, por todos os momentos de companheirismo, desabafo e alegria.

Quero também agradecer aos meus amigos que me acompanharam e apoiaram, e tornaram esta jornada mais interessante e alegre, nomeadamente: Andreia Barros, André Marinho, Bruno Ribeiro, Eduardo Cunha, Gabriel Fernandes, José Lima, José Monteiro, José Santos, João Afonso, João Fernandes, Luís Marques, Luís Pedro, Luís Viana, Mario Real.

Não posso deixar de agradecer à minha família, em especial aos meus pais, Teresa e Jorge, e ao meu irmão, Ricardo, por acreditarem em mim e sempre me apoiarem ao longo da minha vida. Muito obrigada por poder sempre contar convosco.

Adicionalmente, agradeço às instituições que apoiaram o trabalho aqui apresentado. A Fundação para a Ciência e Tecnologia (FCT) apoiou este trabalho através da bolsa de doutoramento DFA/BD/5881/2020. O Departamento de Informática da Universidade do Minho, o HASLab, e INESC TEC ofereceram-me as condições necessárias para o desenvolvimento deste trabalho.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

Braga, October 31st, 2023

Tânia da Conceição Araújo Esteves

(Tânia da Conceição Araújo Esteves)

Resumo

Rastreio e Análise Flexíveis do Comportamento de E/S de Aplicações

A correção, confiabilidade e desempenho de aplicações centradas em dados e de sistemas distribuídos (*por exemplo*, sistemas de ficheiros, bases de dados, plataformas de análise de dados e de aprendizagem de máquina) são influenciados pela forma como estes acedem, trocam e persistem dados. Portanto, compreender o comportamento de Entrada/Saída (E/S) destas soluções é fundamental para as explorar, encontrar possíveis problemas e otimizar. Para tal, existem ferramentas de diagnóstico que dão suporte à coleção, análise e visualização de padrões de E/S (*por exemplo*, chamadas de sistema, funções de kernel). Nesta dissertação, argumentamos que estas ferramentas podem ser melhoradas para alcançar soluções de diagnóstico integradas e automatizadas que permitam capturar informações detalhadas sobre pedidos de E/S, suportar análises múltiplas e automatizadas dos dados colecionados, e fornecer representações visuais que facilitam a interpretação de padrões de comportamento de E/S.

Estes objetivos são alcançados através de três novas plataformas de diagnóstico. Em primeiro lugar apresentamos o CaT, uma solução orientada ao conteúdo que permite uma análise mais abrangente de sistemas distribuídos, revelando como os dados fluem pelos distintos componentes até que sejam persistidos. Através de dois casos de estudo reais mostramos que esta informação é fundamental para identificar padrões de corrupção e adulteração de dados em soluções distribuídas. Em seguida propomos o DIO, uma solução genérica para o diagnóstico de aplicações centradas em dados que oferece funcionalidades de coleção, análise e visualização detalhadas, flexíveis, e personalizáveis. Através de uma avaliação experimental, com quatro aplicações utilizadas pela indústria, mostramos que a nossa solução facilita a análise da origem de problemas conhecidos, e permite observar e validar padrões de E/S ineficientes (e anteriormente desconhecidos). Por fim, apresentamos o CRIBA, uma plataforma que estende o DIO para fornecer uma solução especializada e automatizada que permite caracterizar o comportamento de E/S de *ransomware* criptográfico. O nosso estudo com cinco famílias de *ransomware* para Linux mostra como o CRIBA permite a análise e observação dos seus comportamentos intrínsecos e complexos.

As contribuições anteriores facilitam e melhoram o diagnóstico de aplicações e de sistemas de armazenamento. Acreditamos que soluções de diagnóstico detalhadas, flexíveis e personalizáveis, como as propostas neste trabalho, são fundamentais para a construção de sistemas mais robustos e eficientes.

Palavras-chave: Diagnóstico de E/S, Rastreio, Análise, Visualização, Aplicações centradas em dados

Abstract

Flexible Tracing and Analysis of Applications' I/O Behavior

The correctness, dependability and performance of data-centric applications and distributed systems (*e.g.*, file systems, databases, analytical engines, machine learning frameworks) are highly influenced by the way these access, exchange and persist data. Therefore, understanding the Input/Output (I/O) behavior of such solutions is key for efficiently exploring, debugging and optimizing them. This endeavor is possible through diagnosis tools that provide support for the collection, analysis and visualization of information (*e.g.*, logs, system calls, kernel functions) from targeted applications and storage systems. In this thesis, we argue that these tools can be further enhanced to achieve fully automated and integrated diagnosis pipelines that allow capturing comprehensive information about I/O requests, supporting multipurpose and automated analysis of collected data, and providing informative and summarized visual representations that ease the interpretation of I/O behavior patterns for users.

We accomplish these goals by proposing three novel diagnosis frameworks. First, we introduce CaT, a content-aware solution that enables a more comprehensive analysis of distributed systems by revealing how data requests flow across distinct components until these are persisted. We show that this information is key for identifying data corruption and adulteration patterns in complex distributed solutions. Then, we propose DIO, a general-purpose solution for diagnosing data-centric applications that offers flexible, comprehensive and customizable tracing, analysis and visualization in near real-time. Through an experimental evaluation including four production-level applications, we show that our solution eases the root cause analysis of known issues and allows observing and validating inefficient (and previously unknown) I/O patterns. Finally, we present CRIBA, a framework that extends DIO to provide a specialized and automated pipeline for characterizing the I/O behavior of cryptographic ransomware. Our study, including five Linux ransomware families, shows that CRIBA enables the analysis and observation of intrinsic and complex I/O behavior from malicious samples.

The previous contributions ease and improve the process of diagnosing applications and storage systems for users. We believe that comprehensive, flexible and customizable diagnosis pipelines, such as the ones proposed in this work, are key for building systems that are more robust and efficient.

Keywords: I/O diagnosis, Tracing, Analysis, Visualization, Data-centric applications

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Acronyms

API	Application Programming Interface 12, 14, 58, 86, 90, 104
BCC	BPF Compiler Collection 37
BFS	Breadth First Search 93
C&C	Command and Control 88, 89, 98
CPU	Central Processing Unit 4, 6, 14, 25, 29, 34, 35, 37, 41, 42, 43, 44, 52, 67, 70, 72, 77, 78, 79, 82, 87, 88, 89, 91, 96, 102, 103
CTF	Common Trace Format 17
CUDA	Compute Unified Device Architecture 37
DAG	Directed Acyclic Graph 21
DBSCAN	Density-Based Spatial Clustering of Applications with Noise 21
DFS	Depth First Search 93, 97
DPDK	Data Plane Development Kit 12
eBPF	Extended Berkeley Packet Filter 5, 9, 17, 18, 25, 29, 33, 34, 35, 36, 37, 40, 42, 45, 48, 52, 55, 57, 58, 69, 72, 73, 75, 83, 94, 106, 108
ETW	Event Tracing for Windows 14
FSA	Finite-State Automaton 21
GPU	Graphics Processing Unit 37, 41
gRPC	Google Remote Procedure Call 11

HDD	Hard Disk Drive 37
HPC	High-Performance Computing 20
I/O	Input/Output 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 14, 15, 16, 17, 18, 19, 20, 21, 24, 25, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 40, 42, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 57, 59, 61, 62, 63, 65, 66, 67, 68, 70, 71, 72, 75, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 98, 104, 105, 106, 107
IT	Information Technology 19
J2EE	Java 2 Platform Enterprise Edition 15
JIT	Just-In-Time 17
KVS	Key-Value Store 22, 47, 64, 66
LoC	Lines of Code 2, 36, 37, 47, 57, 58, 61, 65
LOF	Local Outlier Factor 21
LSH	Locality-Sensitive Hashing 35, 36, 37
LSM	Linux Security Module 16, 17, 45
LSM-tree	Log-Structured Merge tree 66
LTTng	Linux Trace Toolkit Next Generation 9, 16, 17, 18, 25, 83, 108
ML	Machine Learning 20, 21, 25, 31, 37, 45, 47, 86, 107, 108
MPI	Message Passing Interface 11, 15, 16
NIC	Network Interface Controller 11
NVMe	Non-Volatile Memory Express 37, 57, 59, 96
OS	Operating System 3, 9, 11, 13, 16, 17, 18, 25, 62, 72, 86, 87, 96, 108
OSD	Object Storage Devices 23
PCA	Principal Component Analysis 21
PID	Process Identifier 5, 14, 15, 16, 17, 27, 31, 32, 33, 34, 36, 43, 48, 51, 52, 54, 84, 87, 88
POSIX	Portable Operating System Interface 5, 6, 12, 47, 48, 51, 85, 106
PVFS	Parallel Virtual File System 16
RAM	Random Access Memory 4, 25, 29, 37, 42, 43, 44, 59, 91, 96
RPC	Remote Procedure Call 15

SCI	System Call Interface 3, 11, 12, 15
SCSI	Small Computer Systems Interface 17
SDK	Software Development Kit 14
SMT	Satisfiability Modulo Theories 21, 32
SPDK	Storage Performance Development Kit 12
SQL	Structured Query Language 22
SSD	Solid State Drive 37, 59, 96
SVM	Support Vector Machine 21
TF-IDF	Term Frequency-Inverse Document Frequency 94
t-SNE	t-distributed Stochastic Neighbor Embedding 21
TID	Thread Identifier 14, 48, 51, 52, 54, 81, 87, 98, 100
UPGMA	Unweighted Pair Group Method with Arithmetic Mean 21
VFS	Virtual File System 3, 11, 12, 18, 83
VM	Virtual Machine 17, 89, 91, 96

1 Introduction

In the last decades, we have witnessed exponential growth in the amount of digital information generated. Further, this data contains more and more information that is crucial for critical services such as health-care, finance, and governance to work properly, which raises the need to design efficient, dependable and secure solutions for storing and retrieving data [22]. As the complexity (*i.e.*, codebase size, number of components, Input/Output (I/O) optimizations) of data-centric applications and storage systems grows, it becomes increasingly difficult to debug, validate, or even improve the aforementioned guarantees.

For example, to scale and reliably handle the sheer volume of data, applications and storage systems typically adhere to distributed designs composed of several components, including coordination services, communication middleware, databases and file systems. These components need to exchange data among themselves and thus often implement complex protocols, such as data replication techniques, that are susceptible to subtle errors [3].

As another example, the interaction between applications and storage systems is also complex and subtle, both in distributed and local designs. Application developers must be knowledgeable about the interface and characteristics (*e.g.*, optimal I/O size, access pattern) of the underlying storage system to take advantage of its optimizations (*e.g.*, caching, scheduling) and extract the best performance from it [31]. Also, the incorrect use of such interfaces (*e.g.*, improper use of asynchronous I/O, absence of `fsync` system calls to make data durable) can introduce critical bugs that compromise dependability, for instance, leading to data loss [96].

In this thesis, we argue that it is crucial to comprehensively study and understand the intrinsic I/O behavior of applications and storage systems and their interplay. This process, which we refer to as *"I/O diagnosis"*, holds critical significance for various purposes.

Debugging. Be it due to human error, lack of detailed knowledge on how to efficiently and correctly access the underlying storage (*e.g.*, file system, block-device), or usage of high-level libraries that obfuscate the actual I/O requests being made to storage systems, applications often exhibit: *i)* costly access patterns, such as small-sized I/O requests or random accesses [31], *ii)* redundant operations, such as unnecessarily re-opening and closing a given file;[1] *iii)* I/O contention caused by having concurrent requests accessing

[1] *Logging improvements* issue from Redis' GitHub repository: `https://github.com/redis/redis/pull/10531`

shared storage resources [12]; and *iv)* erroneous usage of I/O calls, for example, by accessing wrong file offsets.[2,3] Therefore, understanding the I/O behavior of applications is essential for uncovering the root cause of errors, inefficiencies and unattained performance.

Validation. Studying applications and storage systems' I/O patterns is also fundamental for validating their implementations. For instance, it allows checking if applications are following the intended storage access patterns (*e.g.*, training dataset shuffling in deep learning frameworks [1]) or if complex distributed protocols respect their specifications (*e.g.*, synchronous data replication in distributed file systems [46]). Further, by studying I/O patterns, one can also validate that the implementation corrections for a given error or inefficiency found at the debugging phase are working properly.

Exploration. Lastly, understanding how applications and storage systems handle data requests is also valuable for exploring new features and optimizations. For instance, by profiling the I/O requests made by applications (*e.g.*, random file access patterns), one can then configure and optimize storage systems to achieve better performance [31]. Additionally, learning the I/O behavior of closed source applications (*i.e.*, whose source code is unavailable) can be valuable for security purposes, for instance, to uncover specific patterns that characterize malware activity and that can be used in its detection [88].

Despite the importance and benefits of I/O diagnosis to the dependability and performance of key information systems, as the preceding examples depict, how to do it efficiently remains an open challenge. Understanding applications' behavior through manual code inspection is a difficult, if not impossible, task. First, the source code must be available, which is not the case for many applications (*e.g.*, malware binaries, commercial products). Even if the source code is available, the applications may have complex and large codebases with components implemented in different programming languages and by different developers. For example, Fluent Bit's open-source project has around 5K files, most of them containing C code, but it also includes files written in Python, Go, and Java. In total, the project contains ≈1M Lines of Code (LoC) developed by more than 300 contributors. TensorFlow's codebase has even more files, almost 20K, with more than 4M LoC written by 3K contributors in several programming languages. Exploring and understanding these multi-language and extensive codebases is a hard and time-consuming endeavor. Further, code inspection mainly focuses on the structure of the source code, not assessing the impact of different program inputs on the application's behavior, thus limiting the scope of the analysis [23].

These challenges motivate the use of diagnosis tools to automate the processes of I/O debugging, validation, and exploration. The pipeline of these tools is typically divided into two main phases: *i)* data collection and *ii)* data analysis and visualization.

The *data collection* phase involves collecting information related to the I/O requests that applications and storage systems handle at runtime. This can be done by resorting to source code instrumentation, which is the process of adding extra code (*i.e.*, instrumentation code) in key parts of the codebase to record, for instance, the values of function parameters and timing statistics [13, 45, 115]. Another widely

[2]Wrong offsets issue from Fluent Bit's repository: `https://github.com/fluent/fluent-bit/issues/1875`
[3]Log missing issue from Fluent Bit's repository: `https://github.com/fluent/fluent-bit/issues/4895`

used strategy is to rely on tracing tools, which allow instrumenting lower layers of the Operating System (OS) (*e.g.*, the System Call Interface (SCI)) to collect information about the I/O requests submitted by the application to this layer without requiring any modifications to its source code [32, 64, 109, 111].

The *data analysis and visualization* phase involves the analysis and exploration of the previously collected information. The information captured for all I/O requests is inspected and correlated, for instance, by checking the types of requests and their arguments, analyzing the time and order in which they occurred, or identifying the processes and threads that originated different groups of requests. The output of this analysis, which in some cases may be delivered to users through visual representations, is key to better understanding the I/O behavior of applications and storage systems [18, 126].

Currently, many of the existing solutions only consider one of these phases (*e.g.*, tracing tools aim at data collection, and visualization tools focus on data analysis), while few solutions provide a complete pipeline that includes components for collecting, analyzing and visualizing I/O requests.

1.1 Problem Statement and Objectives

Despite the considerable advantages brought by current I/O diagnosis tools, in this dissertation, we argue that the diagnosis process can be made more flexible and practical and can be further automated by overcoming the following challenges:

Tracing Transparency. Many data collection tools rely on source code instrumentation to obtain information about the applications' I/O requests [13, 45, 62, 74, 75, 115, 132]. While code instrumentation solves some of the challenges associated with manual code inspection and exploration (*e.g.*, by observing the runtime I/O behavior of applications), it still requires a manual analysis of complex and extended codebases to identify the key parts of the source code that must be instrumented. Moreover, code instrumentation is unsuitable for diagnosing closed source applications whose codebase is unavailable.

In this work, we explore non-intrusive approaches to intercept I/O requests at lower OS layers, achieving a transparent solution applicable to any traditional kernel-based storage application.

Tracing Accuracy. The OS layer where I/O requests are intercepted influences the detail of information one can infer about the applications' behavior. The lower the layer is at the OS (*e.g.*, Virtual File System (VFS), block device), the more optimizations like I/O merging and reordering are applied, thus masking the exact requests submitted by applications [127]. Also, the amount of information captured for each I/O request influences the types of analysis users can do. While capturing only the type (*e.g.*, `open`, `read`, `write`, `close`) and number of I/O requests already provides valuable insights about the applications' I/O behavior, collecting the requests' arguments and return value can further expand the analysis possibilities, for instance, to uncover scenarios where system calls are specified erroneously.

We study the most appropriate layer for intercepting I/O requests and explore the impact of capturing more or less detailed information about these.

Tracing Overhead. Another key aspect that must be considered is the overhead imposed on the underlying storage and the targeted application's performance when intercepting and collecting I/O requests. By increasing the amount of information being collected, one is also increasing the storage space needed to persist such traces for later analysis. Similarly, capturing more information can result in a higher consumption of system resources (*e.g.*, Random Access Memory (RAM), Central Processing Unit (CPU), and disk), slowing down the targeted application and impacting its performance.

We explore different strategies to balance the amount of collected data (accuracy) with its impact on performance overhead and resource usage.

Integrated and Automated Analysis and Visualization. Another challenge to bear in mind is the need to provide mechanisms for facilitating the analysis and visualization of collected data. The sheer number of storage operations generated by data-centric applications, ranging from hundreds to thousands of operations per second, makes their analysis complex and time-consuming when done manually.

We enhance diagnosis pipelines with efficient and integrated components that allow collecting, analyzing, and visualizing I/O requests in a more practical way. Moreover, we provide mechanisms to further automate the analysis process by preprocessing and correlating the collected data and outputting summarized insights about the analyzed I/O requests.

Flexible and Comprehensive Diagnosis. Existing diagnosis tools are often designed for rigid analysis scenarios, such as detecting unreproducible builds [98], observing file offset access patterns [102], or identifying security issues [64, 126]. Thus, for multipurpose diagnosis tasks, one needs to combine several tools and repeat the data collection and analysis phases multiple times for the same application.

We provide a more comprehensive solution that allows users to simultaneously debug, validate, and explore different kinds of I/O behaviors. By offering users the flexibility to narrow or broaden both data collection and analysis scopes, our solution allows them to explore a wider range of correctness, dependability and performance issues that applications may exhibit.

1.2 Contributions

This thesis presents three main contributions, which are aligned with the aforementioned goals and that advance the state of the art for I/O diagnosis.

Content-aware Diagnosis. As the first contribution, we explore how the contents of I/O requests can be useful for improved diagnosis of distributed systems. Namely, we propose CaT, a novel framework for collecting and analyzing storage and network I/O requests of distributed systems. The key insight and novelty behind CaT relies on intercepting and analyzing the content of data buffers transmitted between different components of a distributed deployment. By following this idea, CaT allows identifying duplicate data as well as near-duplicate data (with a high degree of similarity, for instance, >80%) that was slightly modified while flowing through different components (*e.g.*, messages that include the same payload but have a different metadata header).

To transparently intercept the applications' I/O requests, CaT explores two kernel-level tracing tools, Strace [109] and Extended Berkeley Packet Filter (eBPF) [76], which provide different tradeoffs regarding resource usage, amount of collected information, and I/O performance impact.

Moreover, CaT uses hashing techniques to summarize the requests' content and minimize the storage space overhead, applies near-duplicate detection algorithms to find similarities between data from distinct distributed events, and uses a color-based scheme to visually pinpoint I/O events handling near-similar data. These algorithms are integrated into a complete pipeline that allows automating the process of capturing, analyzing, and visualizing the context and content of applications' I/O requests.

CaT's content-aware approach enables detecting data adulteration, corruption, and leakage patterns in complex systems and I/O flows that would go unnoticed with state-of-the-art context-based approaches.

Comprehensive and Flexible Diagnosis. As the second contribution, we explore how current tools could be improved to aid in the diagnosis of a wider range of storage correctness, dependability and performance issues that applications may exhibit. To this end, we propose DIO, a generic tool for observing and diagnosing the I/O interactions between applications and in-kernel Portable Operating System Interface (POSIX) storage systems.

DIO's main insight is that by combining system call tracing with a customizable analysis pipeline, one can achieve non-intrusive and comprehensive I/O diagnosis for applications. To that end, it offers a new eBPF-based tracer that non-intrusively intercepts storage system calls submitted by applications, collects a compressive set of information for each operation, and enriches such information with additional context obtained from kernel structures (*e.g.*, file offset for `read` and `write` operations). Nonetheless, the amount and detail of collected information can be specified and filtered according to the needs of each user, thus balancing the accuracy and performance/storage overhead of our tracing solution.

Further, DIO follows an inline approach by automatically collecting the desired information and forwarding it directly to the analysis pipeline, allowing users to query and visualize captured data in near real-time. Moreover, DIO's analysis pipeline is flexible and configurable, allowing users to implement correlation algorithms and build custom visual representations that better fit their analysis requirements.

Through a comprehensive and integrated pipeline that automates the process of tracing, filtering, correlating, and visualizing millions of system calls, DIO enables the diagnosis of a wide range of I/O issues, avoiding the need for combining multiple tools and running the application multiple times.

Custom and Improved Analysis. As the third contribution, we explore how DIO's analysis capabilities can be improved through custom correlation algorithms. We focus on a new use case, namely analyzing the I/O behavior of cryptographic ransomware, and on how this endeavor can help security analysts find both characteristic and deviating I/O actions for building or improving detection tools.

Leveraging DIO's pipeline, we built CRIBA, a tool for capturing, analyzing, and visualizing the behavior of Linux cryptographic ransomware. CRIBA supports the collection and analysis of comprehensive information about I/O system calls (*e.g.*, type, arguments), their contextual information (*e.g.*, Process Identifier

(PID), offset), and their correlation with other system metrics (*e.g.*, CPU). Moreover, CRIBA offers automated analysis capabilities through six algorithms that ease the study and comparison of ransomware samples by pinpointing their file system transversal, file access, and file extension manipulation patterns. It also includes a predefined set of visualizations, organized into eight distinct dashboards, for summarizing and exploring the collected information and the outputs of the correlation algorithms in a human-readable and explainable fashion.

A comprehensive analysis and comparison of five Linux ransomware families shows that CRIBA automates the analysis and observation of generic behavior from ransomware samples (*e.g.*, the number of processes, type of system calls, file system transversal). Further, it enables the analysis and comparison of intrinsic and complex I/O behavior (*e.g.*, file access patterns, extension manipulation) related to the creation of ransom notes, file encryption, and evasion techniques used by each family.

1.3 Results

Core Publications. The work discussed in this thesis resulted in a number of publications in international conferences, journals, and workshops.

Tânia Esteves, Francisco Neves, Rui Oliveira, João Paulo. **CaT: Content-aware Tracing and Analysis for Distributed Systems.** In *22nd International Middleware Conference*, 2021.

This conference publication describes CaT, a novel framework for collecting and analyzing storage and network I/O requests of distributed systems. CaT proposes a content-aware tracing and analysis strategy that correlates the context and content of events to better understand the data flow of systems. Through a detailed evaluation of CaT's open-source prototype with real applications, we show that, depending on the target workload, it is possible to capture most of the I/O events while incurring negligible performance overhead. Moreover, we showcase that CaT's content-aware approach can improve the analysis of distributed systems by pinpointing their data flows and I/O access patterns. CaT is publicly available at https://github.com/dsrhaslab/cat.

Tânia Esteves, Ricardo Macedo, Rui Oliveira, João Paulo. **Diagnosing Applications' I/O Behavior through System Call Observability.** In *53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*, 2023.

This workshop publication presents DIO, a generic tool for observing and diagnosing I/O interactions between applications and in-kernel POSIX storage systems. DIO helps users diagnose I/O issues through a pipeline that automates the process of tracing, filtering, correlating, and visualizing millions of system calls. Our experiments with the RocksDB and Fluent Bit systems show that DIO provides key information for observing erroneous I/O access patterns that lead to data loss and identifying resource contention in multithreaded I/O that leads to high tail latency. DIO is publicly available at https://github.com/dsrhaslab/dio.

Tânia Esteves, Bruno Pereira, Rui Pedro Oliveira, João Marco, João Paulo. **CRIBA: A Tool for Comprehensive Analysis of Cryptographic Ransomware's I/O Behavior.** In *42nd Symposium on Reliable Distributed Systems*, 2023.

This conference publication describes CRIBA, a tool for simplifying and automating the exploration, analysis, and comparison of I/O patterns for Linux cryptographic ransomware. CRIBA supports the non-intrusive and comprehensive collection of I/O information from ransomware samples and combines it with an integrated analysis and visualization pipeline. The latter is enhanced with six custom correlation algorithms and different predefined dashboards. As shown in our experimental study, these features are key for *i)* automating the analysis of ransomware families; *ii)* understanding complex and intrinsic behavior from each sample; *iii)* and pinpointing common and distinct traits across families. CRIBA is publicly available at https://github.com/dsrhaslab/criba.

Tânia Esteves, Ricardo Macedo, Rui Oliveira, João Paulo. **Toward a Practical and Timely Diagnosis of Applications' I/O Behavior.** In *IEEE Access*, 2023.

This journal publication extends DIO's workshop paper by providing further details about the tool's design and implementation, introducing two new use cases with the Elasticsearch and Redis production-level applications that demonstrate the readiness and relevance of our solution, and providing new experiments that evaluate and compare DIO with related solutions. The conducted experimental evaluation highlights the different tradeoffs in terms of performance impact, resource usage, and data collection accuracy when using the different tracing modes and configurations provided by DIO while validating our solution against two state-of-the-art system calls tracers: Strace [109] and Sysdig [111]. Results show that when compared with an inline diagnosis pipeline using Sysdig, DIO provides timely analysis for users and improves the number of captured events by up to 28× while keeping performance overhead between 14% and 51%.

Complementary Publications. The following work was published in collaboration with multiple researchers from both academia and industry. While complementary to the core contributions of this thesis, these works leverage the topics discussed in it.

Mariana Miranda, Tânia Esteves, Bernardo Portela, João Paulo. **S2Dedup: SGX-enabled Secure Deduplication.** In *14th ACM International Systems and Storage Conference*, 2021.

This conference publication describes S2Dedup, a secure deduplication system based on trusted hardware. By supporting multiple schemes, S2Dedup can offer tailored deployments for applications with distinct security, performance, and space savings requirements. Moreover, compared to state-of-the-art solutions, its novel Epoch and Exact Frequency scheme enables improved security without sacrificing storage performance or deduplication space savings. S2Dedup is publicly available at https://github.com/dsrhaslab/s2dedup.

Tânia Esteves, Ricardo Macedo, Alberto Faria, Bernardo Portela, João Paulo, José Pereira, Danny Harnik. **TrustFS: An SGX-enabled Stackable File System Framework.** In *38th International Symposium on Reliable Distributed Systems Workshops*, 2019.

This workshop publication describes TrustFS, a programmable and modular stackable file system framework for implementing secure content-aware storage functionalities over hardware-assisted trusted execution environments [41]. It extends the original design of the SafeFS system [91] to provide the isolated execution guarantees of Intel Software Guard Extensions [30]. A preliminary evaluation of a compression prototype built using TrustFS shows that it incurs reasonable performance overhead under most workloads when compared to conventional storage systems, with throughput degradation ranging from 6.5% up to 31.3%. TrustFS is publicly available at https://github.com/-taniaesteves/TrustFS.git.

1.4 Outline

The rest of the document is organized as follows:

- **Chapter 2.** We introduce background concepts of the I/O diagnosis process and discuss the approaches followed by existing diagnosis solutions regarding their techniques for data collection, analysis and visualization.
- **Chapter 3.** We introduce the design, implementation, and evaluation of CaT, a novel framework for collecting and analyzing storage and network I/O requests of distributed systems.
- **Chapter 4.** We discuss the design, implementation, and evaluation of DIO, a generic tool for observing and diagnosing I/O interactions between applications and in-kernel storage systems.
- **Chapter 5.** We present the design, implementation, and evaluation of CRIBA, a tool built upon DIO that simplifies and automates the exploration, analysis, and comparison of I/O patterns for Linux cryptographic ransomware.
- **Chapter 6.** We discuss future research work and present the final remarks of this thesis.

2

I/O Diagnosis Background

The I/O diagnosis process includes two major phases: *i)* data collection, and *ii)* data analysis and visualization (as depicted in Fig. 2.1). These phases are typically carried out by multiple components that work together as part of a pipeline.

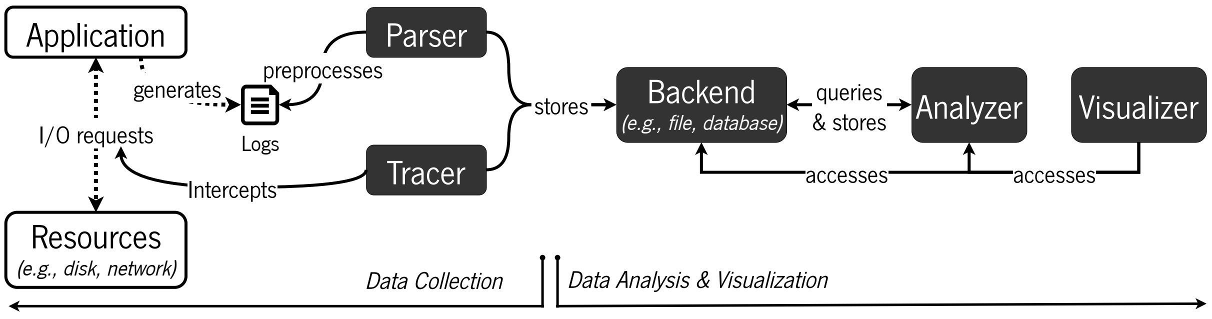


Figure 2.1. High-level design of a diagnosis pipeline.

The data collection phase comprises the process of obtaining information about the targeted application, for instance, through a parser that preprocesses applications or system logs, or a tracer that instruments applications, middleware components (*e.g.*, network protocols or system libraries), or the OS to intercept the applications' requests. The data analysis and visualization phase is dedicated to processing the collected data and presenting summarized insights to the user regarding the analysis' findings. It encompasses a backend component responsible for persisting both collected data and additional information obtained from its analysis and correlation, which is done with the analyzer component. The final component in this phase is typically the visualizer, which provides users with the means to explore and visualize the collected data and the analysis' findings.

Table 2.1 categorizes existing diagnosis tools regarding their strategies for data collection, analysis, and visualization. We focus on solutions for diagnosing the storage I/O behavior of applications, although some of these also consider network I/O. Further, we consider technologies whose purpose is data collection (*e.g.*, eBPF, Linux Trace Toolkit Next Generation (LTTng)), as long as these are used by full-fledged diagnosis solutions included in the table.

Table 2.1. Categorization of diagnosis tools regarding data collection, analysis and visualization.

		Collection				**Analysis**					**Visualization**		
	Scope	*Logs*	*Application*	*Middleware*	*OS*	*Purpose*	*Algorithm*	*Backend*	*Real-time*	*Postmortem*	*Type*	*Tool*	*Interactive*
Logstash [10]	*C*	●	○	○	○	—	—	—	—	—	—	—	—
LogMine [54]	*C*	●	○	○	○	—	—	—	—	—	—	—	—
Xu et al. [124]	*CAV*	●	○	○	○	*P*	*D*	*D*	○	●	*S*	⁎	○
SherLog [128]	*CAV*	●	○	○	○	*R*	*A*	*D*	○	●	*S*	⁎	○
Iprof [131]	*CAV*	●	○	○	○	*P*	*S*	*N*	○	●	*M*	*L*	●
LogLens [36]	*CAV*	●	○	○	○	*P*	*A*	*N*	●	●	*M*	*F*	●
LongLine [126]	*CAV*	●	○	○	○	*P*	*D*	*N*	●	●	*M*	*F*	●
Falcon [82]	*CAV*	●	○	○	●	*R*	*A*	*D*	○	●	*S*	*L*	○
Horus [84]	*CAV*	●	○	○	●	*R*	*G*	*N*	●	●	*S*	*F*	●
Magpie [13]	*CA*	○	●	●	●	*P*	*D*	*D*	○	●	—	—	—
Pivot Tracing [74, 75]	*CA*	○	●	○	○	*P*	*S*	—	●	○	—	—	—
Stardust [115]	*CAV*	○	●	●	●	*P*	*G*	*R*	●	●	*S*	*L*	○
X-Trace [45]	*CAV*	○	●	●	○	*P*	*G*	*R*	○	●	*S*	⁎	○
Zipkin [132], Jaeger [62]	*CAV*	○	●	○	○	*P*	*S*	*RN*	●	●	*M*	⁎	●
Darshan [108]	*C*	○	○	●	○	—	—	—	—	—	—	—	—
EZIOTracer [81]	*C*	○	○	●	●	—	—	—	—	—	—	—	—
Pinpoint [24]	*CA*	○	○	●	○	*R*	*D*	*D*	○	●	—	—	—
ScalaIOTrace [117]	*CA*	○	○	●	○	*P*	*S*	*D*	○	●	—	—	—
IOPin [67]	*CA*	○	○	●	○	*P*	*S*	*R*	○	●	—	—	—
Dapper [106]	*CAV*	○	○	●	○	*P*	*S*	*N*	●	●	*M*	⁎	●
DXT [122]	*CAV*	○	○	●	○	*P*	*S*	*D*	○	●	⁎	⁎	⁎
DXT Explorer [18]	*V*	—	—	—	—	—	—	—	—	—	*M*	*L*	●
Strace [109], Sysdig [111] Re-Animator [5], IOscope [102]	*C*	○	○	○	●	—	—	—	—	—	—	—	—
CamFlow [89]	*C*	○	○	○	●	—	—	—	—	—	—	—	—
Tracee [116]	*CA*	○	○	○	●	*P*	*A*	*D*	●	⁎	—	—	—
S2Logger [110]	*CAV*	○	○	○	●	*P*	*G*	*R*	●	●	*S*	⁎	⁎
RepTrace [98]	*CAV*	○	○	○	●	*R*	*G*	*D*	○	●	*S*	⁎	○
Daoud and Dagenais [32]	*CAV*	○	○	○	●	*P*	*A*	*D*	⁎	●	*M*	*F*	●
Kohyarnejadfard et al. [68]	*CAV*	○	○	○	●	*P*	*D*	*D*	⁎	●	*M*	*F*	●
PerSecMon [64]	*CAV*	○	○	○	●	*P*	*S*	*N*	●	●	*M*	*F*	●

Properties	**Scope**	**Purpose**	**Algorithm**	**Backend**	**V. Type**	**V. Tool**
● Supported	***C*** - Collection	***P*** - Profiling & anomaly detection	***S*** - Statist. methods	***D*** - Disk	***S*** - Specific	***L*** - Libraries
○ Not supported	***A*** - Analysis	***R*** - Root cause analysis	***A*** - Alg. & models	***R*** - Relational DB	***M*** - Multiple	***F*** - Framework
⁎ Unspecified	***V*** - Visualization		***D*** - Data mining & ML	***N*** - Non-relational DB		
— Not Applicable			***G*** - Graphs			

In the following sections, we discuss each diagnosis phase in more detail. First, we present the different levels and strategies for data collection (§2.1). Subsequently, we focus on the analysis phase, covering aspects such as the different analysis purposes (§2.2.1), algorithms (§2.2.2), and backend technologies (§2.2.3). We then detail different types of visualizations used in previous work (§2.3.1), along with the tools that support these (§2.3.2). Finally, we summarize our findings and discuss open research challenges in the I/O diagnosis field (§2.4).

2.1 Data Collection

Applications requiring data transmission and persistence must perform I/O operations (*e.g.*, write, send). These operations typically traverse middleware components and multiple layers within the OS before reaching the intended physical device (*e.g.*, disk, Network Interface Controller (NIC)).

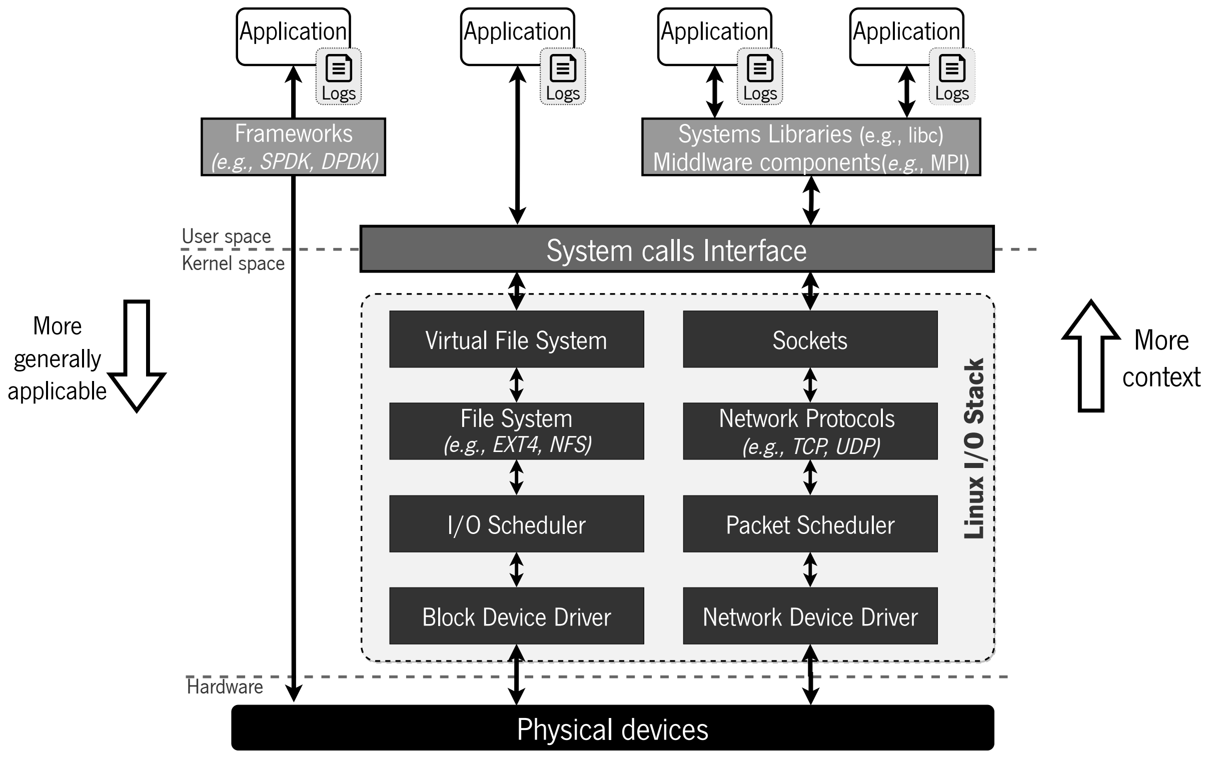


Figure 2.2. Different data collection levels throughout the I/O stack.

As illustrated in Fig. 2.2, for many applications, I/O operations are initially handled by I/O middlewares (*e.g.*, Message Passing Interface (MPI), Google Remote Procedure Call (gRPC)) and/or system libraries (*e.g.*, libc, glibc) before reaching the SCI layer. The SCI lies on the boundary between user space and kernel space, offering the means for user space programs to request the kernel to execute specific actions on their behalf.

When a file system is being used, storage I/O operations that reach the SCI layer are redirected to the VFS layer, which provides the file system interface to user space programs and acts as an abstraction within the kernel that allows different file system implementations to coexist (*e.g.*, Ext4, ZFS, NFS). From the VFS layer, I/O operations are forwarded to the concrete file system implementation, which performs several tasks of address resolution and translation, transforming logical I/O operations into physical operations that are then submitted to the I/O scheduler. The latter is responsible for determining the order in which operations are sent to device drivers. The block device driver layer is the one that actually handles the data transfer operation, issuing the corresponding commands to the hardware interfaces of the disk controller.

Network I/O operations flow through a similar stack. For example, for applications and/or libraries using traditional network sockets, from the SCI layer, the operations are redirected to the socket layer, the network protocol layer, the packet scheduler layer, and finally, the network driver layer, which submits the final network request to the physical hardware.

Although this is the traditional flow of I/O operations submitted by user space applications, there are, however, some applications that use frameworks such as the Storage Performance Development Kit (SPDK) [125] and the Data Plane Development Kit (DPDK) [61] to bypass the kernel and access directly storage and network devices.

I/O Instrumentation. To intercept information about applications' I/O requests, existing diagnosis tools often instrument one or more of these layers [81]. The chosen layer(s) for collecting such information impacts how much knowledge (context) one can gather from the actual I/O operations done by the application(s) and how much the diagnosis solution is generally applicable to different types of applications.

In detail, when instrumentation is done at the application level, the diagnosis solution is application-dependent. This means that to diagnose different applications, one must individually instrument multiple source codes or binaries. If I/O instrumentation is instead done at the middleware level, the same diagnosis solution is applicable to all applications interacting with the instrumented middleware. Alternatively, by instrumenting lower layers of the I/O flow, such as the SCI, it becomes applicable to all applications interacting with the POSIX Application Programming Interface (API), even if these use different middleware libraries to do so.

However, instrumenting lower layers provides less information (context) about the actual requests done by specific applications. Namely, when traversing the I/O stack, requests are exposed to several transformations, like I/O merging [127]. Thus, when intercepting requests at lower layers, such as the file system or the block device, it may not be possible to uncover the concrete user space I/O operations that originated such requests, and it may be hard to distinguish the processes that originated the requests. Alternatively, by instrumenting the SCI layer, one can observe the actual user space operations (*i.e.*, system calls of syscall for short) made by each application to the kernel (*e.g.*, `write`, `read`, `send`, `receive`). It can also be possible to obtain information about the process and/or thread that generated each request. However, it still does not allow understanding the application's logic behind a request, for instance, which application function or method generated the intercepted syscalls. When such context is required, diagnosis solutions must perform instrumentation at the application layer.

Next, we discuss the different approaches followed by current diagnosis tools for collecting data at four different levels: application and system logs (§2.1.1), application instrumentation (§2.1.2), middleware instrumentation (§2.1.3), and OS instrumentation (§2.1.4).

2.1.1 Application and System Logs

When an application experiences a failure or displays inefficient or erroneous behavior, the common procedure followed by users is to inspect application or system logs. These can provide various types of information, including program variable values, invoked program functions, error messages, *etc*.

Since manually analyzing and correlating large and often unstructured logs generated by data-centric applications is a laborious and error-prone task, several solutions have been proposed to automate the process of data extraction from logs. More precisely, these solutions aim at automating the parsing of each log message by identifying their constants (*i.e.*, fixed text written by developers) and variables (*i.e.*, values of program variables that carry dynamic runtime information), and generating event templates that allow transforming unstructured logs into structured data [56].

Handcrafted Regular Expressions. The traditional approach is to rely on handcrafted regular expressions or ad-hoc scripts that separate log messages into different groups, where log messages in the same group share the same event template [56]. Logstash [10] is an example of a log parsing tool that uses a set of regular expressions defined by the user for parsing logs. Some tools, such as LongLine [126], a visual analytics system for large-scale audit logs, rely on Logstash to parse log data.

Other solutions like Falcon [82] and Horus [84], two log-based analysis tools for distributed systems, require the existence of dedicated drivers for each type of log (*e.g.*, Log4j, TShark), which are responsible for translating the library-specific log entries into events that can be effectively processed by these tools.

Although these approaches facilitate log parsing, they require users to manually write the rules for processing each log message, which is still a time-consuming and error-prone process. Moreover, the logging code in modern applications is constantly changing (*e.g.*, by adding new logging messages or changing the log message structure), forcing users to regularly update their parsing rules [55, 123].

Static Analysis. Another broadly adopted approach to ease log parsing is to statically analyze the applications' source code. Solutions like Xu et al. [124] and SherLog [128] use static analysis to identify logging statements in the applications' source code and extract all possible format strings that can appear in log messages. These format strings are then used to automatically generate a collection of regular expressions for parsing the logs. While these solutions can automatically generate rules for parsing logs, they are not always applicable as the applications' source code may be unavailable (*e.g.*, third-party components).

Iprof [131] overcomes this issue by statically analyzing the applications' binary code, searching for logging statements with keywords (*e.g.*, *fatal*, *error*, *warn*, *info*) often employed by commonly used logging libraries (*e.g.*, Log4j and SLF4j). With these statements, Iprof generates the regular expressions for log parsing. However, this solution is highly dependent on the logging libraries and programming languages used by the applications and, therefore, hard to generalize.

Data Mining Techniques. More automated log parsing solutions employ data mining techniques for extracting log templates and splitting raw log messages into different groups. For instance, LogMine [54] follows a clustering-based approach, grouping log messages with high similarity into the same cluster, and extracting, for each cluster, a representative log template. LogLens [36], a real-time log analysis system, leverages LogMine to cluster preprocessed logs.

2.1.2 Application Instrumentation

While logs can provide valuable information about the behavior of applications, these often fail to provide sufficient information for detailed diagnosis [129]. In fact, the data being registered is significantly influenced by the type of application, the chosen logging method (*i.e.*, logging library or ad hoc messages), and the intended purpose of the log. For instance, an error log message may only indicate a system failure without providing further context. Furthermore, while simple log messages can assist developers in their tasks, they may not align with the end user's diagnosis needs. Updating applications to log more comprehensive data can have undesired effects on performance and information overload [75]. These reasons led several tools to use source code or binary instrumentation, as it provides greater flexibility when tracing applications' requests and allows gathering a more comprehensive set of information to aid in the diagnosis process.

Source Code Instrumentation. Zipkin [132] and Jaeger [62] are two examples of commercial distributed tracing systems that rely on source code instrumentation. While Zipkin offers a collection of libraries for adding instrumentation code to applications, Jaeger suggests users to use OpenTelemetry [87], a collection of APIs, Software Development Kit (SDK)s, and tools for instrumenting, generating, collecting, and exporting telemetry data (*i.e.*, metrics, logs, and traces) for software's performance and behavior analysis. The added instrumentation code allows collecting data about applications' requests, such as timestamps at which they occur, duration, type of operation, and additional context added by users.

Magpie [13], a tool chain for automatically extracting systems' workloads under realistic operating conditions, relies on instrumentation for capturing control paths and resource demands of applications' requests as they are serviced across components and machines in a distributed system. Namely, it uses the Windows event logging infrastructure, Event Tracing for Windows (ETW), to add custom event tracing to application-level components, such as *ASP.NET ISAPI*, and middleware components, such as *WinSock2*. The collected information includes events with a timestamp, an event identifier, and the values of zero or more typed attributes (e.g., Thread Identifier (TID), CPU ID). For accounting the thread CPU consumption and disk I/O usage of applications, Magpie combines user space tracing with kernel-level tracing.

Stardust [115], a tracing infrastructure for distributed storage systems, adds instrumentation code to strategic locations in the application's code for capturing the demand of a specific resource (*e.g.*, when a request is sent to disk and then again when it completes). Each generated event contains a common header and a payload. The headers include a timestamp, a breadcrumb, the kernel-level PID, and the user-level TID. The breadcrumb is used for correlating distinct events associated with a given request within and across servers. The payload format depends on the request type. For instance, an event associated with a disk request contains the disk ID, the logical block number, the size of the I/O request, and the operation type. Similarly to Magpie, along with the applications' source code, Stardust also modifies middleware components (*e.g.*, Remote Procedure Call (RPC) layer) and kernel functions (*e.g.*, *KernelProcessSwitch*).

X-Trace [45], another tracing framework for distributed systems, relies on users to instrument their applications by adding metadata to requests when an application task initializes (*e.g.*, a web request). By propagating the metadata down to lower layers through protocol interfaces, which may need to be modified to carry X-Trace metadata, users can then understand the causal paths of requests in network protocols. The X-Trace metadata may include information about the task identifier, IP options, TCP options and HTTP headers.

Binary Instrumentation. Pivot Tracing [74, 75], which also aims at tracing distributed systems, follows a similar approach to X-Trace, instrumenting applications to add and propagate baggage (*i.e.*, metadata) along the execution path of requests. The generated baggage can include information about the host, timestamp, process name and PID. Unlike X-Trace and previous solutions, Pivot Tracing resorts to dynamic instrumentation through the Javassist [28] library, dynamically rewriting and reloading Java class byte code, which avoids the need to manually modify applications' source code.

2.1.3 Middleware Instrumentation

Instrumenting source code allows obtaining the most precise and detailed information about the applications' requests. Nevertheless, it can be difficult to understand what information one wants to record and where to add the instrumentation code. Moreover, source code instrumentation often requires modifications to numerous files, potentially leading to implementation errors [67]. Besides, the applications' source code may not always be available, which inhibits the use of such an approach.

In line with these challenges, some diagnosis tools apply instrumentation at the level of middleware components. These components can be system libraries or network protocols that are commonly used by applications when interacting with lower layers (*e.g.*, SCI layer) or remote servers.

Examples of solutions that follow such an approach are Pinpoint [24], a framework for root cause analysis on the Java 2 Platform Enterprise Edition (J2EE) platform, and Dapper [106], Google's production distributed systems tracing infrastructure. Pinpoint instruments the J2EE server platform to trace client requests, avoiding modifications at the application level, while Dapper leverages the fact that Google's applications use the same threading model, control flow and RPC system to restrict instrumentation to a small set of common libraries.

Other solutions like ScalaIOTrace [117], Darshan [108], DXT [122] and EZIOTracer [81] intercept applications' I/O calls (*e.g.*, `read` and `write` operations) by instrumenting general-purpose parallel I/O libraries, such as MPI, or system I/O libraries (*e.g.*, libc). Adding instrumentation code to these libraries can be done via compile-time wrappers for statically linked executables through the PMPI interface [59] or the GNU linker, or library preloading for dynamic executables with the `LD_PRELOAD` mechanism.

IOPin [67] uses the Pin [72] tool to instrument the binary code of the MPI library and the underlying Parallel Virtual File System (PVFS). By packing trace information into a structure and passing it into sub-layers at runtime, IOPin is able to track the flow of the I/O call from the MPI library to the PVFS server.

2.1.4 OS Instrumentation

Even though middleware instrumentation allows collecting runtime information about applications' I/O requests without modifying their source code or binaries, it is not generally applicable as some applications may not interact with the instrumented middleware components (as depicted in Fig. 2.2). A more broadly applicable approach is to instead instrument the OS, capturing requests' information at one or multiple layers of the Linux I/O stack.

Ptrace. Strace [109] follows such an approach by leveraging the user-level `ptrace` mechanism to intercept syscalls as well as read and write operations to memory and registers. `Ptrace` is itself a syscall that provides the means for a process (*e.g.*, Strace) to observe and control the execution of another process (*i.e.*, the targeted application) [92]. Whenever a syscall is invoked, the target application is temporarily stopped, and Strace is notified to process the information about the syscall (*e.g.*, capture information about the syscall type, arguments, and return value). Once finished, Strace returns control back to the targeted application, allowing it to resume execution.

Due to this constant context switching between processes, `ptrace`-based solutions like Strace are known for highly impacting the targeted applications' performance [51]. Nonetheless, Strace is widely used by users and developers for debugging their applications, as well as by other diagnosis tools, such as RepTrace [98], to collect information about I/O syscalls.

Custom Kernel Modules. Another approach for collecting information about I/O requests at the OS level is to add instrumentation code inside the Linux kernel. For instance, Sysdig [111] offers a custom Linux kernel module that captures all syscalls coming from applications and sends them to a user space daemon for further processing. Similarly to Strace, Sysdig can capture information about the syscall type, arguments and return value.

Linux Security Module (LSM). S2Logger [110] and CamFlow [89] leverage the LSM to hook onto kernel syscalls. LSM is a framework often used by security systems such as SELinux [107] and AppArmor [15] to enforce security policies on kernel functions [121]. Intending to track file provenance rather than implement security policies, S2Logger and CamFlow use the LSM interface to log file and network-related syscalls, recording information about timestamps, syscall types, process and parent PIDs, user credentials, and syscall arguments.

Linux Trace Toolkit Next Generation (LTTng). Other solutions rely on the LTTng technology to instrument one or more layers of the OS [37]. LTTng is a low-level tracing tool that allows users to instrument *i)* kernel tracepoints – statically defined points in the source code of the kernel image or of a kernel module; *ii)* kernel syscalls; *iii)* kernel probes – probes dynamically placed in the compiled kernel code; and *iv)* user space probes – probes dynamically placed at the entry of a compiled user space application or library function through the kernel.

Whenever one of these instrumentation points is reached (*e.g.*, when a syscall reaches the kernel), LTTng produces a new timestamped event with information (*e.g.*, type, arguments) regarding the intercepted request and writes it to *ring buffers* shared with consumer daemons at user space. A *ring buffer* is a contiguous memory area that can be written (by producers) and read (by consumers) simultaneously. Given its circular layout, when the buffer is full, incoming events may either replace the oldest ones or be discarded until some events have been consumed from the buffer [51]. The consumer daemons collect events from the *ring buffers* and persist them to disk or send them through the network in an optimized binary format called Common Trace Format (CTF).

Re-Animator [5], Kohyarnejadfard et al. [68] and Daoud and Dagenais [32] use LTTng to trace syscalls, capturing information about the process that submitted the request (*e.g.*, PID) and details regarding the request itself, such as its type, arguments, return value and duration. In addition to tracing syscalls, Daoud and Dagenais [32] also use LTTng to instrument other OS layers. Specifically, they instrument the block layer to observe when a request is created, inserted into the scheduler, issued to the disk, and completed. Furthermore, they instrument the disk driver, namely the Small Computer Systems Interface (SCSI) interface to track the I/O requests sent to the controller and verify if they were handled correctly, and the network layer to collect information about network packet exchanges. Finally, they leverage LTTng's capabilities to instrument user space probes as well, intercepting requests from the storage daemons of the Ceph [119] distributed storage system.

While LTTng is known for its small performance overhead over targeted applications, it is not integrated into the mainline Linux kernel. Therefore, to enable kernel-level tracing with this technology, one must load multiple kernel modules [50, 51]. Adding instrumentation code to the Linux kernel code or creating and loading a poorly designed kernel module is always a substantial risk. A minor bug in such code can cause disastrous results (*e.g.*, kernel panic) [4]. Thus, solutions such as Sysdig, which develops its own kernel module, as well as solutions based on LSM and LTTng must take extensive precautions to ensure that their kernel modifications are safe and will not disrupt the stability of the system.

Extended Berkeley Packet Filter (eBPF). The eBPF technology, on the other hand, provides an alternative approach for safely and efficiently instrumenting the kernel without requiring modifications to its source code or the loading of new kernel modules [76]. In particular, eBPF is a Virtual Machine (VM)-based framework that facilitates the injection of byte-code programs into the kernel for extending its functionalities. Before an eBPF program is injected into the kernel, it undergoes validation by the eBPF verifier. This verifier ensures the safety of the program by checking, for instance, for infinite loops, uninitialized variables, memory access out of bounds, and program termination, among others[39].

Once approved, the eBPF program is compiled into native kernel code through a Just-In-Time (JIT) compiler and attached to the defined hook (*i.e.*, tracepoints, user space probes or kernel probes). Whenever these hooks are triggered, the programs gather the desired information and exchange it with user space via *ring buffers* or eBPF maps. The latter are in-kernel data structures (*e.g.*, hash tables, arrays) useful for sharing information between different runs of the same program, different programs or a program and user space.

Considering the aforementioned advantages, Sysdig [111] shifted its core instrumentation technology. Namely, Sysdig now offers an eBPF-based tracer as an alternative to its kernel module to intercept syscalls [112]. Tracee [116] and IOscope [102] are other examples of eBPF-based solutions. Tracee focuses on runtime security and forensics analysis, while leveraging this technology to collect information about syscalls and network events. Moreover, Tracee collects security events that expose more advanced behavioral patterns (*e.g.*, detecting code injection via `ptrace`). IOscope is a tracer for profiling I/O patterns of storage systems' workloads and uses eBPF to collect information about I/O requests that are based on the variations of `read` and `write` syscalls (*e.g.*, `pread`, `pwritev`, `readv`, `preadv`, `pwritev2`). To that end, it intercepts requests at both the VFS and block device layers.

To establish the causal relationship between log events from different machines, Falcon [82] and Horus [84] combine applications' logs with kernel-level events collected with the eBPF technology. Specifically, they use kernel probes to capture *i)* process-related events, including the initiation, termination, forking, and joining of processes or threads, and *ii)* network-related events, such as the establishment and acceptance of network connections between two processes and the sending and receiving of messages.

EZIOTrace [81], given its goal of unifying user and kernel-level storage I/O tracing, also relies on eBPF for kernel-level tracing, combining middleware instrumentation with OS instrumentation. Namely, it uses the latter to keep track of the lifetime of read and write operations through the Linux I/O stack, placing probes at the VFS, page cache, Ext4 file system, and block device layer.

PerSecMon [64], a performance and security-aware monitoring framework, also leverages eBPF to instruments multiple OS layers. First, user space probes are used to capture activity statistics for various high-level language applications (*e.g.*, Java, Python). These probes gather information regarding garbage collection, the entry of a method, the start of a process call, and the completion of a process call. Memory tracepoints are used to inspect the memory and check whether the allocated memory is released or not after the process execution completes. Finally, kernel probes are used to intercept syscalls, VFS operations, block device operations, and generic functions to track the kernel stack.

Although this technology provides a flexible and safe way to instrument the Linux kernel, it has some limitations. For example, each eBPF program has a maximum number of instructions (*e.g.*, 4096 instructions for kernels up to version 5.4 and 1 million instructions for newer kernels), its stack cannot exceed 512 KiB, and only bounded loops are allowed [48]. Moreover, similar to LTTng, since the *ring buffers* used to exchange data from the kernel to user space have a circular layout, whenever the buffers are full, eBPF starts discarding events.

2.2 Data Analysis

As mentioned before, a crucial phase of the diagnosis process is the analysis of collected data. Given the substantial volume of events produced each second by the systems under tracing (*e.g.*, distributed systems, data-centric applications), it is crucial to employ automated strategies to process these events, extract relevant information from them, and efficiently persist the resulting data (*i.e.*, collected data and the outputs of the analysis algorithms).

We now review existing diagnosis tools based on their main analysis purpose (§2.2.1), data processing algorithms (§2.2.2), and backends (§2.2.3).

2.2.1 Purpose

As shown in Table 2.1, existing diagnosis tools can be grouped into two major topics: *i)* profiling and anomaly detection, and *ii)* root cause analysis.

Profiling and Anomaly Detection. Numerous solutions aim at profiling applications to gain insights into their typical behavior. Such knowledge can then be leveraged to identify optimization opportunities and discover irregularities or anomalies in applications' executions.

In this context, many solutions focus on analyzing the applications' performance [13, 32, 45, 62, 64, 67, 68, 74, 75, 106, 115, 117, 122, 124, 131, 132]. For instance, Magpie [13] proposes a solution for automatically extracting a system's workload, allowing to understand, for instance, how requests are serviced. Stardust [115] provides information about clients' request latency, showing where a request spends its time as it is processed in the system. Similarly, Dapper [106] can determine which part of a system is experiencing slowdowns, and IOPin [67] can understand the complex interactions across different I/O layers from applications to the underlying parallel file system. The solution proposed by Xu et al. [124] is able to detect performance anomalies such as a disproportionate number of aborting transactions, or transient workload imbalance. Kohyarnejadfard et al. [68] can reveal anomalous subsequences of syscalls based on their execution times and frequencies.

Other solutions apply profiling and anomaly detection for security analysis. LogLens [36] proposes a real-time log analysis system for discovering security attacks such as spoofing attacks. LongLine [126] provides a visual analytics solution for large-scale audit logs that allows detecting unexpected malfunctions of systems or attacks against these. For example, through the analysis of audit logs collected from a global Information Technology (IT) company, LongLine allowed the detection of unusual configuration file changes on the data storage server. S2Logger [110] monitors systems' executions to provide near-real-time detection of data-related security policy violations such as data loss and leakage. Similarly, Tracee [116] provides a runtime security and forensics tool to help uncovering malicious activities. For instance, it has the capability to detect anti-debugging methods employed by malware to remain concealed and obstruct its detection, identify code injection techniques utilized for executing malicious code, and pinpoint the utilization of the `LD_PRELOAD` mechanism to alter applications' behavior or load malicious programs. PerSecMon [64] combines both performance and security analysis, providing a solution for finding issues that may lead to performance degradation and unveiling security vulnerabilities within the system.

Root Cause Analysis. Another common purpose of diagnosis tools is the root cause analysis of a detected failure or anomaly. SherLog [128], for instance, combines source code analysis with information collected from applications' logs to identify the underlying reasons for software bugs (*e.g.*, Apache web server incorrectly handling EOF in the response stream when set up as a proxy server) and configuration errors (*e.g.*, CSV1 version control server incorrectly setting the permission for locking a directory). By tracing requests as they travel through the system, Pinpoint [24] is able to automatically identify the root cause of single-component failures, for instance, in e-commerce environments.

Falcon [82] and Horus [84] show that by combining log analysis with causality tracking, it is possible to explain the reason behind unexpected behaviors in dependable distributed systems. RepTrace [98] also leverages causality analysis to identify the root cause of unreproducible builds.

2.2.2 Algorithms

Diagnosis tools employ a wide variety of strategies to analyze collected data. While several solutions use statistical methods to provide metrics about the targeted system, others opt for different algorithms and models to perform mathematical analyses and establish correlations within the observed data. A different set of tools rely on data mining and Machine Learning (ML) techniques, especially for detecting anomalies. Finally, a considerable number of solutions leverage graph-based approaches to analyze collected events and data dependencies.

Statistical Methods. Several diagnosis solutions rely on statistical methods to provide users with statistics obtained from the data collection phase [62, 64, 67, 106, 117, 122, 131, 132]. For example, IOPin [67] provides throughput and latency performance statistics for each layer of the kernel I/O stack.

ScalaIOTrace [117] replays collected traces and uses analysis interposition functions to obtain statistics on the number of I/O operations and blocking/non-blocking communication calls across all nodes of an High-Performance Computing (HPC) infrastructure. PerSecMon [64] relies on analytics features to generate prompt messages depending on users' requirements (*e.g.*, set a flag for processes with latency higher than a given threshold).

Pivot Tracing [74, 75] correlates metrics and events from arbitrary points in the system at runtime (*e.g.*, understand the disk bandwidth usage across a cluster of nodes, on a per client request basis). This is done by propagating metadata (*e.g.*, process name) along with the events and leveraging their causal relationships to implement a happens-before join operator that groups events occurring within and across process, machine, and application boundaries.

Algorithms and Models. Other solutions apply different algorithms and models to analyze collected data. SherLog [128], for example, tries to infer execution paths from applications' logs and static code analysis, while relying on a satisfiability constraint solver to prune infeasible paths. In a similar way, Falcon [82] models the happens-before relationships between events and uses a Satisfiability Modulo Theories (SMT) solver to yield an execution schedule in which events are guaranteed to be causally ordered.

Daoud and Dagenais [32] employ a fully incremental convex hull algorithm to synchronize user space traces from Ceph's storage daemons with kernel-level traces from various layers of the I/O stack, based on the causality of events. LogLens [36] detects malfunctioning events by analyzing abnormal log sequences based on a Finite-State Automaton (FSA) model.

Tracee [116] employs behavioral pattern matching methods to analyze and compare collected events with a predefined set of malware behavioral signatures, alerting users when a potential threat is found.

Data Mining and Machine Learning. Another analysis strategy is to leverage data mining and ML techniques. For instance, Xu et al. [124] use the Principal Component Analysis (PCA) anomaly detection method to isolate repeating patterns in feature vectors and make abnormal log message patterns easier to detect.

Kohyarnejadfard et al. [68] introduce a supervised anomaly detection method built on a multi-class Support Vector Machine (SVM) classification model. Additionally, their work presents two other anomaly detection techniques, one unsupervised and the other semi-supervised, both using the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm. Similarly, LongLine [126] employs unsupervised non-linear dimensionality reduction and anomaly detection techniques, including the t-distributed Stochastic Neighbor Embedding (t-SNE) and the Local Outlier Factor (LOF) algorithms, which are used to identify unexpected malfunctions of applications or attacks over these.

Pinpoint [24] uses the Unweighted Pair Group Method with Arithmetic Mean (UPGMA), a hierarchical clustering algorithm, to correlate the failures and successes of requests and determine which application's components are most likely to be faulty. Magpie [13] identifies I/O events belonging to the same client request by applying a form of temporal join, and then employs behavioral clustering to build workload models of applications.

Graphs. Graph-based analysis is also a frequently utilized approach. Horus [84] uses Directed Acyclic Graph (DAG) to explicitly encode causality, where nodes represent events and edges indicate the causal dependencies between them. Similarly, RepTrace [98] uses dependency graphs to conduct causality analysis over syscalls and identify the root causes for unreproducible builds.

X-Trace [45] leverages the causal relationships obtained from metadata propagation and builds a graph that provides users with a comprehensive view of all the network operations executed as part of a given client request. Likewise, Stardust [115] builds latency graphs that allow observing how requests flow from component to component in the distributed system and where requests spend their time.

S2Logger [110] generates direct graphs to obtain an end-to-end overview of the data flow in distributed virtualized environments. By using these graphs, S2Logger is able to perform real-time enforcement of data protection policies.

2.2.3 Backends

The type of backend used to persist collected data, along with the output of its analysis, varies across diagnosis tools. While some solutions store this information on disk, others rely on relational and non-relational databases.

Disk. The majority of solutions store collected data directly in one or multiple files [13, 24, 68, 82, 98, 116, 117, 122, 124, 128]. The analysis phase is then conducted over such file(s), while the outputs are stored on separate file(s) or shown through visual representations. To conduct different analyses, these solutions may need to iterate over the full content of file(s) multiple times.

Daoud and Dagenais [32] follow a different approach by using the *modeled state system* [78, 79], a disk-based data structure that keeps the state of the system in a tree-like fashion, allowing for its efficient storage and access.

Relational Databases. Other tools store diagnosis-related information in relational databases, which provide stronger data consistency and integrity, while offering the flexibility to execute Structured Query Language (SQL)-based queries over stored data. Examples of used relational databases include SQLite [67, 115], MySQL [110], and PostgresSQL [45].

Non-Relational Databases. Non-relational databases can accommodate various data types (*e.g.*, key/value pairs, JSON documents, graphs), efficiently manage large-scale distributed data, and provide good performance for specific use cases, such as real-time processing [63]. Examples of non-relational databases utilized by existing diagnosis tools encompass document-oriented databases like Elasticsearch [36, 64] and MongoDB [131], Key-Value Store (KVS) such as Cassandra [126], tabular databases such as BigTable [106] and graph databases like Neo4j [84].

Multiple Backends. In some cases, diagnosis tools provide support for multiple database types. For instance, Zipkin [132] uses Cassandra (a non-relational KVS) as the default storage backend. However, it has native support for Elasticsearch (a non-relational document-oriented database) and MySQL (a relational database). Similarly, Jaeger [62] has multiple built-in backends such as Cassandra and Elasticsearch, and has support for several other backends, including InfluxDB (a non-relational time-series database), ScyllaDB (a non-relational wide-column data store), and PostgreSQL (a relational database).

Postmortem vs. Real-time Analysis. Besides the aforementioned features, the chosen backend also influences the timing for conducting the analysis phase. Specifically, while some database-based solutions are capable of delivering inline (*near-real-time*) analysis [36, 64, 84, 106, 110, 115, 126], the majority of disk-based solutions only mention support for offline (*postmortem*) analysis. The only exception is Tracee [116], as its analysis process occurs simultaneously with the data collection process. Furthermore, among the six database-based solutions that offer near-real-time analysis, only two of them employ relational databases. Finally, Pivot Tracing [74, 75] is the only solution that does not include a storage backend, which explains its lack of support for postmortem analysis.

2.3 Data Visualization

Although some tools analyze collected data and deliver the corresponding results in a raw format (*e.g.*, through new files or messages printed in the standard output) [13, 24, 67, 74, 75, 116, 117], others defend that providing analysis results through visual representations facilitates the diagnosis endeavor for users. Next, we overview the different types of visual representations offered by current solutions (§2.3.1) and the tools employed to create these visualizations (§2.3.2).

2.3.1 Type

According to their analysis' purpose, solutions may provide a single or multiple types of visual representations.

Single Visualization. Falcon [82] and Horus [84] employ space-time diagrams to visually depict the interactions between different components in distributed systems and the causal dependencies between their events. Xu et al. [124] use decision trees to pinpoint types of abnormal behaviors and help users quickly understand the anomaly detection result.

Other works rely on graph-based representations. RepTrace [98] uses dependency graphs to show the root cause of unreproducible builds. X-Trace [45] uses task trees to depict all sub-operations of a given main task (*e.g.,* client request), and Stardust [115] depicts where clients' requests spend their time through latency graphs. SherLog [128] and S2Logger [110] use control flow and data flow graphs to visually represent how requests and data flow through the targeted system.

Multiple Visualizations. A different approach is to provide multiple visual representations that best fit the heterogeneity of the information being observed. For instance, Dapper [106] provides a web-based user interface that allows observing tables with performance summaries, frequency histograms over selected metrics (*e.g.*, execution's latencies), and trace trees, depicting the causal and temporal relationships between spans (*i.e.*, different events originated by the same request). Zipkin [132] and Jaeger [62] offer web interfaces with representations for visualizing individual spans, including dependency diagrams and Gantt charts. Iprof [131] supports a web-based application with histograms and time-series graphs to visualize requests' latency over time, request count and trend over time, and average latency per node.

LogLens[36], PerSecMon[64], DXT Explorer [18] and Kohyarnejadfard et al. [68] provide user interfaces with different types of visualizations, ranging from simple representations such as tables, pie charts and histograms to more elaborate ones such as time-series and heat maps.

Besides supporting simple representations, Daoud and Dagenais [32] developed three types of visualizations tailored to observe their analysis findings. These include the *Object Storage Devices (OSD) activity view* that shows the throughput of the storage devices supported by the cluster, the *Ceph processes view* that provides low-level details about Ceph processes, and the *Network view* that summarizes the network exchanges that happen between the different cluster nodes. LongLine [126] also includes representations specific to their analysis' goal, such as calendar views and two-dimensional visualizations. These ease the work of security analysis when searching for abrupt changes in the large number of collected logs and identifying anomalies by comparing the daily I/O patterns of observed applications.

2.3.2 Tools

While some diagnosis solutions rely on pre-existing visualization frameworks to customize their visual representations, others rely on libraries to build their own visualization component.

Visualization Libraries. Falcon [82], Iprof [131] and LongLine [18] implement their visualizer components as JavaScript programs, using libraries such as SVG.js, Highcharts, and D3.js, respectively. DXT Explorer [18] implements its web-based representations in R, resorting to the ggplot2 and Plotly libraries.

Visualization Frameworks. An alternative approach is to integrate existing visualization frameworks with the diagnosis pipeline. These frameworks already provide visual representations, which can then be customized by developers while avoiding the need to implement them from scratch through libraries.

Stardust [115] uses GraphViz, a graph visualization software, to build latency graphs, and Horus [84] relies on ShiViz, a visualization engine that generates interactive communication graphs from distributed system execution logs, to depict space-time diagrams. Kohyarnejadfard et al. [68] and Daoud and Dagenais [32] use Trace Compass, a Java framework for observing the information contained at logs and traces through both general and specific representations. LogLens [36] and PerSecMon [64] use Kibana, a framework for visualizing data stored at Elasticsearch through dashboards that assist in the data analysis and exploration process.

Interactive Visualizations. Several solutions further ease the diagnosis process through interactive visualizations. These can include highlighting specific areas of representations (*e.g.*, of tables, pie charts, time-series graphs) to obtain more details about these. New visual representations can even be generated automatically to further explore the selected information. From the solutions based on visualization frameworks, only Stardust [115] does not provide this functionality. As for the approaches using custom visual components, only five out of eleven support interactivity [18, 62, 106, 131, 132].

2.4 Lessons Learned

We now summarize key insights provided by this chapter. Learned lessons are provided for the different main tasks of the diagnosis process, namely data collection (***C****x*), analysis (***A****x*) and visualization (***V****x*).

C1: Transparency. Source code instrumentation is often used to gather information about I/O requests. This intrusive approach requires a considerable manual effort when applied to large codebases and/or several components, being only applicable when source code is available. Ideally, data collection would be non-intrusive, treating applications as opaque boxes and requiring the least possible knowledge about the targeted system.

C2: Applicability. Data collection can be done at different levels of the I/O stack (*e.g.*, applications, libraries, middlewares, OS). The choice on which levels to consider must be based on: *i)* the diagnosis purpose(s); *iii)* the amount of context needed regarding the original application's requests; and *iii)* how much one needs the diagnosis solution to be transparent and generally-applicable.

C3: Comprehensiveness. The detail of collected data is highly related with the analysis purpose of each tool. Consequently, for multi-purpose diagnosis tasks, users need to combine several tools and repeat the data collection process multiple times. Ideally, diagnosis tools should capture comprehensive information from I/O requests to enable a more efficient and richer study of these.

C4: Performance Impact. Intercepting I/O events requires extra processing in the critical path of requests. Depending on the I/O stack level where these are intercepted and the tracing technology being used (*e.g.*, `ptrace`, eBPF, LTTng), one can add undesired performance overhead over the application, and even hide subtle concurrency issues such as I/O contention or starvation [51]. Data collection should aim at reducing performance overhead over applications to support timely and accurate diagnosis.

C5: Resource Usage. The amount of captured information and the chosen tracing technology also impact the usage of system's resources (*e.g.*, CPU, RAM, disk). Having resource-efficient data collection is important to lower I/O diagnosis' hardware requirements and to avoid competition between targeted applications and tracers for system's resources, which can affect the traced information's accuracy.

C6: Accuracy. To reduce tracing performance overhead and resource usage, many solutions wittingly discard sets of I/O requests, for instance, by using sampling [106]. However, missing rare but important requests can have a direct impact on the analysis' accuracy [75]. Therefore, diagnosis solutions must strive to find an appropriate balance between accuracy, performance overhead and resource usage.

C7: Flexibility. When diagnosing applications, one might want to capture as much information as possible to explore unknown I/O behaviors, or capture only events of interest to debug specific issues. Therefore, data collection should be flexible regarding the amount of intercepted requests and the detail of information captured from these to accommodate these different needs.

A1: Automation and Summarization. While efficient data collection is important for diagnosing applications, by itself it is not sufficient. Given the potentially large number of collected events, solutions must support automatic strategies to analyze such events, while highlighting useful insights about these in a concise fashion.

A2: Multi-purpose. Current tools resort to a vast set of algorithms, which is explained by their different and specific analysis requirements. For example, ML is widely used for anomaly detection, while graphs and solvers are often used to infer the causal order of distributed requests. Ideally, multi-purpose diagnosis solutions should provide support for these different algorithms.

A3: Real-time and Postmortem. Real-time analysis is useful for users to debug and explore their applications in a timely fashion. At the same time, saving collected data for posterior analysis is key for its postmortem exploration, for example, to try out other algorithms or compare the I/O behavior of different applications. Therefore, solutions should aim to support both real-time and postmortem analysis.

V1: Informative. Most of the existing tools avoid generic visualizations, focusing instead on one or more visual representations aligned with their diagnosis goals and analysis algorithms. This is an important feature as it allows users to focus on the most relevant information and easily interpret the analysis' findings.

V2: Versatility. It is hard, if not impossible, to find a single visualization that accurately depicts collected data and the outputs of multiple analysis algorithms due to their potential heterogeneity. Therefore, solutions should support multiple representations tailored for different types of information (*e.g.*, tables, charts, time-series graphs).

V3: Exploration and Customization. Current solutions based on frameworks like Kibana and Trace Compass enable users to build new visual representations and customize existing ones. Also, these allow users to interact with the representations (*e.g.*, filter or select specific time frames). These features are important for diagnosis tasks with a more exploratory nature.

3

Content-aware Tracing and Analysis for Distributed Systems

The development, configuration, and management of distributed systems are usually difficult, costly, and challenging tasks. A distributed deployment can easily become a complex system due to the heterogeneity of software and hardware components, diversity of protocols, programming models and interfaces, sheer concurrency, asynchrony of events, faults, *etc*.

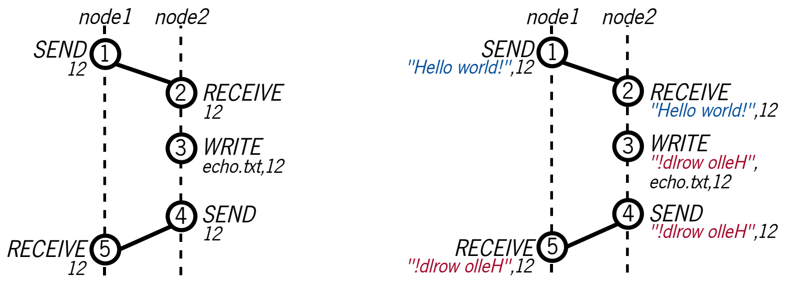
Diagnosis frameworks can assist these tasks by facilitating the observation of the applications' I/O requests as they propagate through the distributed system, and by providing valuable insights into how the system's state evolves over time. Such knowledge is key for performance analysis, diagnosing anomalies, and even for assessing correctness or security properties [86]. However, to efficiently diagnose a distributed system, one must consider several of the challenges discussed in §2.4:

Challenge C1. Distributed systems are composed of several components whose source code may be difficult or even impossible to instrument, thus requiring non-intrusive solutions for tracing their I/O operations.

Challenges C4, C5 and C6. As distributed systems may generate a large volume of storage and network I/O requests, one must aim at reducing data collection's (tracing) performance impact and resource usage, while capturing all the relevant information for making an accurate analysis of the system's behavior.

Challenges A1 and V1. Given the size and complexity of the collected traces, one must automate the analysis and visualization of the events contained in these, while preserving their causal order to provide accurate insights about the I/O flows of the system.

Current diagnosis solutions targeting distributed systems either take an intrusive approach, requiring source code or binary instrumentation [24, 74, 106], or only take into account the requests' context [82, 89, 110], such as, in general, timestamps and PID, for network messages their source, destination and protocol, and for files their descriptor, name and offset.



(a) Context-based tracing (b) Content-based tracing

Figure 3.1. Context vs content-aware tracing analysis.

Indeed, the context of requests provides useful insights about different components interactions (*e.g.*, it can tell when a file is written or an application sends data via a socket). As an example, let us consider an echo application sending a message to be persisted in a file on another node, while expecting to receive the same message from that node as the reply. From the captured I/O events, context-based solutions can provide an analysis similar to the one shown in Fig. 3.1a. Namely, one node is sending 12 bytes to another, which stores 12 bytes in file *echo.txt* and replies with another message of 12 bytes, suggesting that the application is acting as expected.

However, we defend that the analysis of the requests' content, transmitted and stored by the system's different components, can further enrich these tools when validating distributed solutions. For instance, by analyzing the requests' contents from the previous example (Fig. 3.1b), it is possible to see that, despite storing 12 bytes and replying with a message of the same size, node 2 is actually storing and sending different contents. This can happen due to data adulteration or corruption, which is not visible when looking only at the requests' context (*i.e.*, type of operation, filename, and size). Therefore, in this chapter, we innovate by also exploring the network messages' payload and contents of storage accesses.

Namely, to address the previously identified challenges, this chapter proposes CaT, a novel framework for analyzing both the context and content of distributed system's I/O requests. CaT is the first framework to combine: *i)* kernel-level tracing tools to capture the context and content of network and storage events in a non-intrusive fashion; *ii)* summarization and similarity-based techniques to efficiently correlate the content of captured events and visually depict their data flows. In detail, this chapter makes the following contributions:

Content-based Tracing. A novel algorithm that captures and analyzes the context and content of applications' I/O requests. It resorts to hashing techniques to summarize the requests' content while reducing storage space overhead and applies near-duplicate detection algorithms to find similarities between data of distinct distributed events. By performing a similarity-based analysis, CaT can identify duplicate data, as well as near-duplicate data (with a high degree of similarity (*e.g.*, > 80%)) that was slightly modified while flowing through different components (*e.g.*, messages that include the same payload but have a different metadata header). This knowledge is key to detecting data adulteration, corruption or leakage for similar I/O messages.

Non-intrusive Tracing. The previous algorithm is integrated with two kernel-level tracing tools (Strace [109] and eBPF [77]) for capturing storage and network I/O requests in a non-intrusive fashion. These two technologies provide different tradeoffs in terms of resources usage (*e.g.*, CPU, RAM and disk space), accuracy (amount of collected information), and I/O performance. Also, these can filter requests from specific processes or file paths to collect only events of interest.

Pipeline Integration and Prototype. An open-source prototype that provides a fully integrated pipeline to capture, analyze and visualize the context and content of I/O requests. The pipeline design allows decoupling the tracing from the analysis phase, enabling an offline (postmortem) analysis that can even be performed at different and more powerful servers. Therefore, the main focus of this work, and the conducted experimental evaluation, resides on the tracing phase as it has a direct performance impact on the critical I/O path of applications.

Evaluation. A detailed evaluation with two real Big Data applications: TensorFlow [1] and Apache Hadoop [46]. Experimental results show that it is possible to trace how data flows over a distributed system while incurring negligible performance overhead. Moreover, usability experiments demonstrate how CaT can improve distributed systems analysis while adding new and relevant insights on how data is handled in complex multi-node systems. Namely, we show that, with CaT, users can validate the data access patterns performed by TensorFlow when reading the training dataset or verify if the Apache HDFS replicated file system is correctly storing data across the replicas (dependability and correctness). For the latter application we also show that CaT can help identifying erroneous or suspected flows that may lead to security flaws, namely data corruption or adulteration.

All artifacts discussed in this chapter, including CaT, workloads, and scripts are publicly available at https://github.com/dsrhaslab/cat.

3.1 Falcon

In mind with the challenges described earlier, we have selected one of the most recent solutions from the state of the art, named Falcon, to use as the basis for building our framework.

Falcon [82] is a log-based analysis tool for distributed systems whose components operate together as a pipeline, allowing it to combine several logging sources and generate coherent space-time diagrams of distributed events in a non-intrusive way. Its design contains three main components:

Trace Processor. This module is responsible for translating entries from multiple log sources into events to be processed by Falcon. Namely, it can extract useful knowledge about the system execution from logging libraries (*e.g., log4j*) and network sniffers (*e.g., libpcap*-based tools). The extracted information is then organized into process (*fork*, *join*, *start*, *end*) and socket (*connect*, *accept*, *send*, *receive*) events.

Happens-Before Model Generator. After the input data normalization procedure, this module organizes the events according to their logical clocks and their happens-before relationship constraints, building a single causally-consisted schedule. A constraint can, for instance, state that a *send* event must happen-before the corresponding *receive* event.

Visualizer. In the end, the *Visualizer* component generates a space-time diagram depicting both the events executed by each process and the inter-process causal relationships.

By combining the application's logs with kernel-level tracing tools, Falcon can observe the system's behavior, creating causal traces without needing to known the target system's architecture and the interactions among its components.

In CaT, events collected by our novel content-aware tracers are provided to Falcon's Trace Processor. As Falcon can only analyze the context of network requests, its pipeline was extended to provide context and content-aware analysis capabilities for both network and storage I/O requests. These modifications are detailed in the next sections.

3.2 CaT in a Nutshell

CaT is a non-intrusive content-aware tracing and analysis framework for distributed systems that highlights how their components interact with each other and how data flows through the system.

Its design enables the capture of information related to I/O network and storage events, such as the context of the request and the data processed by the event. With this information, CaT proposes an analysis of the events content based on their similarity, allowing the detection of data flow patterns that are not visible when inspecting only the context of events. CaT's design is built over five core principles:

Kernel-level Tracing. CaT resorts to kernel-level tracing tools to capture the context and content of network and storage I/O requests, without requiring previous knowledge about the application or the instrumentation of its source code.

Accuracy vs Performance. CaT's modular design enables the support of different tracing tools, each providing different tradeoffs in terms of the total percentage of collected requests (accuracy), I/O performance, resource usage and storage space overhead.

Summarization. CaT uses hash functions to persist digests of requests' content instead of their full data, thus reducing the storage space of trace logs.

Causality Inference. CaT extends Falcon to correlate and infer the causality of distributed I/O events [82].

Similarity-based Analysis and Visualization. To automate the analysis process, CaT resorts to similarity estimation techniques to compare and highlight data dependencies of complex systems. Also, a color-based scheme is used to visually pinpoint I/O events handling near-similar data.

3.2.1 System Overview

CaT is designed to assist developers and system administrators in analyzing their system behavior and identifying erroneous or suspected I/O flows that may lead to protocol or security flaws.

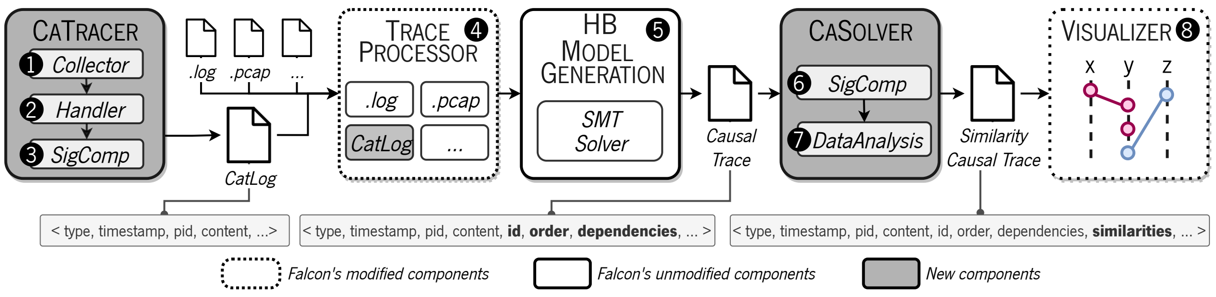

CaTracer
❶ Collector
❷ Handler
❸ SigComp
.log
.pcap
...
CatLog
Trace Processor ❹
.log
.pcap
CatLog
...
HB Model Generation ❺
SMT Solver
Causal Trace
CaSolver
❻ SigComp
❼ DataAnalysis
Similarity Causal Trace
Visualizer ❽
x
y
z
< type, timestamp, pid, content, ...>
< type, timestamp, pid, content, **id**, **order**, **dependencies**, ... >
< type, timestamp, pid, content, id, order, dependencies, **similarities**, ... >
Falcon's modified components
Falcon's unmodified components
New components

Figure 3.2. CaT's architecture.

As depicted in Fig. 3.2, CaT operates as a pipeline that allows combining multiple data sources and assessing the happens-before relationships between events, while adding the functionality of capturing and analyzing their content. First, the *CaTracer* component runs along with the targeted application to intercept its I/O requests and outputs a file (*CatLog*) containing the captured events' information. Then, the *CatLog* file is passed as input to the *Trace Processor* component, which parses the events and shares the information with the *Happens-Before (HB) Model Generator*. The latter causally correlates the events and produces a new file, the *Causal Trace*, containing both the events and their causal relationships. The *Causal Trace* is then passed to the *CaSolver* component, which computes the events' similarities and outputs a *Similarity Causal Trace* file with the inferred similarity information. Finally, a web page is provided to the user through the *Visualizer* component for visualizing the data contained in the latter file (*i.e.*, the events, their causal relationships and their data similarities).

3.2.2 Architectural Components

CaT extends Falcon's architecture for analyzing data in transit and at rest, while providing further information about the targeted system. Next, we detail each of CaT's main components.

CaTracer. The pipeline's first component is the *CaTracer*, which is responsible for collecting I/O events information. It runs simultaneously with the targeted system, observing requests from the different components and storing them as events in a log file (*CatLog*). Its *collector* submodule (❶) resorts to kernel-level tracing facilities to intercept the context (*e.g.*, type of event, timestamp, PID) and content of network (*e.g.*, send, receive) and storage (*e.g.*, read, write) requests in a non-intrusive way. Namely, this component captures the following type of requests: `connect` / `accept` (connection / acceptance of a socket), `send` (SND) / `receive` (RCV) (writing / reading from a socket), `open` (opening a file), and `write` (WR) / `read` (RD) (writing / reading from a file descriptor). Note that the interception of requests is performed at the kernel-level of I/O calls, thus allowing CaT to be used transparently for different applications (*e.g.*, databases, analytical and ML frameworks).

To minimize the tracing performance and storage overheads, the *CaTracer* offers the possibility of saving only events of interest. It can filter events by *i)* PID and *ii)* file path. The former sets *CaTracer* to collect only the events of a given PID and its child processes, discarding all requests that do not belong to them. The latter allows recording only storage events (*i.e.*, `open`, `write`, `read`) that work within a given path or group of paths (*e.g.*, a file or subdirectory). By combining these two filters, *CaTracer* can significantly reduce the number of captured events, saving only the most relevant ones.

The captured information is then sent to the *handler* submodule (❷) that parses and organizes it into the *CatLog* events format. This log file holds the events' type, context, and content. For instance, for the example shown in Fig. 3.1, the resulting *CatLog* file would contain the event: `{"type":"SND", "pid":123, "socket":"TCP", "src":"node1", "dst":"node2", "size":12, "message": "Hello world!"}`. To minimize the resulting log size, *CaTracer* offers the option to compute hash sums of events' content, at the *Signature Computation (SigComp)* submodule (❸). When this submodule is enabled, the *CatLog* file will store the corresponding hash sums instead of the full data buffers' content being intercepted. The CaTracer's submodules are further detailed in §3.3.1.

Trace Processor. After collecting the events, which is done at runtime (*i.e.*, at the target system's critical I/O path), the remaining pipeline initiates the analysis phase that is performed in background and even at different servers. First, the *CatLog* file is forwarded to the *Trace Processor* (❹). This component parses and organizes events into different data structures according to their type. Specifically, a `SocketEvent` data structure groups information about network-related events, including the socket type, source and destination addresses, and the data buffer transmitted. Similarly, a `StorageEvent` data structure gathers information related to storage events, comprising the file's path, descriptor and offset, as well as the data buffer read / written. This component is identical to the one provided by the Falcon solution, with the exception of some minor design modifications to support the *CatLog* file as input and to include the parsing of storage events metadata (*e.g.,* file's path, descriptor and offset).

Happens-Before (HB) Model Generator. The next step is to find the happens-before relationships between the events, which is done at the *HB Model Generator* (❺). This component accesses the data structures (in memory) created by the *Trace Processor* and combines the events into a single causally-consistent schedule. With the aid of an off-the-shelf SMT solver, the *HB Model Generator* outputs a new file (*Causal Trace*) with an identifier for each event (ID), the order it happened, and its dependencies (*e.g.*, the ID of the network `send` event from which a `receive` event depends on). For instance, the *Causal Trace* of the example from Fig. 3.1 would indicate that the RCV events with ID 2 and 5 depend on the SND events with ID 1 and 4, respectively, and that the events from *node2* happened after event 1 and before event 5. This component is identical to the one provided by Falcon without any design modifications required.

CaSolver. The *Causal Trace* is then forwarded to the *CaSolver* module, which analyzes the events' content. The module selects the content for each event, which can either be signatures (hash sums) that were provided by the tracer (❸), or the full data buffers. In the latter case, the *CaSolver* module resorts to its *SigComp* submodule to compute buffers' signatures in place (❻). By having a *SigComp* submodule in the *CaSolver* component, we allow using CaT with third-party log sources that cannot provide *a priori* the events' content signatures. After obtaining all the signatures, the *CaSolver* relies on the *DataAnalysis* submodule (❼) for applying data similarity estimation algorithms to find events with a high probability of operating over the same data. These algorithms are further detailed in §3.3.2. The inferred similarity information (*i.e.*, list of similar events) is then added to the original *Causal Trace* data, producing the *Similarity Causal Trace*. For the example from Fig. 3.1, the *CaSolver* would indicate that events 1 and 2 have 100% of similarity between their content as well as events 3, 4 and 5.

Visualizer. The pipeline's last component is the *Visualizer* (❽), which receives the *Similarity Causal Trace* file and builds a space-time diagram representing the targeted system execution, the events causal relationships and their data flows. A more detailed description of the *Visualizer* is provided in §3.3.3.

3.3 Algorithms and Prototype

CaT's open-source prototype is based on the Falcon project (commit #997b531 [83]). As depicted in Fig. 3.2, the latter was extended to include the new *CaTracer* and *CaSolver* components, while the *Visualizer* was modified to provide a visual representation for content flow across I/O events. Next, we detail these novel functionalities: content-aware tracing (§3.3.1), similarity-based data analysis (§3.3.2) and content flow visualization (§3.3.3).

3.3.1 Content-aware Tracing

CaT's prototype supports two implementations of the new *CaTracer* component, one based on the Strace tool (*CatStrace*) and the other based on the eBPF technology (*CatBpf*). These two tracing technologies were chosen since they provide different tradeoffs regarding accuracy, I/O performance, and resource usage, as shown in §3.5.

CatStrace. The Strace-based tracer uses Strace to trace an application's execution, capturing network (*e.g.*, `connect`, `recvfrom`) and storage-related (*e.g.*, `openat`, `pwrite64`) syscalls, and then parses its output into a *CatLog* file. Fig. 3.3 shows CatStrace's components.

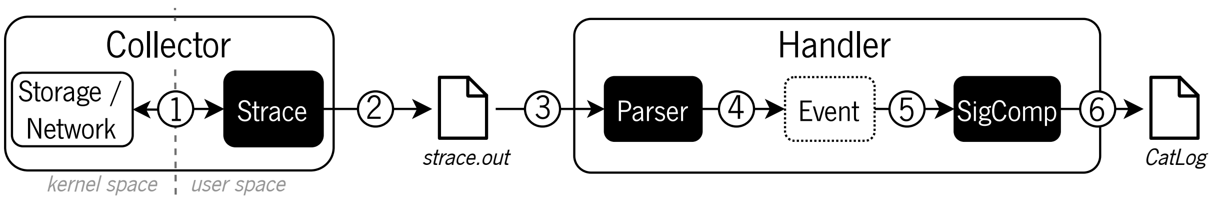


Figure 3.3. *CatStrace*'s components.

The *collector* module spawns a process that runs Strace for a given command or PID. Strace intercepts the syscalls issued by the traced process (①) and saves them to a file (strace.out) (②). The collected information is then parsed by the *handler* module (③). Specifically, a *parser* submodule starts by producing, for each syscall, a generic JSON structure with the type (*e.g., pwrite64*), timestamp, PID, arguments (*e.g.*, filename, buffer, size, offset), and the return value (*e.g.*, number of bytes written). With this information, the *handler* is then able to generate the corresponding events structures (④). Then, each event structure goes through the *SigComp* submodule (⑤) that checks if it has content and resorts to the MinHash algorithm (described later in §3.3.2) to compute the content's signature. The event with its content signature is then persisted into the *CatLog* file (⑥).

CatBpf. The eBPF-based tracer relies on the eBPF technology to capture process (*e.g.*, `fork`), network (*e.g.*, `sendmsg`) and storage (*e.g.*, `write`) requests. As pictured in Fig. 3.4, it has a *collector* module that runs at kernel space and an *handler* module that runs at user space.

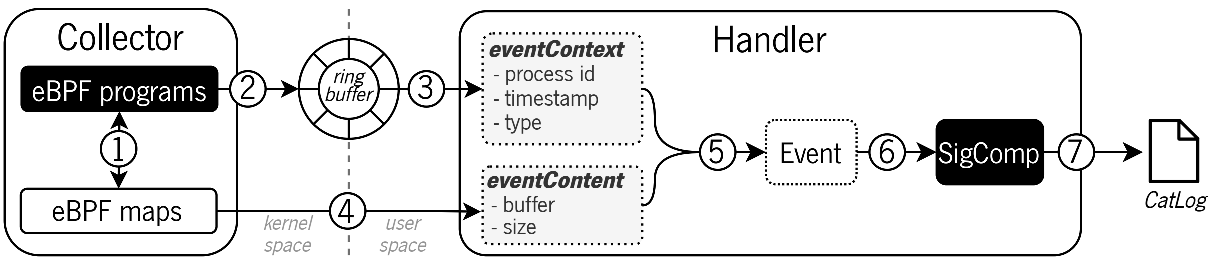


Figure 3.4. *CatBpf*'s components.

The *collector* module contains an eBPF program that defines the code to run when an I/O request (*e.g.*, write) is intercepted. Namely, it first checks if the request was issued by the targeted process and builds a structure (`eventContext`) that gathers contextual information (*e.g.*, type, PID, timestamp). If the request is handling data (*i.e.*, has content), for instance a write request persisting a buffer to disk, the *collector* builds another structure (`eventContent`) that gathers the data buffer and its size.

`EventContent` structures are placed in an eBPF map of type per-CPU array (①) that can be accessed from user space, while `eventContext` structures are submitted to user space via a *ring buffer* (②). At user space, the *handler* is continuously polling events' context from the *ring buffer* (③) and, when applicable, gets their content from the per-CPU array (④). It then merges all the collected data into an `Event` structure (⑤), computes its signature (⑥), and persists it to the *CatLog* file (⑦). CatBpf's *SigComp* submodule is similar to the one from CatStrace.

Our strategy of splitting the context and content of events into two different structures is based on that of unixdump[38] to reduce event loss. As the *ring buffer* (data structure used to submit the events from kernel to user space) has a circular format and a fixed size, once the buffer is filled, the *collector* module starts rewriting the buffer from the beginning. If the *handler* module cannot process events at a fast pace, the *ring buffer*'s data can be overwritten or lost. From preliminary experiments, we observed that the larger the size of the structure submitted to the ring buffer, the higher the percentage of lost data. Therefore, by splitting the events' content from the context, we can submit a smaller structure (`eventContext`) to the *ring buffer* and access the corresponding `eventContent` directly via the per-CPU array.

Although this separation allows decreasing the number of lost events, it can result in incomplete events. Namely, the number of elements on the per-CPU map has to be statically defined due to the limitations imposed by the eBPF *Verifier*. Thus, once the map positions are filled, the old ones start being overwritten. If, after collecting the request's context, the *handler* cannot access the specific `eventContent` in time, the event is persisted only with the context details. Even though this approach can lose the events' content, it can still capture their context, thus enabling a context-based analysis.

3.3.2 Similarity-based Data Analysis

The similarity-based analysis of events' content is performed in two phases: *i)* the signatures computation (at the *SigComp* submodule), and *ii)* the processing of events' signatures (at the *DataAnalysis* submodule), as depicted in Fig. 3.5.

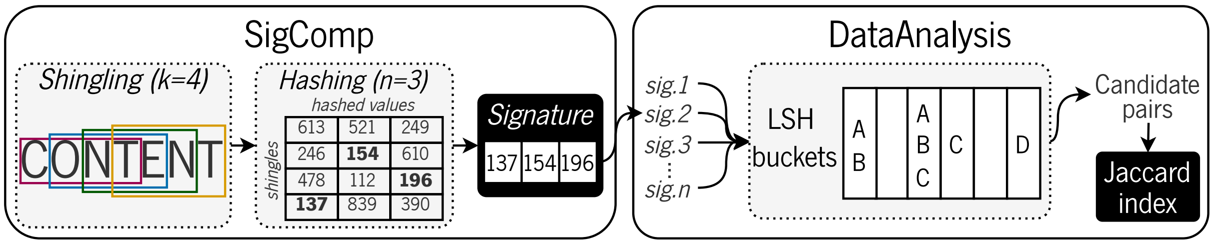


Figure 3.5. *CaSolver*'s components.

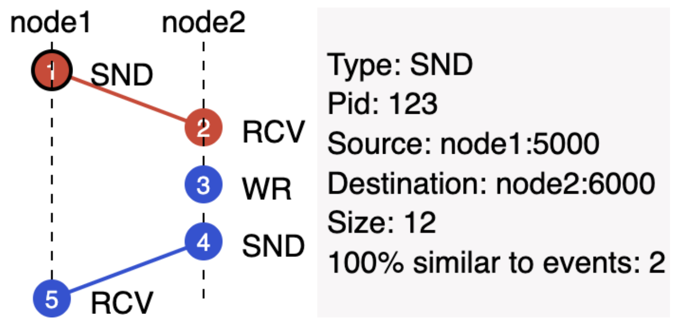
In the first phase, the *Minwise Hashing (MinHash)* algorithm is used to summarize the content of each I/O event into a small set of signatures [21]. In a nutshell, the MinHash algorithm applies n different hash functions to each shingle (*i.e.*, consecutive overlapping sequences of k bytes) of the content buffer. Then, for each hash function output it is selected the smallest value, resulting in a signature with n values.

To assess the similarity between the content of two events, at the second phase, we calculate the Jaccard index of their signatures, which determines the percentage of identical values present in them [52]. However, computing the Jaccard index for all signature pairs (*i.e.*, all events captured by CaT) is a costly operation, whose complexity increases exponentially with the number of signatures to compare. Thus, to efficiently compare all MinHash signatures and find the pairs with a similarity greater than a given threshold, we rely on the Locality-Sensitive Hashing (LSH) algorithm [60]. This algorithm uses several hash functions to group MinHash signatures referring to similar content into the same bucket. This way, the Jaccard index is only computed for strong candidate pairs (i.e., signatures placed at the same bucket).

In the end, the *CaSolver* outputs a list of tuples indicating the ID of similar events and their Jaccard index. Such information will allow the *Visualizer* to represent the events data flow and dependencies. Namely, by looking at the Jaccard index, the *Visualizer* can highlight both identical data (*i.e.*, 100% similar) and similar data (*e.g.*, 80% similar) that have undergone slight changes when flowing across components (*e.g.*, addition of metadata headers).

3.3.3 Content Flow Visualization

The *Visualizer* generates a space-time diagram depicting the events executed in each host and the inter-host causal relationships. Moreover, by relying on the similarity information computed by the *CaSolver* module, the *Visualizer* employs a color-based scheme that depicts the events' content similarities.



Type: SND
Pid: 123
Source: node1:5000
Destination: node2:6000
Size: 12
100% similar to events: 2

Figure 3.6. CaT's *visualizer* output example.

Fig. 3.6 shows the *Visualizer* output for the example from Fig. 3.1. Each host's events are represented as circles positioned along a dashed line according to the order in which they occurred. For instance, two events occur on host *node1*, a network *send* (❶) followed by a network *receive* request (❺). Each event is accompanied by its ID (*e.g.*, 1, 5) and type (*e.g.*, SND, RCV).

Causal relationships are represented by a line linking two events (*e.g.*, line between events ❶ and ❷). Data dependencies (*i.e.*, events whose content is similar) are colored with the same color (*e.g.*, events ❶ and ❷, and events ❸, ❹ and ❺). Events whose content is unique are assigned with the black color (ⓧ).

When selecting a specific event or relationship (*i.e.*, line) it is possible to observe additional information. Fig. 3.6 shows such information for event ❶. Namely, it states that it is a *send* request of 12 bytes from *node1* (port 5000) to *node2* (port 6000), issued by a process with PID 123. Moreover, it summarizes the event's similarities, showing that its content is 100% similar to the one from event ❷.

3.3.4 Implementation

CatStrace is implemented in Python with ≈1K LoC. For capturing syscalls information, the tracer uses Strace [109]. For parsing Strace's output, *CatStrace* extends the *strace-parser* tool [58]. For computing the hash sums of events' content, *CatStrace*'s *SigComp* submodule uses *pylsh*, a Python implementation of LSH with the MinHash algorithm [73].

CatBpf is implemented in ≈2K LoC, divided into two parts: *i)* the eBPF programs that run in kernel space (*collector* module) and *ii)* the user space code including the remaining tracer's logic (*handler* module). The eBPF programs (25 in total attached to 15 kernel probes and 10 tracepoints) are implemented in restricted C (≈900 LoC) and are responsible for collecting and filtering relevant I/O requests. The user space code is implemented in >1K LoC written in Go (v1.14) and is responsible for enabling the desired probes (*i.e.*, attaching eBPF programs to tracepoints and kernel probes), specifying the user-defined filters to apply at kernel space, gathering and parsing the information collected in the kernel, and saving it to disk. This is done using the BPF Compiler Collection (BCC) framework through the *gobpf* lib (v0), which provides an abstraction for creating, attaching, and interacting with eBPF programs. For the *SigComp* submodule we use a Go implementation of the Minhash algorithm provided by the *minhash-lsh* lib (v0).

The *CaSolver* has two implementations with ≈300 LoC, one in Python to use with *CatStrace*, and another in Go to use with *CatBpf*, that use *pylsh* and *minhash-lsh*, respectively, for computing the MinHash and LSH algorithms. The *Trace Processor* and *Visualizer* extend Falcon's original components and are implemented in ≈2K LoC in Java and 2K LoC in JavaScript, respectively. The *HB Model Generator* is identical to the one provided by Falcon and is implemented in Java with ≈800 LoC.

3.4 CaT in Action

To showcase CaT's usability, we next evaluate the new insights it can offer with its content-aware approach. To that end, we selected two widely used Big Data applications, TensorFlow [1] and Apache Hadoop [46], and analyzed (with CaT) their data access patterns and the correction/adulteration of their protocols.

Testbed Configuration. TensorFlow experiments are deployed on a server equipped with a 8-core Intel Core i9-9900K, 16 GiB of RAM, a 1 TiB Non-Volatile Memory Express (NVMe) Solid State Drive (SSD), and a NVIDIA GeForce RTX 2070 Graphics Processing Unit (GPU) with Compute Unified Device Architecture (CUDA) (v10.2). Hadoop (v2.7.1) experiments consider three DataNodes, one NameNode, and one client running on servers equipped with a 6-core Intel Core i5-9500 CPU, 16 GiB of RAM, a 500 GiB Hard Disk Drive (HDD) and a 250 GiB SSD, interconnected by a switched 10 Gigabit Ethernet network.

3.4.1 Observing TensorFlow's Dataset Shuffle Pattern

Next, we show how CaT can be used to analyze TensorFlow's training phase and observe the access pattern used to read the dataset from disk.

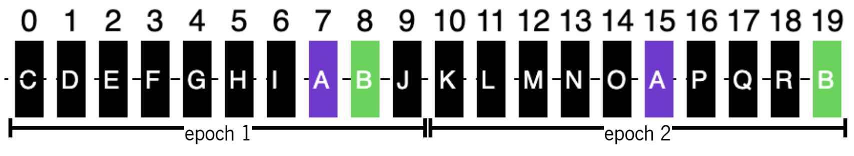
TensorFlow is an ML platform used for the training and inference of deep neural networks [1]. During the training phase, TensorFlow performs disk I/O operations to read the dataset being used to build the deep-learning model.

In this use case, CaT is used to capture TensorFlow's interactions with the storage medium while reading a sample of the ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC2012) dataset [101], often used for computer vision research, with the LeNet CNN model [69], which, due to its disk I/O-bound nature, provides a scenario where CaT's tracers must capture efficiently multiple disk operations [103].

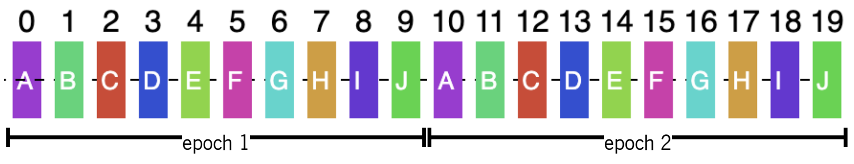
A dataset is typically split into three groups: train, validation, and test. During the training phase, TensorFlow uses the training set to train the model for a given number of times (epochs). The training set of the dataset sample used in these experiments includes 64 TFRecords with a total of 64 images[1]. Moreover, on each epoch, it is usual to randomly shuffle the data records that are going to be read to

[1]A detailed description of the original dataset is provided in §3.5

keep the model general while avoiding overfitting and decreasing its accuracy. If shuffling is disabled, all epochs will fetch (read) data records in the same order. Therefore, in these experiments, the LeNet model runs for two training epochs with shuffling enabled and disabled. *CatBpf* executes along with TensorFlow to capture its I/O events, and the resulting *CatLog* file is then provided to the remaining CaT's pipeline.



(a) Shuffle enabled



(b) Shuffle disabled

Figure 3.7. Disk access pattern for TensorFlow's dataset shuffle.

Fig. 3.7 shows the disk access pattern output of CaT's *Visualizer*. For clarity purposes, only the first ten disk read events are compared for each training epoch. Each event is represented as a rectangle. Events with the same color (and symbol) have similar content while events colored as black do not match, in terms of content, to any other depicted event.

With the shuffling mechanism enabled (Fig. 3.7a), TensorFlow accesses disk records (ImageNet images) in random order. Thus the order in which data is read differs between epochs. The only similarities found are between events 7 and 15 (A) and events 8 and 19 (B). While on the first epoch, event 7 is the eighth operation, on the second epoch, the same data is read in sixth place (event 15). The same happens for events 8 and 19. The uniqueness of data and the different order used to read the same data on the two epochs shows how, with the shuffling mechanism, TensorFlow reads the data randomly.

When the shuffling mechanism is disabled (Fig. 3.7b), TensorFlow reads the train set files in the same order (deterministic access pattern) at each epoch, as depicted by CaT's output.

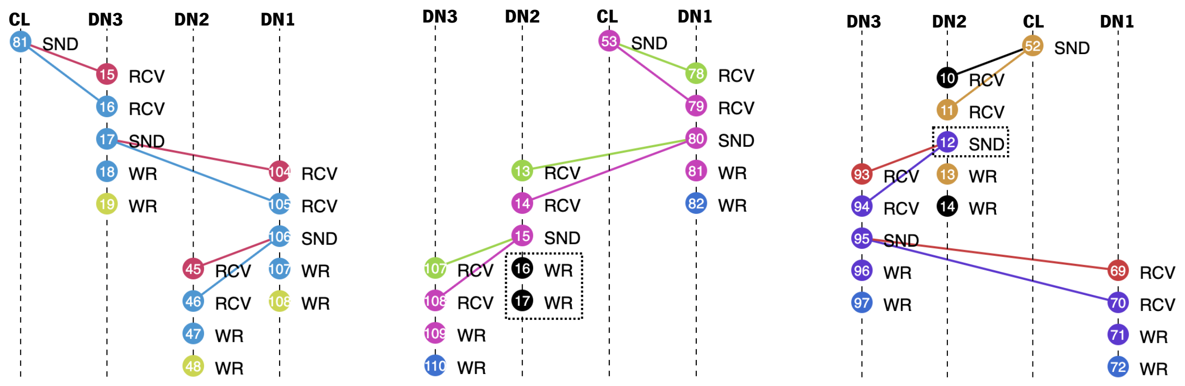
3.4.2 Verifying the HDFS File Replication Protocol

Now, we use CaT to verify the replication protocol of HDFS, the Apache Hadoop's distributed file system.

Apache Hadoop is a framework for distributed storage and processing of Big Data, which resorts to the HDFS distributed filesystem for persisting and retrieving data [46]. The latter has a master/slave architecture, with a NameNode responsible for managing metadata operations and several DataNodes where the data is actually persisted.

In this use case, CaT is used to intercept network and disk I/O calls across HDFS client, NameNode, and DataNodes. More precisely, four instances of *CatBpf* are executed, one running on the client machine to capture events issued by the HDFS `copyFromLocal` command, and the others running along with the three DataNodes. The resulting *CatLog* files are then fed into CaT's pipeline.

Briefly, the HDFS replication protocol, with a 3-factor replication, works as follows: after interacting with the NameNode, the client receives a list of available DataNodes. Then, it selects one of them to whom it will send the file. Once the elected DataNode receives the data, it sends a copy to another DataNode and persists it to disk. This process is repeated until all three DataNodes have a copy of the data.



(a) Normal execution (b) Storage corruption (c) Network corruption

Figure 3.8. HDFS replication of a file.

Fig. 3.8a depicts the visual output from CaT's pipeline. Event 81 corresponds to sending (SND) the file's content by the client (CL) to DataNode 3 (DN3). In turn, DN3 receives the data in two receive (RCV) events (15 and 16), forwards it to DN1 (17), and then saves the corresponding data (18) and metadata (19) on disk. DN1 does the same process, sending the data to DN2. Circles with the same color identify similar content. From this example, it is possible to observe the client's data path (blue color), going from the client's machine through all the DataNodes. Moreover, it shows that the three DataNodes have persisted a copy of the data (X) and the metadata (X) to disk.

In order to further prove how the similarity of events' content can add useful information about the system, we modified the source code for DN2 to observe two adultered behaviors: *i)* storage corruption, where DN2 alters the file content before persisting it on disk (Fig. 3.8b); and *ii)* network corruption, where DN2 sends the wrong data content to another DataNode (Fig. 3.8c).

For the first case (Fig. 3.8b), DataNodes 1 and 3 have a *write* event (81 and 109) with the same color as the *send* event from the client (53), indicating that their content is similar. However, the *write* events from DN2 (16 and 17) have a black color, as the data and metadata persisted is no longer equal to the one DN2 received (14), or to other data being handled by the system. As the chunk checksum verification is only performed once, upon the data arrival to the DataNode, and the data corruption happened when writing data to disk, HDFS did not reported any inconsistency.

In the second case (Fig. 3.8c), the client sends a file (52) to DN2 (11). Then, DN2 forwarded the data to DN3 (12) and persisted it to disk (13). While event 13 has the same color as events 52 and 11, event 12 has a different color, meaning that DN2 sent different content to DN3. This time, along with the chunk adulteration, we also modified its checksum (*e.g.*, mimicking a possible man-in-the-middle attack) to match the new content. Thus, DataNodes 3 and 1 were unaware of the data corruption, and both persisted wrong copies of the client's data and metadata.

3.4.3 Summary

The previous use cases showcase the advantages of combining the tracing and analysis of both the context and content of I/O network and storage requests. CaT provides a more complete strategy to analyze complex systems which can pinpoint correctness and dependability flaws that are not visible when using context-based state-of-the-art tools and are not detected by the integrity mechanisms of the applications. Even for scenarios where the data is encrypted, therefore limiting the ability to find equal data (as different ciphertexts can correspond to the same plaintext data), CaT can be used to ensure that encryption algorithms are being correctly applied. For instance, when using a probabilistic encryption scheme, the content of different events should never have high similarity degree.

3.5 Experimental Evaluation

We now focus on the evaluation of CaT's content-aware tracers for answering the following questions:

- *What is the performance, resource usage, and storage overhead of CaT's tracers at the application's critical I/O path*[2]*?*
- *How do the two different tracers vary in terms of accuracy (number of captured events)?*

To accomplish this, we compare the performance and accuracy of CatStrace and CatBpf when tracing the aforementioned Big Data applications (TensorFlow and Apache Hadoop) with I/O intensive workloads.

Setups. Experiments include three distinct setups:

- *Vanilla*: The targeted application running isolated (*i.e.*, without any tracing tool).
- *CatBpf*: The eBPF-based tracer running simultaneously with the targeted application and intercepting its events. To optimize the number of I/O events handled, CatBpf was configured to capture only the first 4 KiB of content from each request. As shown by the results, this configuration allows capturing the context and content of more events while still providing useful content-aware insights.
- *CatStrace*: The Strace-based tracer intercepting the targeted application's events, while capturing 256 KiB of requests's content, which allowed obtaining the full content buffers for most events.

Workloads. For TensorFlow experiments, we use the LeNet model [69] with the entire ImageNet dataset [101], which includes 1.28M images for training (≈138 GiB) and 50K images for validation (≈6 GiB), distributed across 1K classes. The dataset was previously converted to the TFRecord format, resulting in 1152 TFRecords files (1024 for training and 128 for validation), occupying ≈144 GiB.

[2]Note that the remainder of CaT's pipeline components run in background.

For Apache Hadoop experiments, we resort to BigDataBench (v5.0) [118], a Big Data benchmark suite that provides representative real-world datasets. BigDataBench is used with the Naive Bayes algorithm (a classification algorithm used in data mining) and the Amazon movie review dataset. The experiments also consider the loading phase of the dataset into the HDFS store with two dataset sizes (16 GiB and 32 GiB).

Collected Metrics. Besides measuring the elapsed time and throughput metrics, the Dstat [90] tool is used to observe CPU and memory usage on each cluster node. The events' statistics reported by each tracer are also collected, including *handled* (i.e., the total of events processed by each tracer), *saved* (*i.e.*, events persisted in the *CatLog* file), *incomplete* (*i.e.*, events including only context information), *truncated* (*i.e.*, events whose content was truncated to a smaller size due to their original request buffer size being greater than the captured size), and *lost* events. The experiments were conducted in the same testbed as described in §3.4 and considered two runs for TensorFlow and three runs for BigDataBench tests.

3.5.1 TensorFlow

The TensorFlow experiments consist of running the training workload (for the sample dataset) for 20 epochs with one GPU and a batch size of 64.

Table 3.1. Performance and accuracy results for TensorFlow experiments. '—' indicates that the metrics are not applicable, while '⋆' means that the values cannot be measured.

Setup	**Performance**		**Events**			
	Elapsed time (*mins*)	Images per second	Handled	Incomplete	Truncated	Lost
Vanilla	170	2,528	—	—	—	—
CatBpf	174	2,496	11,836,041	0	11,788,963	0
CatStrace	611	703	⋆	—	⋆	—

Performance Impact and Accuracy. Table 3.1 shows the elapsed training time, the number of images processed per second, and the events' statistics. As expected, the *vanilla* setup processes the highest number of images per second (≈2,528) and executes in the shortest time (170 mins).

Comparing to the *vanilla* setup, the *CatBpf* deployment decreases the images processed per second by 1.3% and increases the elapsed training time by 2.4%. *CatBpf* collects all the events and their content (*i.e.*, there were no *incomplete* or *lost* events). For the TensorFlow use case, most events correspond to read requests targeting different files of the ImageNet dataset. As each read operation has approximately 256 KiB, the captured content is truncated to the first 4 KiB, resulting in 99.6% *truncated* events. The resulting *CatLog* file, with all the collected events and corresponding context metadata and content signatures, occupies approximately 5.1 GiB.

The performance impact imposed by *CatStrace* is higher, achieving only 703 images processed per second (a decrease of 72.2% of *vanilla* throughput), with an elapsed time of 611 mins, almost 3.6× more than the *vanilla* execution time. Moreover, the Strace command invoked by *CatStrace* produced a file (*strace.out*) with 7.6 TiB of the collected information. As the generated file exceeded the disk capacity, we could not save and posteriorly analyze all the collected information (depicted as '$*$' in Table 3.1).

Resource Usage. For the TensorFlow tests, the *Vanilla* deployment used 5.6 GiB of RAM and 43% of CPU. The *CatBpf* setup increased those values up to 12.1 GiB and 54.0%, respectively. This increase is justified by the extra processing done at the critical I/O path and the size of the ring buffer and eBPF maps necessary to obtain more accurate logs. Conversely, the *CatStrace* deployment required only 4.8 GiB of RAM and 14.5% of CPU. As *CatStrace* delays I/O requests and generates less load on the system, resource utilization is also lower.

Takeaways. *Results show that CatBpf offers the best balance in terms of I/O performance, storage space usage and accuracy for this specific scenario. Although truncating the events to 4 KiB, it imposes negligible performance overhead and collects all events. On the other hand, CatStrace collects the full content of requests but imposes high performance and storage overheads.*

3.5.2 BigDataBench

The BigDataBench experiments include a loading phase (*load*), where the dataset is written to HDFS, and a running phase (*run*), where the Naive Bayes algorithm is executed.

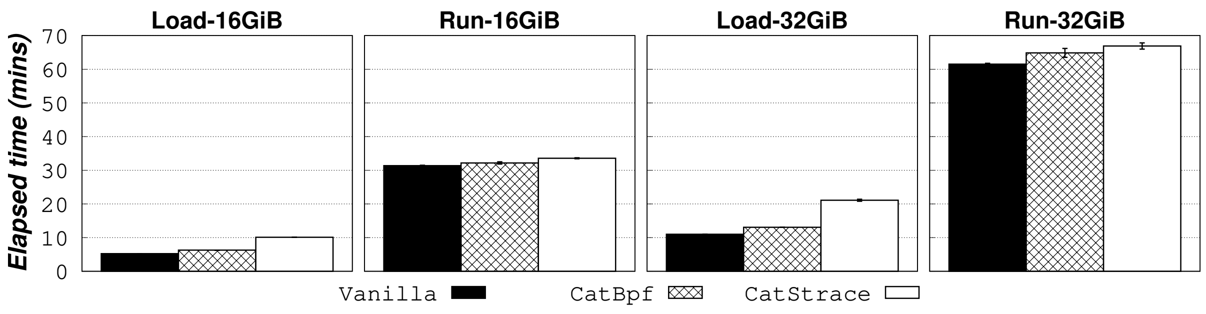


Figure 3.9. BigDataBench elapsed times.

Performance Impact. Fig. 3.9 depicts BigDataBench elapsed times for each phase and dataset size (16 GiB and 32 GiB). The elapsed times for the *vanilla* setup round the 5.2 mins for *load-16GiB* and 11 mins for *load-32GiB*. The *CatBpf* setup increases the elapsed time by almost 1.20×, taking about 6.3 and 13.1 mins for *load-16GiB* and *load-32GiB*, respectively. The *CatStrace* setup lasts around 10.1 mins for *load-16GiB* and 21.1 mins for *load-32GiB* (almost 1.93× more than the *vanilla* setup). Concerning the *run-16GiB* test, the *vanilla* setup runs in 31.4 mins, while *CatBpf* and *CatStrace* executions last for 32.2 and 33.6 mins, respectively. As for *run-32GiB*, the elapsed times are 61.5, 64.9 and 66.9 mins for *vanilla*, *CatBpf* and *CatStrace* setups, respectively.

Accuracy. The loading phase generates more I/O requests in a shorter time span when compared to the running phase, explaining why the performance impact is more significant in the former. As shown in Table 3.2, at the loading phase, *CatBpf* captures all the network and storage requests (around 8M on

Table 3.2. Accuracy results for the BigDataBench experiments. '–' indicates that the metrics are not applicable.

Events	**Load-16GiB**		**Run-16GiB**		**Load-32GiB**		**Run-32GiB**	
	CatBpf	*CatStrace*	*CatBpf*	*CatStrace*	*CatBpf*	*CatStrace*	*CatBpf*	*CatStrace*
Handled	8 M	16 M	18 M	7 M	17 M	32 M	35 M	14 M
Saved	8 M	6 M	18 M	6 M	17 M	12 M	35 M	12 M
Incomplete	0.8 M	–	16 M	–	1 M	–	33 M	–
Truncated	3 M	1	2 M	1	7 M	1	4 M	2
Lost	0	–	337	–	0	–	235	–

the *load-16GiB* test and 17M on the *load-32GiB* test), with approximately 9.18% of *incomplete* events and truncates the captured content of 42% of *handled* events. The *CatStrace* setup collects about 16M and 32M requests for the loading phases of 16 GiB and 32 GiB, respectively. *CatStrace* saves all relevant events to the *CatLog* file and only truncates 2 content buffers that are larger than 256 KiB.

As for the running phase, *CatBpf* loses up to 337 events for *run-16GiB* test and only saves the content for 89% of the 18M *handled* events. For the *run-32GiB* test, it loses 235 events and saves as *incomplete* 93% of the 35M *handled* events. The percentage of truncated content from the *handled* events is up to 13% for both dataset sizes. *CatStrace* handles around 7M of requests for *run-16GiB* and 14M for *run-32GiB*. Again, *CatStrace* saves all relevant events to the *CatLog* file and only truncates 3 of them.

Tracers are configured to only capture HDFS's data and metadata operations, while requests to third-party libraries and applications (*e.g.*, Java) are ignored. While the requests that reach the *CatBpf handler* (*handled* events) no longer include ignored operations, as these are filtered at kernel space, the same does not happen for *CatStrace* where requests are only filtered by the *handler* at user space. This explains the difference between the number of *handled* events observed for both tracers, and the difference between *handled* and *saved* events for CatStrace. Moreover, while *CatStrace* collects requests from a given PID or command and their newly created processes, *CatBpf* also captures events from running processes that were created by the target application before the tracing phase. That is why the number of *saved* events, specially for the running phases, is higher than the one from *CatStrace*.

Resource Usage. Regarding BigDataBench load experiments, the *vanilla* setup uses 3.7 GiB of RAM on the client node, 0.8 GiB on the NameNode, and 0.6 GiB on DataNodes. *CatBpf* requires additional 3 GiB, for each type of node, while *CatStrace* reduces RAM consumption by 1 GiB at the client node. For the remaining nodes, *CatStrace* uses around the same amount of RAM as the *vanilla* setup.

For the same experiments, the *vanilla* setup uses 1.5% of CPU on the NameNode, 10.3% on the DataNodes, and 98.2% on the client node. *CatBpf* increases CPU usage by 15% for the NameNode and 30% for the DataNodes. The values for the *CatStrace* setup are similar to the *vanilla* ones, except for the CPU usage on the client machine that requires 70% of CPU. As for BigDataBench run experiments, the *vanilla* setup uses 1.9 GiB of RAM on the client node, 1 GiB on the NameNode, and 2.4 GiB on the DataNodes. *CatBpf* imposes an increase of 2 GiB on each server while *CatStrace* uses 1 GiB less at the client node. CPU usage is similar for the *vanilla* and *CatStrace* setups, ≈0.4% for the client node, 1.3% for the NameNode and 46% for DataNodes. *CatBpf* increases CPU usage by 15% across all nodes.

Takeaways. *Results show that CatStrace captures all the events, truncating almost none of them, but generates significant performance overhead and a large output trace log. For example, it creates a file of 120 GiB when tracing only the network and storage events for one of the HDFS DataNodes during the running phase of 32 GiB. Once again, the CatBpf deployment shows to be the one with the best tradeoffs, if the loss of content for some events can be tolerated, namely if it is still able to provide insightful analysis information at the later phases of the pipeline. Although presenting a high percentage of incomplete events at the running phase, it captures the context of almost all the events while being the tracer that incurs the least performance overhead.*

3.5.3 Summary

The previous results show that, depending on the workload, it is possible to collect the context and content of I/O requests with negligible performance overhead.

CatBpf imposes the least performance and storage space overheads but captures only 4 KiB of each request and can lead to events' loss in scenarios with increased I/O loads. When tracing an application with lower I/O throughput (*e.g.*, TensorFlow with ≈1147 events/s), *CatBpf* can collect the content and context of all requests. When tracing a more I/O intensive application (*e.g.*, BigDataBench with ≈22,308 events/s for *load-16GiB*), *CatBpf* starts losing information (*i.e.*, presents a high percentage of *incomplete* events and a few *lost* events). Moreover, *CatBpf* can increase resource consumption (CPU and RAM) considerably. Yet, for scenarios where one wants to debug applications or trace non-CPU-intensive applications, *CatBpf* is still a good approach.

When CPU consumption is a major criterion, *CatStrace* provides a good alternative. Contrarily to *CatBpf*, *CatStrace* presents lower resources usage values and can capture all the events and their full content for any I/O throughput, but it incurs significant performance and storage space overheads. Indeed, *CatStrace* can easily create intermediate log files in the order of TiBs, while *CatBpf*, by computing hash sums before storing the corresponding logs persistently, can reduce such values to few GiBs.

One aspect to take into account is the implications of lost information at the analysis phase. Namely, when truncating the events' content thus capturing only the first X bytes of their payload, events with the same first X bytes but with a different payload for the remaining content will be matched as equal. Additionally, *incomplete* or *lost* events do not provide sufficient information to apply our similarity-based analysis. For *lost* events, the causality inference analysis is also impossible to conduct.

To sum up, if I/O performance overhead must be minimized and one can relax CPU and RAM usage and accuracy criteria, *CatBpf* is the best option. On the other hand, if all events must be captured and resource usage must be kept low, at the cost of additional I/O performance and storage space overhead, *CatStrace* should be used.

3.6 Related Work

The analysis of systems' behavior has been a subject of extensive research for diverse purposes such as troubleshooting, debugging, performance analysis, and anomaly detection.

Applications' Logs Analysis. A common approach is to use static analysis or ML algorithms to extract information from application logs [36, 124, 128, 131]. However, the typical information available at these logs makes it hard, if not impossible, to correlate events across heterogeneous and distributed components.

Instrumentation-based Tracing. Another approach is to trace applications' events by instrumenting their source code or binaries. These solutions modify applications or middleware libraries to collect the necessary information or propagate context across the different components of a distributed system [24, 45, 74, 106, 115]. However, this approach requires prior knowledge and access to the source code of targeted systems, thus making it less transparent and less applicable to a wider range of scenarios.

Non-intrusive Tracing. Non-intrusive approaches resort to kernel-level tracing tools (*e.g.*, Strace, LSM, eBPF) to capture applications requests [82, 84, 89, 98, 110]. Although some of them can infer the data flow across multiple nodes by correlating network events with file operations, their analysis is focused solely on the requests' context, thus overlooking possible data corruption scenarios (such as the example from Fig. 3.1) or content flows such as those depicted for HDFS in Fig. 3.8. These can only be revealed when observing the content of requests. Unixdump[38] and Re-Animator[5] are the only non-intrusive solutions that can capture the content of I/O events. However, none of these solutions can capture network and storage I/O requests simultaneously, while being restricted to request tracing, thus not providing any analysis and visualization mechanisms.

Unlike previous solutions, CaT is able to capture the context and content of both network and storage events. Also, it can track the causality of events across a distributed system deployment. Finally, CaT contemplates a complete content-aware pipeline including the non-intrusive tracing, correlation, analysis, and visualization of distributed I/O events.

3.7 Summary and Discussion

This chapter introduces CaT, a novel framework for collecting and analyzing storage and network I/O requests of distributed systems. The key contribution is a content-aware tracing and analysis strategy that correlates the context and content of events to better understand the data flow of systems.

A detailed evaluation of CaT's open-source prototype with real applications shows that CaT's content-aware approach can improve the analysis of distributed systems by pinpointing their data flows and I/O access patterns. These improvements are key to finding correctness, dependability and performance issues in today's complex systems.

Moreover, experimental results show that depending on the target workload, it is possible to capture most of the I/O events while incurring negligible performance overhead. When choosing the appropriate set of tracing tools (§3.5), we show that CaT can be used over real systems while imposing a balanced tradeoff in terms of accuracy, performance overhead and resources usage. At the proposed framework, only the tracers are deployed on the critical network and storage I/O path of applications, while the remainder of CaT's pipeline can be executed in background and on dedicated servers.

CaT is designed with the main goal of diagnosing how I/O requests' flow in distributed systems. Therefore, the data collection, analysis and visualization strategies followed by our solution are tightly coupled to such objective. In the next chapter we show that data-centric applications, even non-distributed ones, can also benefit from multi-purpose diagnosis frameworks. However, these frameworks must address different requirements (*e.g.*, in terms of performance, resource usage and accuracy) and features (*e.g.*, comprehensive, flexible and near-real-time diagnosis), which are not supported by CaT.

4

Practical and Timely Diagnosis of Applications' I/O Behavior

The correctness, dependability, performance of data-centric applications (*e.g.*, databases, KVSs, analytical engines, ML frameworks) is highly influenced by the way these interact with in-kernel POSIX storage systems, such as file systems and block devices [49, 100]. However, the sheer amount of storage operations generated by these applications, ranging from hundreds to thousands of operations per second, makes the analysis of such interaction a complex and time-consuming task when done manually.

Diagnosis tools can automate this task for users and developers, while aiding in error debugging, finding performance and dependability issues, and identifying potential I/O optimizations for applications [32, 102]. Indeed, the main insight of this chapter is that, by combining syscall tracing with a customizable analysis pipeline, one can achieve non-intrusive and comprehensive I/O diagnosis for applications using in-kernel POSIX storage systems. However, doing so requires overcoming multiple challenges, as discussed in §2.4:

Challenge C1. Instrumenting applications' source code, when available, is still undesirable as it requires users to manually analyze and instrument distinct and potentially large codebases (*e.g.*, RocksDB has approximately 440K LoC written in six different programming languages).

Challenges C4, C5 and C6. Given the large amount and burstiness of storage requests issued by data-intensive applications, one must use tracing solutions that minimize the performance impact and resource usage, while still capturing all the information that is relevant to accurately diagnose the targeted application.

Challenges A1 and V1. The analysis and visualization of collected traces must be automated, given the large number of I/O events (easily reaching tens of millions) that must be parsed, correlated, and visually represented to provide insightful information.

Challenges C3, C7, A2, V2 and V3. Current solutions offering a complete pipeline for application diagnosis, including CaT, are designed for rigid analysis scenarios [64, 98, 102, 126]. Ideally, diagnosis tools should provide the flexibility to narrow or broaden tracing, analysis and visualization scopes based on user goals. Avoiding this limitation would enable exploring a wider range of correctness, dependability and performance issues that applications may exhibit, such as those identified at §4.4.

Challenges A3. Postmortem (offline) analysis requires users to wait for all I/O events to be collected before starting the analysis of the targeted application. For data-intensive workloads this may lead to long waiting periods (*e.g.*, hours). Ideally, users should be able to observe I/O requests as these are collected (*i.e.*, in near real-time) to speed up their analysis tasks.

This chapter proposes DIO, a generic tool for observing and diagnosing applications' storage I/O. It addresses the aforementioned challenges with the following contributions:

Non-intrusive, Comprehensive and Flexible Tracing. DIO offers a new eBPF-based tracer that intercepts syscalls issued by applications without requiring changes to their source code or instrumentation of binaries. By operating at kernel space, DIO is able to intercept syscalls submitted by any application that makes POSIX requests to the storage system. The tracer supports 42 storage-related syscalls and records a comprehensive set of information for each operation, including its type, arguments, return value, timestamp, PID, and TID. By offering a flexible design, DIO allows collecting only events of interest, filtering them (at kernel-level) by syscall type, PID, TID, or file paths. This enables narrowing the tracing scope according to users' requirements, reducing the size of the stored trace, and minimizing performance overhead over the targeted application.

Enriched Analysis. DIO enriches data gathered for each syscall with additional context available at the kernel (*e.g.,* process name, file type, offset), which can be used to improve the correlation and analysis of requests (*e.g.*, associating different syscalls to a file path, differentiating operations over regular files or directories). These features enable a richer and wider analysis of incorrect or inefficient I/O patterns.

Asynchronous Event Handling. Only syscall interception is done synchronously, while traced events are collected and processed in user space asynchronously. This avoids adding extra latency in the critical path of I/O requests and enables practical analysis of data-intensive workloads.

Near Real-time Pipeline. DIO offers a practical and customizable pipeline so that users can create their queries, correlation algorithms, and dashboards to analyze collected data. It follows an inline approach, meaning that traced events are automatically parsed and forwarded to the analysis and visualization components as soon as they are collected in user space, without requiring manual user intervention.

DIO is implemented as an open-source prototype using eBPF [76], Elasticsearch [8], and Kibana [9], and validated with production-level systems. Results show that DIO enables the diagnosis of *i)* inefficient use of syscalls that lead to poor storage performance in Redis, *ii)* unexpected file access patterns caused by the usage of high-level libraries that lead to redundant I/O calls in Elasticsearch, *iii)* erroneous file accesses that cause data loss in Fluent Bit, and *iv)* resource contention in multithreaded I/O that leads to high tail latency for user workloads in RocksDB.

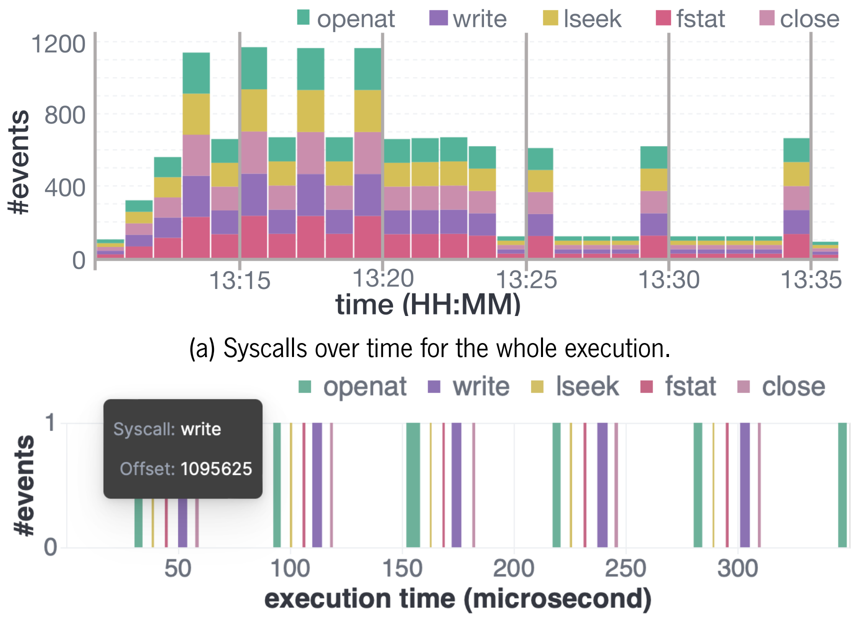
Moreover, we conduct a thorough experimental evaluation that highlights the different tradeoffs in terms of performance overhead, resource usage, and tracing accuracy when using different tracing modes and configurations provided by DIO while validating our solution against two state-of-the-art syscall-based tracers: Strace [109] and Sysdig [111]. Results show that when compared with an inline diagnosis pipeline using Sysdig, DIO provides timely analysis for users and improves the amount of captured events by up to 28× while keeping performance overhead between 14% and 51%.

All artifacts discussed in this chapter, including DIO, workloads, scripts, and the corresponding analysis and visualization outputs, are publicly available at `https://github.com/dsrhaslab/dio`.

4.1 Motivation

To showcase the benefits that integrated syscall tracing, analysis, and visualization bring towards validating inefficient I/O behavior from applications, let us consider a previously known issue identified in the Redis in-memory data store [97]. Specifically, the server log file is repeatedly opened and closed for every written line, which adds extra latency for log operations and can potentially slow down Redis performance.[1]

To identify this behavior, users could run a workload on top of Redis and trace the syscalls submitted to kernel. In this example, we used the *redis-benchmark* to generate 5M requests to the database, which yield >200M syscalls.[2] Inspecting these events without proper filtering, correlation, and visualization mechanisms is a non-trivial and time-consuming task.



(a) Syscalls over time for the whole execution.

(b) Sample of first 350μs of a millisecond.

Figure 4.1. Log file access pattern, depicting syscalls issued within second and microsecond resolutions, for Redis's version including inefficient I/O patterns (commit #e9ae037).

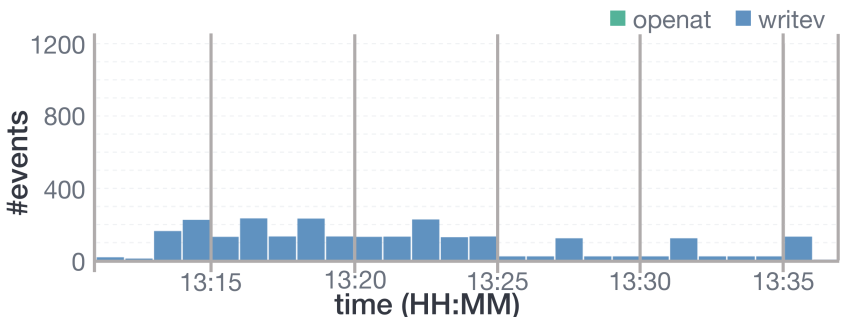
In this chapter, we argue that an analysis pipeline integrating the previous mechanisms would greatly simplify users' work. In particular, for this specific use case, by intercepting only the syscalls submitted

[1] *Logging improvements* issue from Redis' GitHub repository: `https://github.com/redis/redis/pull/10531`

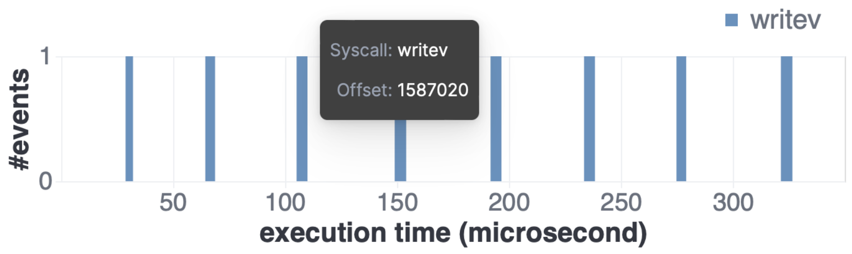
[2] Redis benchmark: `https://redis.io/docs/management/optimization/benchmarks/`

to the file system, while discarding Redis's `read` and `write` operations for network sockets, one would just need to trace ≈600K storage-related syscalls (*i.e.,* ≈0.3% of the original tracing sample).

Then, through correlation, users could further filter these storage events and explore only syscalls directed into the log file. Finally, through visualization, it would be possible to observe the pattern shown in Figs. 4.1a and 4.1b. The former shows a set of syscalls being made repeatedly over the log file. The latter depicts a sample of the first 350μs within a millisecond, showing the exact order and duration of requests for one of these sets (*i.e.*, `openat`→`lseek`→`fstat`→`write`→`close`). By interacting with the latter visualization (*i.e.*, visually exploring the syscalls' arguments), it would be possible to observe that those syscalls are accessing consecutive file offsets, suggesting a sequential file access pattern.



(a) Syscalls over time for the whole execution.



(b) Sample of first 350μs of a millisecond.

Figure 4.2. Log file access pattern, depicting syscalls issued within second and microsecond resolutions, for Redis's version including the corrections (commit #d4c8dff).

As suggested in the pull request for the issue, this inefficient I/O pattern can be corrected by: *i)* keeping the log's file descriptor opened while the file is being used, and *ii)* using `writev` to write log lines more efficiently. As depicted in Figs. 4.2a and 4.2b, by using the same analysis pipeline, users can validate the suggested correction, where redundant `open` and `close` operations are avoided, along with the need for using `lseek` before every write operation. Also, `writev` is now used to write log lines instead of `write`.

The aforementioned visualizations are real outputs of using DIO for this use case (available at `https://dio-tool.netlify.app/use-cases/redis`). Next, we describe the proposed solution, and in §4.4 we show that it can be used to discover other types of undesired I/O behaviors, observe erroneous file access patterns that cause data loss, and assist with the root cause analysis of applications exhibiting high tail latencies.

4.2 DIO in a Nutshell

DIO is a generic tool for observing and diagnosing the I/O interactions between applications and in-kernel POSIX storage systems. Its design is built over the following core principles.

Transparency and Reduced Overhead. DIO relies on the Linux kernel tracing infrastructure (i.e., tracepoints, kernel probes) to intercept applications' syscalls without modifying their source code or underlying libraries. Moreover, DIO uses tracing technologies that allow minimizing the extra processing done in the critical path of I/O requests to reduce the performance overhead imposed on targeted applications.

Practical and Timely Analysis. Traced data is asynchronously sent to a remote analysis pipeline, avoiding adding extra latency on the critical I/O path of applications while still enabling users to visualize collected data in near real-time.

Postmortem Analysis. DIO allows storing different tracing executions from the same or different applications and posteriorly analyzing and comparing them.

Flexible and Comprehensive Tracing. DIO intercepts different types of storage-related syscalls, covering data (*e.g.*, `write`, `read`), metadata (*e.g.*, `openat`, `stat`), extended attributes (*e.g.*, `getxattr`, `setxattr`), and directory management (*e.g.*, `mknod`, `mknodat`) requests. Users can choose to capture only the relevant ones for their analysis goals and further filter them based on targeted PIDs, TIDs, and file paths. Moreover, two tracing modes (described in §4.3.1) are provided and allow users to configure the amount of detail collected for each I/O syscall. These tracing modes and filters allow minimizing the performance impact and storage overhead (i.e., the size of traced data) imposed by DIO.

Enriching Syscall Analysis. DIO enriches the information provided directly by each syscall (*i.e.*, type, arguments, return value) with additional context from the kernel, such as the name of the process that originated the request, the type of the file being accessed by it, and its offset.

Data Querying and Correlation. With DIO, users can query traced data, apply filters to analyze specific information (*e.g.*, syscalls executed by a specific TID), and correlate different types of data (*e.g.*, associate file descriptors with file paths).

Customized Visualization. DIO comprises a visualization component that provides mechanisms for simplifying data exploration and building customized visualizations.

4.2.1 System Overview

DIO consists of three main components, namely the *Tracer*, the *Backend*, and the *Visualizer*, as depicted in Fig. 4.3. The *Tracer* intercepts syscalls from applications, filters them according to users' configurations (*e.g.*, by TID), and packs their information into events that are asynchronously sent to the *Backend* (❹). The latter persists and indexes events (❺) and allows users to query and summarize (*e.g.*, aggregating) stored information (❻). Meanwhile, the *Visualizer* provides near-real-time visualization of the traced events by directly querying the *Backend* (❼). Users rely on the *Visualizer* to ease the process of data exploration

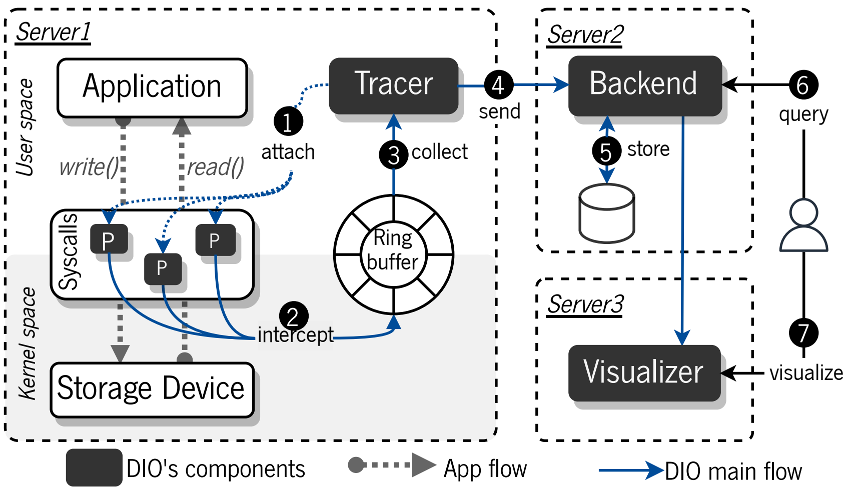


Figure 4.3. DIO's design and flow of events.

and analysis by selecting specific types of data (*e.g.*, syscall types, arguments) to build different and customized representations.

4.2.2 Architectural Components

Next, we detail each of DIO's architectural components and their functionalities.

Tracer. The *Tracer* intercepts syscalls done by applications in a non-intrusive way. To that end, it relies on the eBPF technology [76] to instrument the Linux kernel by executing small programs (*i.e.*, eBPF programs) whenever a given point of interest (*e.g.*, tracepoint, kernel probe) is called.

In detail, DIO's *Tracer* comprises a set of eBPF programs that, at the initialization phase (❶), are automatically and transparently attached to syscall tracepoints. Whenever these tracepoints are reached (*i.e.*, a syscall is invoked), the eBPF program gathers the desired information about the request, including *entry* (*e.g.*, arguments) and *exit* (*e.g.*, return value) related data, and places it in a per-CPU *ring buffer* (❷), which is a contiguous memory area used for exchanging data between kernel (producers) and user space (consumers) processes. At user space, the *Tracer* asynchronously fetches information from the *ring buffer* (❸), parses it into events (specified in JSON objects), and sends these to the *Backend*. To minimize both network and performance overhead, the *Tracer* groups several events into buckets that are sent and indexed in batches at the *Backend*.

Table 4.1 depicts the syscalls supported by DIO. Since instrumenting syscalls can introduce extra processing in the critical path of I/O requests, DIO allows users to filter requests by:

(*a*) *type of syscall* - activates only the tracepoints for the provided syscall types.
(*b*) *process name* - captures only syscalls issued by a process with the provided name.
(*c*) *process or thread identifiers* - captures only syscalls issued by a given list of PIDs or TIDs.
(*d*) *file or directory path(s)* - captures only syscalls targeting one of the provided file or directory paths.

The flexibility offered by these filters allows users to better configure the *Tracer* according to their goals and balance the tracing accuracy with the storage and performance overheads. Namely, by specifying the targeted syscall types *(a)*, the *Tracer* avoids activating unnecessary tracepoints, thus reducing the amount of I/O requests with extra processing in their critical path. Moreover, by applying the remaining filters (namely, *(b)*,*(c)*, and *(d)*) in kernel space, DIO reduces the amount of information to be sent and processed in user space.

Backend. The *Backend* allows persisting, searching, and analyzing data from traced events. It uses the Elasticsearch [8] distributed engine for storing and processing large volumes of data. Its flexible document-oriented schema allows indexing events as documents, even if these have potentially different structures (*e.g.*, distinct fields corresponding to syscall arguments). Moreover, it provides an interface for searching, querying, and updating documents, which allows users to develop and integrate customized data correlation algorithms.

Visualizer. The *visualizer* provides an automated approach towards exploring (*e.g.*, query and filter events) and visually depicting (*e.g.*, through tables, histograms, time-series graphs) the analysis findings. This component uses Kibana [9], the data visualization dashboard software for Elasticsearch, which is often used for log and time-series analytics and application monitoring. Kibana also allows building custom visualizations, thus being aligned with the design principles of DIO.

Table 4.1. System calls supported by DIO.

Type	Syscall
Data	`read, pread64, readv, write, pwrite64, writev, fsync, fdatasync, readahead`
Metadata	`creat, open, openat, close, lseek, truncate, ftruncate, rename, renameat, renameat2, unlink, unlinkat, readlink, readlinkat, stat, lstat, fstat, fstatfs, fstatat`
Extended	`getxattr, lgetxattr, fgetxattr, setxattr, lsetxattr, fsetxattr, listxattr, llistxattr, flistxattr, removexattr, lremovexattr, fremovexattr`
Directory	`mknod, mknodat`

4.3 Algorithms and Prototype

Next, we detail the information collected, preprocessed and enriched by DIO (§4.3.1), along with new algorithms (§4.3.2) and visualizations (§4.3.3) that can aid in the interpretation of such data. Also, we discuss the implementation (§4.3.4) and usage details (§4.3.5) of our prototype.

4.3.1 Collected information

For each intercepted syscall, DIO collects information related to the: *i) syscall* - type, arguments, and return value; *ii) process* - PID, TID, and process name; *iii) time* - *entry* and *exit* timestamps. Since the amount of captured information can influence the performance overhead imposed on the targeted application, DIO offers different tracing modes: *raw* and *detailed*.

Raw. The less detailed mode (referred to as *raw*) captures the aforementioned information without preprocessing it. This means that for arguments referring to memory regions (*e.g.*, `char *pathname` in `stat` syscall, or `char *name` in `getxattr` syscall), only their hexadecimal value is saved (*e.g.*, `"name": "0x55555556feab"`). Moreover, for numerical arguments (*e.g.*, `int flags` in `openat` syscall, or `int fd` in `write` syscall) no translation is done and their values are saved in their original form (*e.g.*, `"flags": 1089`). By saving information in its raw format, DIO reduces the extra processing in the critical path of I/O requests, the amount of data transferred to user space, and the information that must be analyzed posteriorly. As shown n §4.4.3, there are scenarios where the information provided by this mode is sufficient for diagnosing I/O issues.

Detailed. For a more in-depth analysis, DIO offers a *detailed* mode, which provides comprehensive information about the requests by preprocessing some arguments before saving the events (§4.3.1.1), enriching the traced information with context from the kernel (§4.3.1.2) and translating file descriptors to their corresponding file paths (§4.3.1.3).

4.3.1.1 Data Preprocessing

Instead of keeping collected information in its raw format, the *detailed* mode transforms values into a human-readable format (*e.g.*, `"flags": "O_WRONLY|O_CREAT|O_APPEND"`), simplifying the analysis done by users. Further, this mode captures the actual memory content for pointer arguments instead of saving their hexadecimal values (*e.g.*, `"name": "system.posix_acl_access"`). For buffers being handled by data-related syscalls (*e.g.*, `void *buf` in `read` syscall), the *detailed* mode provides the option to compute a hash sum of their content (*e.g.*, `"buf": "This is the first log line"` → `"signature": 114a83d4`). This way, DIO compacts the amount of information that reaches the analysis pipeline, minimizing the storage overhead while still allowing the observation of syscalls handling the same data content (as shown in §4.4.2). The hash sum can either be computed in user space, which requires transferring buffers' content from kernel, or in kernel, which reduces the amount of information sent to user space but adds extra processing to the critical path of I/O requests.

4.3.1.2 Enriched Information

While the previous information already provides valuable insights about applications' I/O behavior, correlating this data with other types of information further enriches and eases the analysis made by users

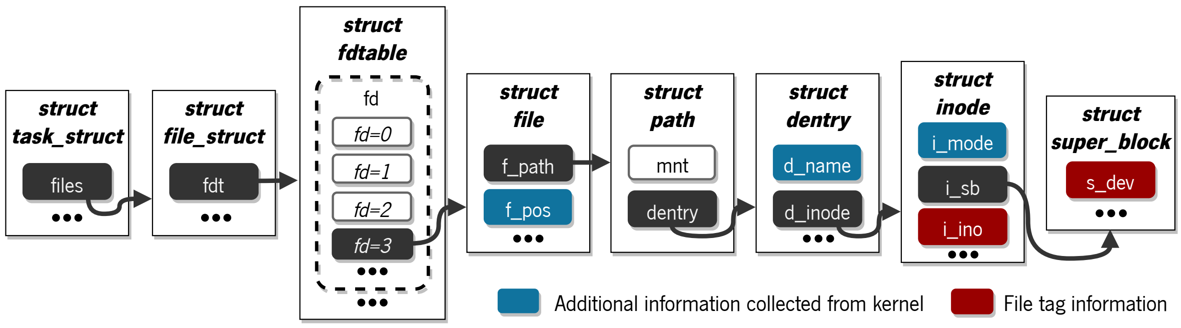


Figure 4.4. Kernel structures used by DIO. *d_name*, *i_mode*, and *f_pos* are used to obtain the file path, offset, and type. *i_ino* and *s_dev* are used to create a unique file tag.

(as discussed in §4.4). Therefore, the *detailed* mode leverages eBPF's access to kernel structures (as depicted in Fig. 4.4) and complements traced information with:

- The *file path* being accessed by syscalls. Since many syscalls access files through a file descriptor (*e.g.*, `read`, `close`, `fgetxattr`), obtaining the corresponding file path provides more specific information about the file being handled.
- The *file type* targeted by syscalls. This additional information allows differentiating accesses to regular files, directories, sockets, block/char devices, pipes, symbolic links, and others.
- The *file offset* being accessed by data-related syscalls. Information about offsets allows observing file access patterns (*e.g.*, sequential/random accesses), even for syscalls that do not provide the file offset as an argument (*e.g.*, `read`).

4.3.1.3 File Descriptor Translation

Associating syscalls with their corresponding file paths is fundamental to enable detailed tracing information. However, this is not a trivial task. The typical approach to address this challenge is to correlate the file descriptor with the file path argument of the previous `open` call that initialized it (*i.e.*, the `open` syscall that originated the file descriptor). However, this approach is not accurate as there are other mechanisms to obtain a file descriptor (*e.g.*, the creation of new processes via `fork`, duplication of file descriptors through `dup`, `dup2`, or `fcntl`). CaT (Chapter 3) accesses kernel structures to find the file path corresponding to a specific file descriptor. However, transferring file paths from kernel to user space for each event accessing a file induces significant tracing overhead and leads to potential loss of traced information.

DIO introduces a different approach by creating a custom event (*EventPath*) that contains information about a specific file (*i.e.*, file path, file type) and by sending it only once to user space. Each *EventPath* is labeled with a unique *file tag*, which is then associated with any syscall accessing that specific file. Alg. 4.1 further details how DIO creates both the *file tag* and *EventPath* event. First, whenever an intercepted syscall accesses a file through a file descriptor, DIO goes through Linux kernel structures (as depicted in Fig. 4.4) and obtains information about the file's inode number and the file system's device number (L2-L3). DIO relies on the assumption that every inode has a unique number inside the same file system. Thus, by combining the inode number with the device number, DIO creates a unique file tag (L4).

After generating the *file tag*, DIO verifies if it is included in the list of opened inodes (*i.e.,* inodes accessed during the tracing execution). If the list does not contain the current *file tag*, DIO calculates the *file path* for the current file descriptor (L6) and sends to user space a new *EventPath*, containing the file path, type, and tag (L7). The *file tag* is then added to the list of opened inodes (L8) and associated with the current event being handled (L9). If the *file tag* already exists in the list of opened inodes, it only needs to associate the *file tag* to the current event being handled (L11-L12). Finally, when an inode is destroyed, the corresponding *file tag* is removed from the list of opened inodes.

Algorithm 4.1: DIO's EventPath and file tag generation algorithm.

```
   Input:
       $fd$: file descriptor
       $curTimestamp$: current timestamp
       $openedInodes$: list of currently opened inodes
       $curEvent$: current event structure
1  Function checkInode($fd$, $curTimestamp$)
2      $inodeNo \leftarrow$ getInodeNo($fd$)
3      $deviceNo \leftarrow$ getDeviceNo($fd$)
4      $fileTag \leftarrow$ createFileTag($inodeNo$, $deviceNo$)
5      if $fileTag$ not in $openedInodes$ then
6          $filePath \leftarrow$ getFilePath($fd$)
7          submitNewEventPath($filePath$, $fileTag$)
8          $openedInodes$.append($fileTag$, $curTimestamp$)
9          $curEvent$.add($fileTag$, $curTimestamp$)
10     else
11         $timestamp \leftarrow$ getTimestamp($fileTag$)
12         $curEvent$.add($fileTag$, $timestamp$)
```

Since inodes are recycled over time, DIO distinguishes a reused inode by assigning to each file tag the timestamp of the first captured access to that inode, while in user space each event has an associated tag composed of the device number, inode number, and timestamp (*e.g.*, `"file_tag": "7340032|12|6707719730779287"`).

This approach minimizes the amount of redundant data transferred between kernel and user space and, consequently, the storage and performance overheads. Moreover, even if an *EventPath* is lost, which prevents the association of the file path to the event, a file access pattern analysis is still possible by using the *file tag*. In the next section, we discuss how each event is correlated to the corresponding file path through the generated *file tags* and *EventPaths*.

4.3.2 File Path Correlation Algorithm

We have implemented a custom algorithm to enable the correlation of syscalls with specific accessed file paths. Using Elasticsearch's data querying and updating features, the *file tags* (*i.e.*, unique identifiers generated by the *Tracer* component) associated with syscalls are translated into the actual file paths being accessed at the *Backend* (*e.g.*, `/tmp/app/log.txt`).

Algorithm 4.2: DIO's file path correlation algorithm.

```
Input:
    $sysEvents$: list of events with a file tag
    $evtPaths$: list of file paths
1 Function correlateFP($sysEvents$, $evtPaths$)
2     for $sys \leftarrow sysEvents$ do
3         for $path \leftarrow evtPaths$ do
4             if $sys.FileTag = path.FileTag$ then
5                 $sys$.FilePath ← $path$.FilePath
6                 $sys$.FileType ← $path$.FileType
```

Alg. 4.2 shows the file path correlation performed by the *Backend*. The algorithm receives two lists as arguments: *i)* the syscalls events (*sysEvents*), and *ii)* all *EventPath* events (*evtPaths*) generated during the Tracer execution. By relying on the unique *file tags*, the algorithm matches each syscall event with the corresponding *EventPath* (L2-L4), updating the former with the file path and type information (L5-L6).

4.3.3 Nanosecond Visualization

The minimum time resolution supported by Kibana for visualization is restricted to the millisecond time unit. This prevents users from observing the order and time spread for requests occurring in sub-millisecond time windows, which occur frequently when using modern storage and network hardware (*e.g.,* NVMe, persistent memory, InfiniBand). Namely, data-intensive applications can generate several thousands of I/O operations per second. Consequently, many of these operations can occur within the same millisecond. As shown in §4.1 and 4.4.1, observing the order and time spreading of requests at a smaller time interval (*i.e.*, microsecond or nanosecond) is important to diagnose applications' I/O by allowing, for instance, to visualize duplicate syscalls or to understand the sequence and duration of syscalls made in such a short time window. Therefore, we designed a new representation that depicts I/O events order and time spacing at the nanosecond time unit resolution (Figs. 4.1b, 4.2b and 4.5b). It is fully integrated with Kibana's dashboards and automatically queries the *Backend* to collect the required data.

4.3.4 Implementation

The *Tracer* is implemented in ≈8K LoC. The kernel space code responsible for collecting and filtering I/O events is implemented in 25 eBPF programs written in restricted C (≈800 LoC for the *raw* tracing mode and

≈2K LoC for the *detailed* mode) attached to 1 kernel probe and 86 tracepoints in total (*i.e.*, including *entry* and *exit* points). The user space code responsible for the remaining tracer's logic is implemented in ≈6K LoC written in Go (v1.17) and uses the *gobpf* lib (v0.2.0) for interacting with the eBPF programs, and the *go-elasticsearch* (v7.13.1) module for communication with the *Backend*, taking advantage of its bulk indexing API for sending multiple events simultaneously. The *Backend* and *Visualizer* use Elasticsearch (v8.5.2) and Kibana (v8.5.2), respectively. The file path correlation algorithm can be automatically executed by the *Tracer* or on-demand by users. The nanosecond representation is implemented with the Vega-lite visualization grammar and provided along with DIO's predefined dashboards.

4.3.5 Configuration and Usage

The installation and configuration of DIO are performed in two phases: *i)* the setup and initialization of the analysis pipeline and *ii)* the configuration and execution of the *Tracer*.

Analysis Pipeline. Although all DIO's components can be deployed in the same server, to avoid negatively impacting the performance of the targeted application (*e.g.*, additional resource consumption), the analysis pipeline can be installed on separate servers (Fig. 4.3). Further, as the *Tracer* component labels each tracing execution with a unique session name, one can deploy DIO as a service, setting up the analysis pipeline on dedicated servers and allowing multiple executions of DIO's *Tracer* on different machines and by distinct users. The deployment and configuration of the analysis pipeline comprise its software installation (*i.e.*, Elasticsearch and Kibana) and importing its predefined dashboards. As soon as traced data arrives at the pipeline, users can access Kibana's web page and visualize DIO's dashboards, apply analysis filters, and edit or create new visualizations and dashboards.

Tracer. Once the analysis pipeline is deployed, users can use DIO's *Tracer* to collect information. The *Tracer* executes along with the targeted application, stopping once its main and child processes finish or upon explicit users' instruction.[3] By default, DIO's *Tracer* enables the tracepoints for the full set of supported syscalls. However, users can specify a list of syscalls to observe, and the *Tracer* will only activate the tracepoints for those operations. Also, users may specify a list of files/directories to observe, instructing the *Tracer* to only record events that target them. Moreover, users can choose between *raw* or *detailed* tracing modes and further configure the latter to deactivate the collection of arguments that require transferring large amounts of data to user space. Namely, data buffers' content and file paths obtained from syscall arguments may only be relevant to some specific cases and, therefore, can be collected/ignored according to the analysis goals. As we show in the next section, data buffers are irrelevant for use cases §4.1, §4.4.1 and §4.4.3, while the file paths obtained from syscall such as `lstat` or `unlink` are only relevant for use cases §4.4.1 and §4.4.2. All these configurations, along with the analysis pipeline's parameters (*e.g.*, Elasticsearch URL), can be set through a configuration file.

[3]Multiple instances of DIO's tracer can be deployed to diagnose distributed applications across the servers where their components are running. Each instance of DIO's tracer will generate an independent tracing index at the same *Backend* (containing information about the targeted host), allowing for later analysis and correlation of each tracing execution.

4.4 DIO in Action

Our evaluation showcases how DIO eases the process of observing and validating known issues or exploring unknown applications and finding potential problems. To this end, besides the Redis use case discussed at §4.1, we analyzed three additional production-level applications: Elasticsearch, RocksDB, and Fluent Bit. Results show that DIO:

- provides valuable information about applications' I/O requests that can be used to uncover or confirm inefficient (§4.1) or unexpected (§4.4.1) I/O patterns;
- is a practical tool for validating the root causes of correctness (§4.4.2) and performance (§4.4.3) issues, without instrumenting large codebases.

With the exception of Fig. 4.10, all the remaining figures in this section were generated with DIO (with minimal modifications for readability). The full set of DIO's visual representations is available at `https://dio-tool.netlify.app`.

Experimental Setup. Our testbed comprises three servers running *Ubuntu 20.04 LTS* with kernel *5.4.0*. The server running the application and DIO's *tracer* is equipped with a 4-core Intel Core i3-7100, 16 GiB of RAM, a 250 GiB NVMe SSD (used for storing traced data), and a 512 GiB SATA SSD (used for hosting the datasets). DIO's *Backend* and *Visualization* components run on two separate servers, both equipped with a 6-core Intel i5-9500, 16 GiB of RAM, and a 250 GiB NVMe SSD.

Workloads. Both benchmarks and custom workloads were selected to reproduce specific but realistic interactions between the targeted applications and underlying storage systems. As shown next, these workloads validate that DIO can be used to explore the I/O patterns of real applications and identify the root cause of real known issues. Further details of the workloads and benchmark configurations are provided along with each use case.

4.4.1 Top-Down Exploration and Diagnosis of Elasticsearch

Next, we show how DIO can be used to explore and obtain additional insight into the I/O behavior of applications and then, by following a top-down approach, how one can use our solution to diagnose inefficient file access patterns.

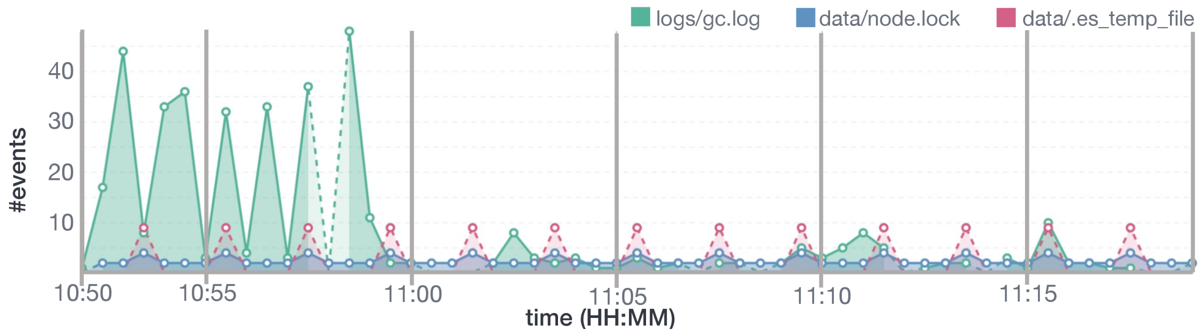
We chose Elasticsearch (v8.3.0), a distributed search and analytics engine, as the targeted application [8].[4] Due to the use case's exploratory nature, DIO was configured to capture all supported syscalls. We used the *Rally*benchmark with the default workload (*geonames*) to generate load for Elasticsearch.[5] This workload indexes ≈11M documents and executes different queries (*e.g.*, filter, sort).

Under this workload, and with the help of the information collected and organized by DIO, we observed that Elasticsearch generates >1M storage-related syscalls, 99.7% of them targeting regular files and the

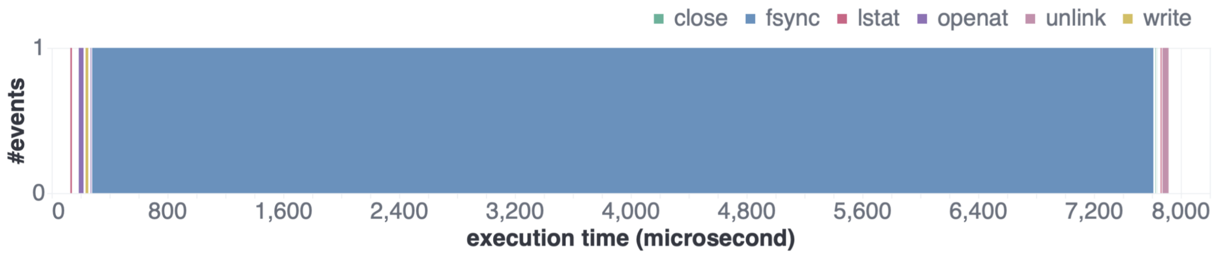
[4]Note that in this section Elasticsearch is used as the targeted application and should not be confused with the one used to implement DIO's *Backend*.

[5]Rally benchmark: `https://esrally.readthedocs.io/en/stable/install.html`

remaining ones targeting directories. Elasticsearch uses mainly data-related operations (88%), most of them being `write` (71%), `pread64` (7%) and `read` (5%). Further, it spawns a total of 42 processes and 118 threads while accessing almost 4000 files.



(a) Sample of Elasticsearch's file access pattern.



(b) Sequence of syscalls issued by Elasticsearch for file `.es_temp_file` (nanosecond visualization).

Figure 4.5. Elasticsearch's file access pattern.

As depicted in Fig. 4.5a, some files exhibit a constant access pattern, even in the absence of client requests. Namely, every 30s, Elasticsearch submits 2 syscalls to the `node.lock` file (🟦 line), and every 2 mins, 9 syscalls to `.es_temp_file` (🟥 line). For the latter, DIO's nanosecond visualization (Fig. 4.5b) uncovered an unexpected duplication of `openat` (🟪) and `close` (🟩) syscalls.

Listing 4.1. System calls made by Elasticsearch to the `.es_temp_file` file and observable with DIO.

```
❶ openat(".es_temp_file", "O_WRONLY|O_CREAT", ...) = 53
❷ write(53, ...) = 22
  ┌ openat(".es_temp_file", "O_WRONLY", ...) = 56
❸ │ fsync(56) = 0
  └ close(56) = 0
  ┌ close(53) = 0
❹ │ lstat(".es_temp_file", ...) = 0
  └ unlink(".es_temp_file") = 0
```

Listing 4.1 shows the syscalls and their corresponding arguments and return values observable with DIO. Listing 4.2 shows Elasticsearch's Java source code responsible for accessing the `.es_temp_file`.

The first `openat` is generated by the `Files.newOutputStream` method (❶), which opens an output stream used for writing data to the file (❷). An `IOUtils.fsync` method is then invoked to flush dirty pages to disk (❸). However, rather than using the already opened file descriptor, created from the first `openat` call, it internally reopens and closes the file again. Finally, upon the file's removal request (❹), three syscalls are issued: a `close` corresponding to the first `openat`, a `lstat`, and an `unlink`.

Listing 4.2. Elasticsearch source code for accessing the `.es_temp_file` file.

```
class FsHealthMonitor implements Runnable {
    static final String TEMP_FILE_NAME = ".es_temp_file";
    ...
    private void monitorFSHealth() {
        ...
        final Path tempDataPath = path.resolve(TEMP_FILE_NAME);
        Files.deleteIfExists(tempDataPath);
    ❶   try (OutputStream os = Files.newOutputStream(tempDataPath, ↩
            StandardOpenOption.CREATE_NEW)) {
         ❷  os.write(bytesToWrite);
         ❸  IOUtils.fsync(tempDataPath, false);
        }
    ❹   Files.delete(tempDataPath);
        ...
```

The information provided by DIO is relevant for identifying the file where this pattern happens (`.es_temp_file`) and, therefore, reducing the search space through the application's source code from >2.5M LoC to a single Java class[6] with 195 LoC.

This behavior shows that applications' methods can be translated into multiple syscalls by the high-level libraries these are using. For I/O-intensive files, this duplication may lead to performance degradation and I/O contention at the storage system [19].

This use case shows two important features of DIO: *i)* how it aids in exploring I/O interactions between applications and storage systems, and *ii)* how it can be used to find unexpected I/O patterns made by applications and help users narrow the portion of source code that must be inspected to correct these.

4.4.2 Identifying Fluent Bit's Erroneous Actions That Lead to Data Loss

DIO can assist developers and users in diagnosing the correctness of their applications. We demonstrate this by showing erroneous I/O access patterns that result in data loss.

For this use case, we consider Fluent Bit (v1.4.0), a high-performance logging and metrics processor and forwarder [44]. Existing issues report that data is lost when using the `tail` input plugin, which is used

[6] Elasticsearch's *FsHealthService* Java class: `https://github.com/elastic/elasticsearch/blob/91413fbd685ba022648abf2e8a0e291665a15a1b/server/src/main/java/org/elasticsearch/monitor/fs/FsHealthService.java#L130`

to fetch new content being added to log files.[7,8] Thus, we implemented a client program that simulates the generation of log files to be processed by Fluent Bit and mimics the I/O behavior reported in Issue #1875.[7] DIO was used to simultaneously trace and analyze the client program and Fluent Bit by filtering the syscalls belonging to these applications' processes.

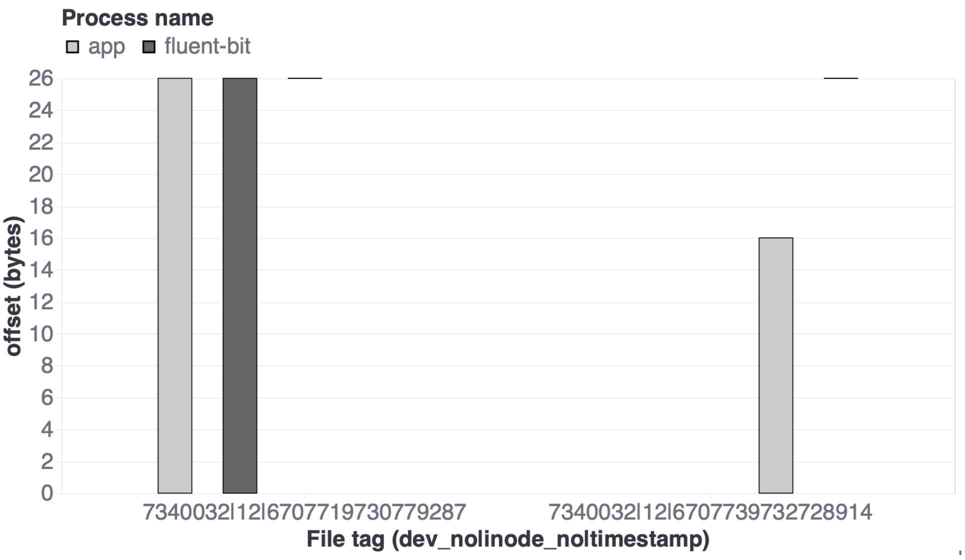
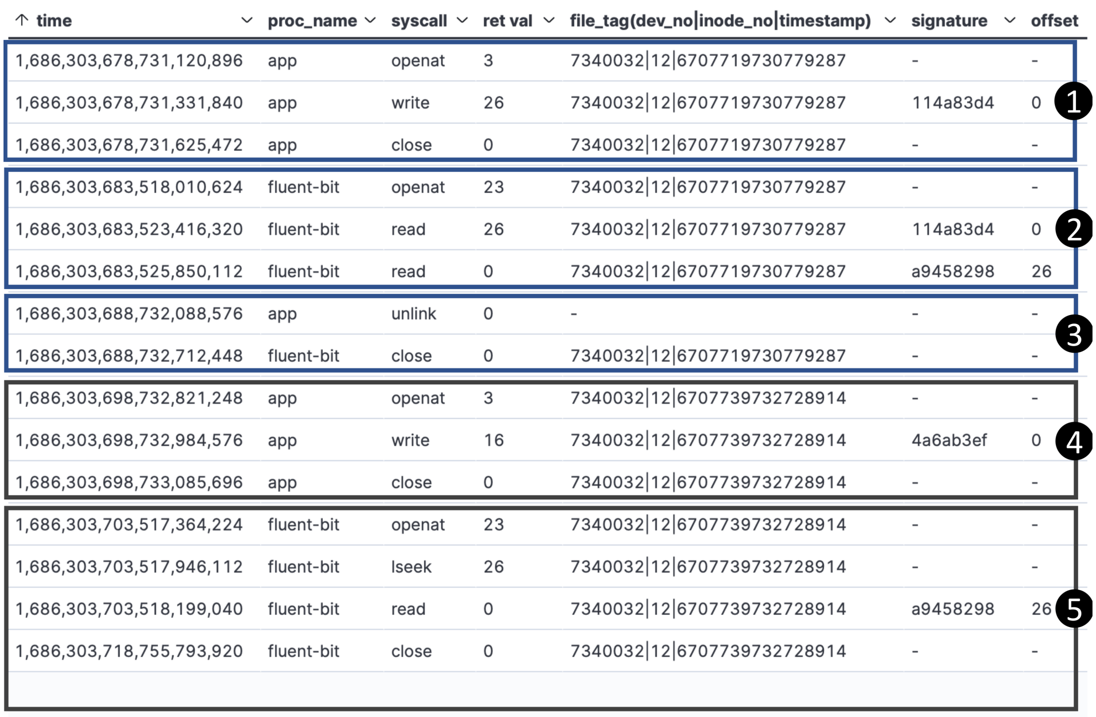


Figure 4.6. Accessed file offsets for Fluent Bit (v1.4.0).

Fig. 4.6 shows a visualization generated by DIO representing the accessed offsets for the `app.log` file for both client (*app*) and Fluent Bit (*fluent-bit*) applications. This visual representation shows that: *i)* two files are being accessed (different file tags); *ii)* the first file is accessed by both *app* and *fluent-bit* from offset 0 to offset 26; and *iii) app* accesses the second file from offset 0 to offset 16, but *fluent-bit* only accesses the offset 26.

Complementing this information with the one provided by the tabular visualization of Fig. 4.7 (also generated by DIO), one can further understand these file accesses. The *app* program starts by creating the `app.log` file, writing 26 bytes starting from offset 0, and closing the file (❶). Then, Fluent Bit (*fluent-bit*) detects content modification at the file, opens it, and reads 26 bytes from offset 0, which means that *fluent-bit* processes the full content previously written by *app* (❷). The hash *signatures* at the table validate that *fluent-bit* reads exactly the same content as written by *app*. Later, *app* removes the file with the `unlink` syscall, and *fluent-bit* closes the corresponding file descriptor (❸). At the OS level, this means that the inode number associated with this file (12) is now unused and will later be attributed to a new file. However, a possible scenario is this inode number being mapped to a newly created file with the same name. This happens when *app* creates a new file with the same name as the previous one (`app.log`) and writes 16 bytes to it (❹). The incorrect behavior reported at the issue, and observable with DIO, happens when *fluent-bit* opens the new log file for reading its content, but instead of reading from offset 0, as expected, it starts reading at offset 26 (❺). By starting at the wrong offset, the `read` syscall returns

[7]Wrong offsets issue from Fluent Bit's repository: `https://github.com/fluent/fluent-bit/issues/1875`
[8]Log missing issue from Fluent Bit's repository: `https://github.com/fluent/fluent-bit/issues/4895`



time	proc_name	syscall	ret val	file_tag(dev_no|inode_no|timestamp)	signature	offset
1,686,303,678,731,120,896	app	openat	3	7340032|12|6707719730779287	-	-
1,686,303,678,731,331,840	app	write	26	7340032|12|6707719730779287	114a83d4	0
1,686,303,678,731,625,472	app	close	0	7340032|12|6707719730779287	-	-
1,686,303,683,518,010,624	fluent-bit	openat	23	7340032|12|6707719730779287	-	-
1,686,303,683,523,416,320	fluent-bit	read	26	7340032|12|6707719730779287	114a83d4	0
1,686,303,683,525,850,112	fluent-bit	read	0	7340032|12|6707719730779287	a9458298	26
1,686,303,688,732,088,576	app	unlink	0	-	-	-
1,686,303,688,732,712,448	fluent-bit	close	0	7340032|12|6707719730779287	-	-
1,686,303,698,732,821,248	app	openat	3	7340032|12|6707739732728914	-	-
1,686,303,698,732,984,576	app	write	16	7340032|12|6707739732728914	4a6ab3ef	0
1,686,303,698,733,085,696	app	close	0	7340032|12|6707739732728914	-	-
1,686,303,703,517,364,224	fluent-bit	openat	23	7340032|12|6707739732728914	-	-
1,686,303,703,517,946,112	fluent-bit	lseek	26	7340032|12|6707739732728914	-	-
1,686,303,703,518,199,040	fluent-bit	read	0	7340032|12|6707739732728914	a9458298	26
1,686,303,718,755,793,920	fluent-bit	close	0	7340032|12|6707739732728914	-	-

Figure 4.7. Fluent Bit (v1.4.0) erroneous access pattern.

zero bytes, and the 16 bytes written by *app* are lost. Note that the hash signatures are different for the content written by *app* and read by *fluent-bit*.

To understand the reason for this behavior, we examined Fluent Bit's code responsible for reading new content entries in log files. Before reading a file, Fluent Bit updates the file position to the number of bytes already processed. This value is kept on a database for each tracked file, identified by its name plus inode number. Erroneously, database entries are not deleted when files are removed from the file system. Therefore, and going back to our running example, since the same file name (`app.log`) and inode number (12) are attributed to the newly created file, *fluent-bit* erroneously assumes that the first 26 bytes of the latter log file were already processed.

To validate the correction of this access pattern, we used DIO to analyze a more recent version of Fluent Bit (v2.0.5), where corrections were applied to avoid this data loss issue. Figs. 4.8 and 4.9 show similar visualizations for the new version. While the erroneous and correct versions present similar initial behavior (same file accesses for ❶-❹), the difference relies on the file offset being accessed by Fluent Bit (*flb-pipeline*) when reading from a new file (❺). This time, Fluent Bit starts reading from the beginning of the file (offset 0), being able to read the new 16 bytes written by *app*. In the correct version, the hash signatures for the 16 bytes written by *app* and read by *fluent-bit* match.

This example shows that DIO helps users diagnose incorrect I/O behavior from applications and find the root cause for dependability issues such as data loss. Further, while this example only showcases a small amount of lost data, it can be significantly higher when dealing with larger log files. Moreover, this use case also exemplifies how DIO helps validate the corrections applied to the applications' implementation.

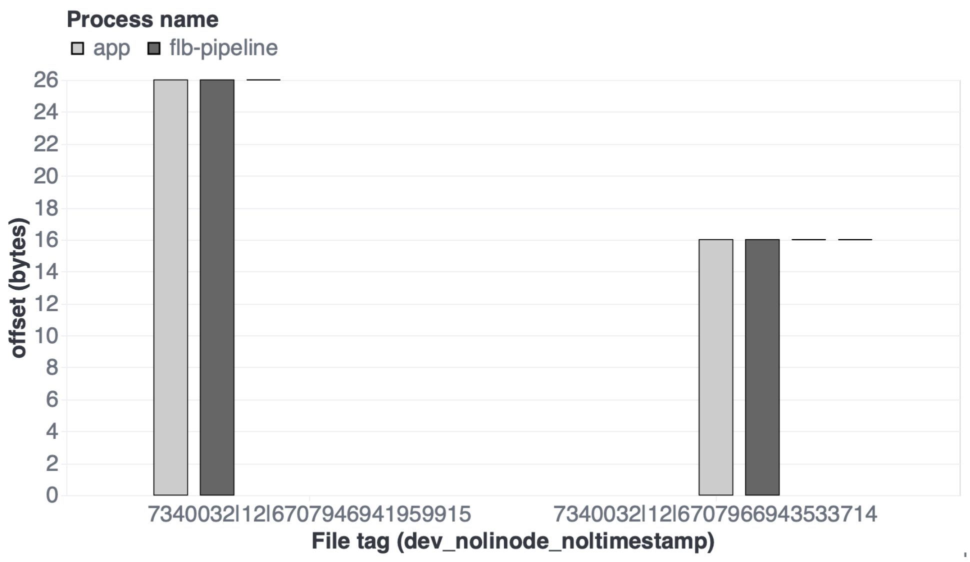
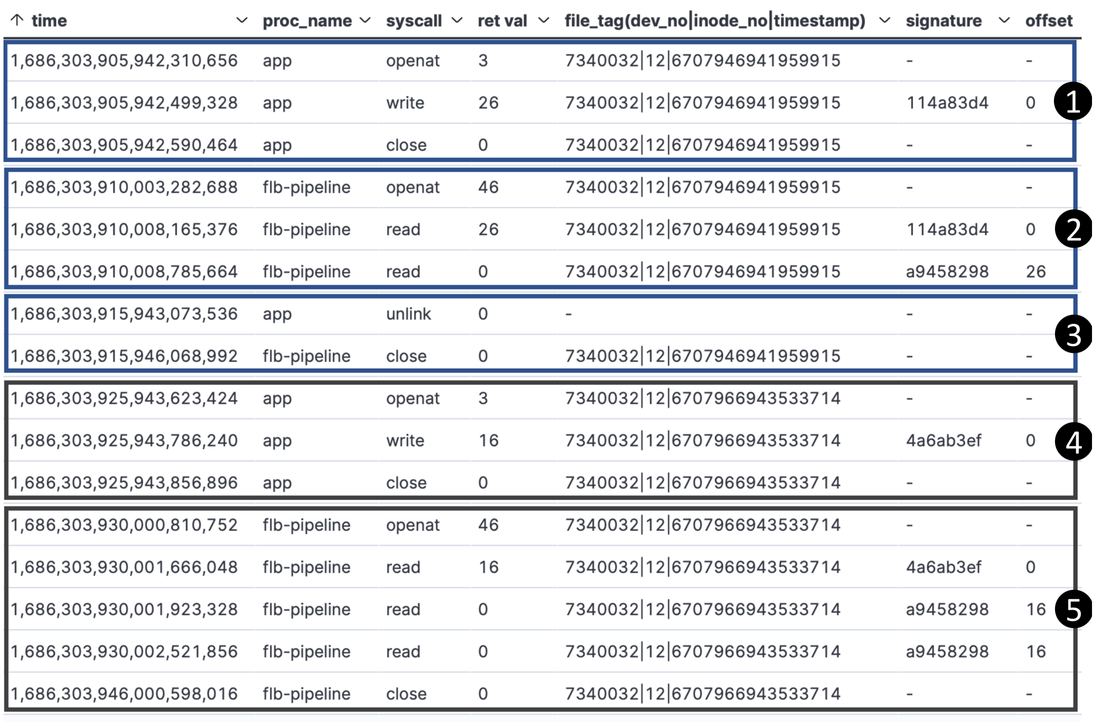


Figure 4.8. Accessed file offsets for Fluent Bit (v2.0.5).



↑ time	proc_name	syscall	ret val	file_tag(dev_no|inode_no|timestamp)	signature	offset	
1,686,303,905,942,310,656	app	openat	3	7340032|12|6707946941959915	-	-	
1,686,303,905,942,499,328	app	write	26	7340032|12|6707946941959915	114a83d4	0	1
1,686,303,905,942,590,464	app	close	0	7340032|12|6707946941959915	-	-	
1,686,303,910,003,282,688	flb-pipeline	openat	46	7340032|12|6707946941959915	-	-	
1,686,303,910,008,165,376	flb-pipeline	read	26	7340032|12|6707946941959915	114a83d4	0	2
1,686,303,910,008,785,664	flb-pipeline	read	0	7340032|12|6707946941959915	a9458298	26	
1,686,303,915,943,073,536	app	unlink	0	-	-	-	3
1,686,303,915,946,068,992	flb-pipeline	close	0	7340032|12|6707946941959915	-	-	
1,686,303,925,943,623,424	app	openat	3	7340032|12|6707966943533714	-	-	
1,686,303,925,943,786,240	app	write	16	7340032|12|6707966943533714	4a6ab3ef	0	4
1,686,303,925,943,856,896	app	close	0	7340032|12|6707966943533714	-	-	
1,686,303,930,000,810,752	flb-pipeline	openat	46	7340032|12|6707966943533714	-	-	
1,686,303,930,001,666,048	flb-pipeline	read	16	7340032|12|6707966943533714	4a6ab3ef	0	
1,686,303,930,001,923,328	flb-pipeline	read	0	7340032|12|6707966943533714	a9458298	16	5
1,686,303,930,002,521,856	flb-pipeline	read	0	7340032|12|6707966943533714	a9458298	16	
1,686,303,946,000,598,016	flb-pipeline	close	0	7340032|12|6707966943533714	-	-	

Figure 4.9. Fluent Bit (v2.0.5) correct access pattern

4.4.3 Finding the Root Cause of RocksDB's Performance Anomalies

We now demonstrate how DIO can also ease the process of diagnosing performance issues by identifying the root cause for high tail latency at client requests issued to RocksDB, an embedded KVS [43].

This phenomenon was first observed in SILK [12] and, therefore, we followed the same testing methodology to reproduce it. We used the *db_bench* benchmark configured with 8 client threads performing a mixture of read-write requests in a closed loop (YCSB A [29]).[9] RocksDB was configured with 8 background threads, namely 1 for flushes and 7 for compactions. Fig. 4.10 reports a sample of a 5-hour-long execution and depicts the 99th percentile latency experienced by clients. Throughout this sample, clients observe several latency spikes that range between 1.5 ms and 3.5 ms.

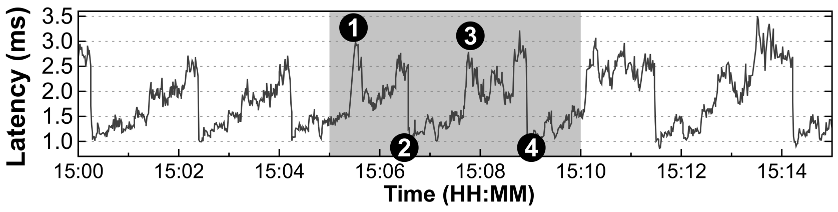


Figure 4.10. 99th percentile latency for RocksDB client operations.

Finding the root cause for this performance penalty through RocksDB codebase instrumentation would require inspecting more than 440K LoC and adding debugging code to several core components. Alternatively, with DIO, one can easily trace, analyze, and visualize RocksDB execution, as depicted in Fig. 4.11. Since the workload is data-oriented, we configured DIO's *tracer* to capture exclusively `open`, `read`, `write`, and `close` syscalls. Client threads are represented as `db_bench`, while `rocksdb:high0` respects to the flushing thread, and the remainder (`rocksdb:lowX`) to compaction threads.

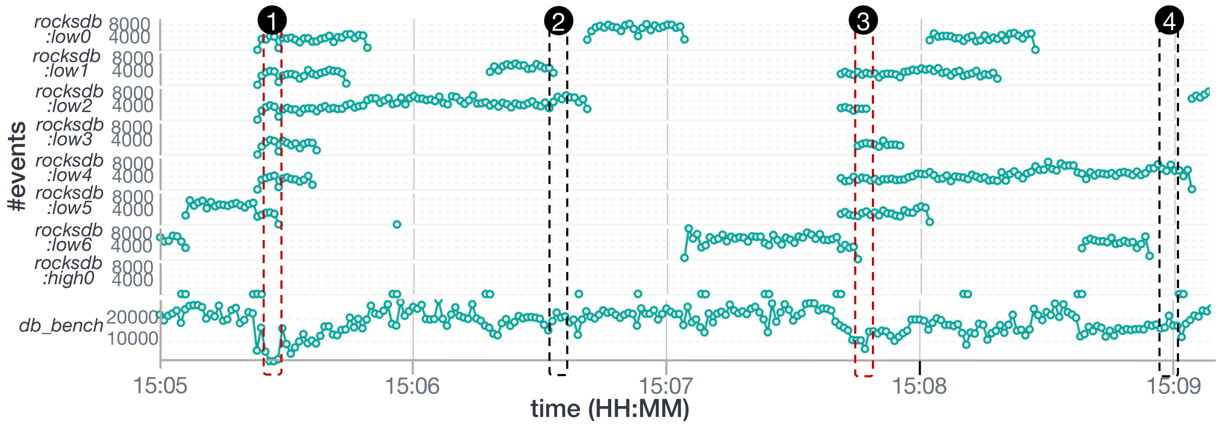


Figure 4.11. Syscalls issued by RocksDB over time, aggregated by thread name. `db_bench` includes the 8 client threads, `rocksdb:low[0-6]` refers to each compaction thread, and `rocksdb:high0` refers to the flush thread.

By observing the syscalls submitted over time by different RocksDB threads, one can identify performance contention. Namely, as shown by the highlighted red boxes, when multiple compaction threads submit I/O requests, the number of syscalls of `db_bench` threads decreases, causing an immediate tail latency spike perceived by clients, as depicted in Figs. 4.11 and 4.10 (in intervals ❶ and ❸, at least 5 compaction threads submit requests). When fewer compaction threads perform I/O, the performance of `db_bench` improves both in terms of tail latency and throughput (in intervals ❷ and ❹, only 1 to 2 compaction threads are performing I/O).

[9]`https://github.com/facebook/rocksdb/wiki/Benchmarking-tools`

If one complements the previous observation with knowledge of how Log-Structured Merge tree (LSM-tree) KVSs work, the problem becomes clear: RocksDB uses foreground threads to process client requests (`db_bench` threads), which are enqueued and served in FIFO order. In parallel, background threads serve internal operations, namely flushes (`rocksdb:high0`) and compactions (`rocksdb:lowX`). Flushes ensure that in-memory key-value pairs are sequentially written to the first level of the persistent LSM-tree (L_0), and these can only proceed when there is enough space at L_0. Compactions are held in a FIFO queue, waiting to be executed by a dedicated thread pool. Except for low-level compactions ($L_0 \rightarrow L_1$), these can be made in parallel.

A common problem of compactions, however, is the interference between I/O workflows, generating latency spikes for client requests. Specifically, latency spikes occur when client threads cannot proceed because $L_0 \rightarrow L_1$ compactions and flushes are slow or on hold, which happens, for instance, when several threads compete for shared disk bandwidth (creating contention).

This is precisely the phenomenon identified in SILK, which can negatively impact the response time and even the availability of KVSs and services that use them [35, 70], and that can be observed with DIO without any code instrumentation.

4.4.4 Performance Impact and I/O Events Handling

We now analyze the performance impact induced by diagnosing I/O calls with DIO.

Table 4.2. Minimum DIO's tracing mode for successfully diagnosing each use case.

	raw	***detailedP***$_{fds}$	***detailedP***$_{all}$	***detailedP***$_{all}$***C***$_{khash}$
RocksDB	✓✓	✓	✓	✓
Redis	✗	✓✓	✓	✓
Elasticsearch	✗	✗	✓✓	✓
Fluent Bit	✗	✗	✗	✓✓

DIO's Setups. For these experiments, we configured DIO to capture only the required information for diagnosing the aforementioned I/O issues. Namely, the RocksDB use case (§4.4.3) requires information about the type and number of syscalls, their timestamp, and the name of the processes that issued them. Thus, DIO can be configured with the less detailed tracing mode (*raw*).

For the Redis use case (§4.1), we also need to collect information about file paths and file offsets, thus requiring DIO's detailed tracing mode. Since the syscalls relevant for this use case all handle file descriptors, we can avoid collecting the syscalls arguments that require transferring large amounts of data from kernel to user space, as explained in §4.3.5 (*i.e.*, file paths contained at the syscalls' arguments and data buffers' content). We refer to this setup as *detailedP*$_{fds}$.

Elasticsearch use case (§4.4.1), on the other hand, requires the analysis of syscalls whose file path information is obtained from their arguments. Thus, for this use case, we configure DIO with the *detailedP*$_{all}$ setup, which also collects the file paths from syscall arguments.

Lastly, the Fluent Bit use case (§4.4.2) was configured with the $detailedP_{all}C_{khash}$, which also captures the data buffers' content and calculates a hash sum in kernel space, which is helpful for validating when both applications are processing the same data content.

Table 4.2 shows DIO's minimal configuration for successfully diagnosing each use case (✓✓), while pointing other more comprehensive configurations that could be used to observe these (✓). §4.5 provides further information about DIO's setups.

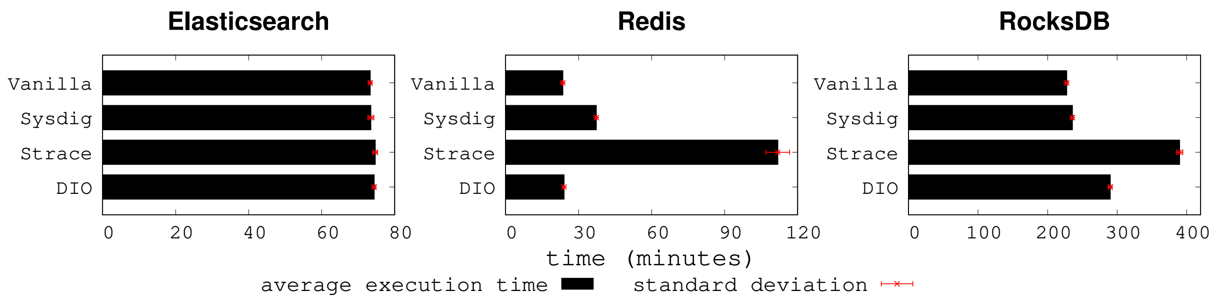


Figure 4.12. Average execution time for Elasticsearch, Redis and RocksDB use cases with DIO, Sysdig and Strace.

Fig. 4.12 shows the execution times of Elasticsearch, Redis, and RocksDB under the workloads described at §4.4.1, §4.1, and §4.4.3, respectively. Fluent Bit's use case was excluded as it does not include a benchmark. For each application, we compared its *vanilla* deployment (*i.e.*, without tracing its execution) with DIO and two state-of-the-art syscall tracers: Strace [109] and Sysdig [111].

Performance Analysis. The performance overhead imposed over *vanilla* setups is influenced by the I/O load generated by each application. For Elasticsearch, the least I/O intensive application, all tracers introduce negligible overhead, increasing the *vanilla* execution time (73.31 min) by up to 82s.

For Redis, DIO, Sysdig and Strace increase *vanilla* execution time (23.5 min) by 1.04x (24.0 min), 1.62x (37.3 min), and 4.86x (111.9 min), respectively. By filtering events to Redis' working directory, Sysdig and DIO can discard non-storage related requests (*i.e.*, `read` and `write` syscalls issued to network sockets), which account for ≈99% of the events generated by Redis (as shown in §4.1). However, by applying these filters in kernel space, DIO reduces the computation in the critical path of I/O requests and the amount of data sent to user space, imposing less overhead than Sysdig. Strace cannot filter events by directory paths (only by specifying all file paths) and, consequently, intercepts all generated syscalls, including those targeting network sockets, which explains its significantly higher performance overhead.

For RocksDB, the most I/O intensive application, Sysdig, DIO, and Strace increase *vanilla* execution time (227.5 min) by 1.07x (235.6 min), 1.38x (290.1 min), and 1.74x (389.8 min), respectively. Although Sysdig presents the smallest performance overhead, DIO is the only tracer capable of providing near-real-time analysis and visualization of collected data and can still reduce the overhead imposed by Strace.

I/O Events Handling. As discussed in §4.2, DIO uses a fixed-sized ring buffer to collect information at user space, which was configured with 256 MiB per CPU core for these experiments. When this buffer is full (*i.e.*, if kernel processes are producing I/O events to the ring buffer at a faster pace than the user space processes can consume them), new I/O events being intercepted at the kernel-level are discarded. For the aforementioned experiments, DIO is able to capture all storage syscalls generated by Elasticsearch (1M) and Redis (600K). For RocksDB, given its more intensive I/O behavior, 6% of the issued syscalls (≈34M of 538M) were discarded at the ring buffer and, therefore, not stored at DIO's *Backend*.

Regarding the storage space needed by DIO's *Backend* to store the collected information, the Redis, Elasticsearch and RocksDB use cases require approximately 86MiB, 282MiB and 90GiB, respectively.

4.4.5 Summary

The previous use cases demonstrate that DIO is useful for diagnosing distinct I/O patterns. Namely, with Redis, RocksDB, and Fluent Bit, we show that DIO can be used by developers to observe and confirm known issues and to validate their corrections. With Elasticsearch, we show that our tool is useful when users wish to explore unknown applications. Indeed, DIO is used to observe an inefficient I/O pattern that was not known a priori. Our integrated tracing and analysis pipeline enables users to observe these I/O patterns without resorting to code instrumentation or manually combine multiple tools.

Experimental results show that DIO can collect, parse, and forward to the analysis pipeline all the required tracing information while imposing reduced performance overhead. When compared to Strace, DIO reduces execution time for all applications. When compared with Sysdig, performance overhead varies with the amount of information captured at kernel, sent to user space, and reported to users. Despite the discarded I/O events in RocksDB, we show that DIO can pinpoint resource contention and help diagnose its root cause. Moreover, unlike in Strace and Sysdig, DIO's traced information is automatically made available for analysis as soon as it is collected and transmitted to the *Backend* component.

4.5 Experimental Study

We now study how DIO behaves under intensive I/O workloads for answering the following questions:

- *What is the performance and resource usage of DIO when tracing I/O intensive applications?*
- *How much information can DIO capture without discarding events?*
- *What is the performance and accuracy impact of DIO's configurations (e.g., ring buffer size, batch size) and optimizations (i.e., tracing modes and filters)?*
- *How does DIO compare to other state-of-the-art syscall tracers?*

To that end, we first compare DIO with other state-of-the-art solutions (§4.5.1), and then further study DIO's inline pipeline (§4.5.2), adaptability to different I/O rates (§4.5.3), and the impact of its filtering mechanisms (§4.5.4).

Table 4.3. Description of each setup used in the experiments for Strace, Sysdig, and DIO tracers.

Setup	Description	Tracer
Raw	Syscalls information is saved in its *raw* format without any kind of preprocessing.	Strace DIO
DetailedP$_{args}$	Syscalls information is preprocessed, and file paths are obtained from syscall arguments only (*i.e.*, no translation of file descriptors to file paths).	Strace
DetailedP$_{fds}$	Syscalls information is preprocessed, and file paths are obtained from file descriptors only (*i.e.*, file paths from syscalls' arguments are discarded).	DIO
DetailedP$_{all}$	Syscalls information is preprocessed, and file paths are obtained from both file descriptors and syscalls' arguments.	Strace Sysdig DIO
DetailedP$_{all}$C$_{plain}$	Similar to *detailedP$_{all}$*, but including data buffers' content in plaintext.	Strace Sysdig
DetailedP$_{all}$C$_{uhash}$	Similar to *detailedP$_{all}$*, but including a hash sum of data buffers' content computed in user space.	DIO
DetailedP$_{all}$C$_{khash}$	Similar to *detailedP$_{all}$*, but including a hash sum of data buffers' content computed in kernel space.	DIO

State-of-the-art Syscall Tracers. For the conducted experiments, we compared DIO with:

- *Strace*: a popular diagnostic, debugging, and instructional user space utility that leverages the `ptrace` kernel feature to non-intrusively intercept syscalls [109].
- *Sysdig*: a tool for system troubleshoot, analysis, and exploration that also leverages the eBPF technology to intercept syscalls [111].

Similar to DIO, these tracers intercept syscalls invoked by user space applications and collect information regarding their type, arguments, and return value. They also offer different filtering capabilities and allow configuring which data to collect (*e.g.*, enabling/disabling the collection of data buffers).

Setups. In other to evaluate the impact of collecting more or less detailed information, we configured each tracer (whenever possible) with the setups described in Table 4.3.

Storage Backends. While DIO offers an integrated analysis pipeline to collect, analyze and visualize traced data, Strace and Sysdig focus only on the tracing phase, saving collected data to disk. To fairly compare the three tracers and study the impact of different storage backends, in addition to the default deployment of DIO (*i.e.*, sending collected information directly to the remote analysis pipeline), we also evaluated DIO's performance when saving traced data directly to disk.

Moreover, to study the feasibility of building an integrated diagnosis pipeline with existing tracing solutions, we use the Logstash [10] data processing tool to automatically parse and forward Sysdig events to a similar inline analysis pipeline as in DIO (*i.e.*, composed by Elasticsearch and Kibana). Specifically, for these experiments, Sysdig is configured to write collected data to the Standard Output, which is redirected (via a Unix pipe) to Logstash. The latter reads, parses, and forwards collected events, in batches, to Elasticsearch.

Workload and Collected Metrics. To produce a stress-test scenario, we used the Filebench benchmark, a popular framework for file system and storage benchmarking [114].[10] Experiments consisted of running Filebench with the *FileServer* workload, configured to access 10,000 files, each sizing 128 KiB, through 4 threads performing storage I/O requests for 20 mins. The experiments were conducted in the same testbed as described in §4.4. Results include the average and standard deviation of the number of operations per second (*ops/s*) for three independent runs. Unless stated otherwise, the standard deviation for all experiments is equal or inferior to 3% of the corresponding throughput.

The Dstat [90] tool was used to obtain system statistics, including CPU, memory, and network usage. Finally, the storage overhead imposed by each tested setup (*i.e.*, size of the generated trace file or Elasticsearch's index size) was also computed, as well as the number of intercepted syscalls, including *complete* (events saved with all the information), *incomplete* (events saved with partial information), and *lost* events (events discarded in kernel space).

4.5.1 Comparison With State-of-the-art Tracers

First, we compare DIO's performance, resource usage, and tracing accuracy against Strace and Sysdig in a stress-test environment (depicted in Fig. 4.13). We start by doing an individual analysis for each tracer, and then we discuss how these compare to each other.

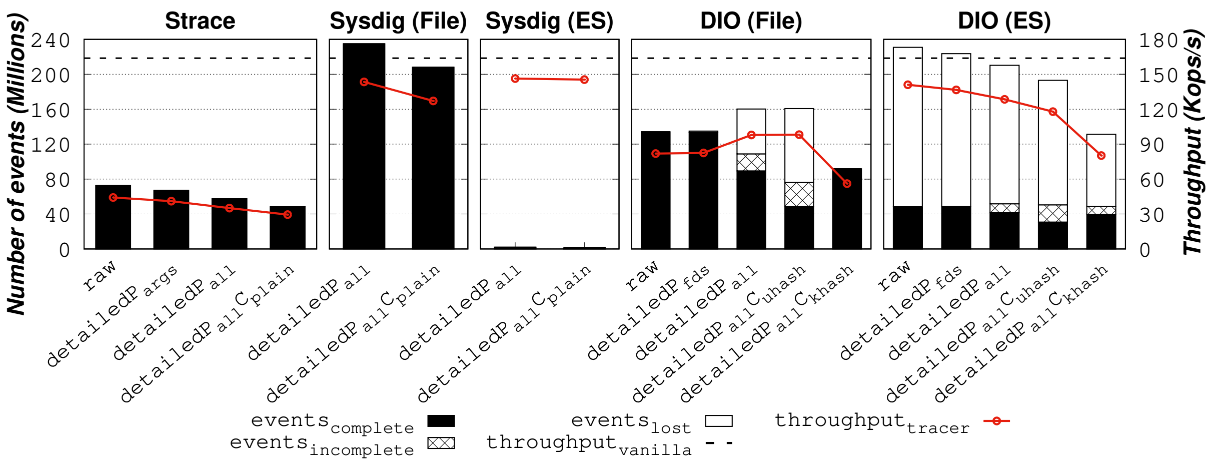


Figure 4.13. Filebench's performance overhead and collected events for Strace, Sysdig, and DIO.

4.5.1.1 Per-tracer Analysis

Vanilla. The Filebench benchmark running without any tracer (*vanilla* setup) generates approximately 164 Kops/s (depicted by the black dashed line - - -). Unless stated otherwise, the performance overhead values discussed in this section correspond to the decrease in throughput percentage of a given setup when compared with the *vanilla* setup.

[10]Filebench benchmark: `https://github.com/filebench/filebench`

Strace. Strace imposes high performance overhead over Filebench's workload (depicted by the red line —), reducing throughput by 73% with the least detailed setup (*raw*). The imposed overhead increases further as more detailed information is captured. Namely, with the more detailed setup ($detailedP_{all}C_{plain}$), Strace achieves only 30 Kops/s, increasing the overhead to 82%.

While Strace can save all intercepted syscalls without losing traced data (*i.e.*, no *incomplete* or *lost* events), the number of collected events decreases with more detailed setups, ranging from 73M (*raw*) to 48M ($detailedP_{all}C_{plain}$). This is a consequence of Strace's performance overhead, which decreases the number of ops/s done by Filebench and, consequently, the total number of issued syscalls.

Sysdig. Sysdig incurs reduced performance overhead over Filebench, decreasing throughput by 12% with the $detailedP_{all}$ setup. Similarly to Strace, capturing more information imposes higher overhead. With the $detailedP_{all}C_{plain}$ setup, Sysdig achieves 127 Kops/s, increasing the overhead to 22%.

With the reduced performance overhead, Filebench generates more ops/s, and consequently, more syscalls are performed in total. When writing data to disk, Sysdig saves between 208M ($detailedP_{all}$) to 235M ($detailedP_{all}C_{plain}$) of *complete* events, with only 41 incomplete events (*i.e.*, missing information about file paths). However, in these experiments, Sysdig is unable to report the process name for all captured events.

When sending collected data to Elasticsearch, Sysdig still imposes reduced overhead (≈11%) but only saves 2M events, some of them with incomplete data (between 131 and 314 *incomplete* events). This is a consequence of Sysdig producing data faster than Logstash can process it, filling the Unix pipe connecting Sysdig and Logstash and forcing Sysdig to discard events. Interestingly, when saving data to Elasticsearch, Sysdig can obtain the process name for all events.

DIO. The performance overhead imposed by DIO varies depending on the storage backend used (*i.e.*, file or Elasticsearch) and the amount of computation performed in kernel space (*i.e.*, in the critical path of I/O requests).

When writing data to disk, the *raw* and $detailedP_{fds}$ setups process all intercepted syscalls, saving ≈134M *complete* events, but reduce Filebench's throughput by 50% (with a standard deviation of 6% for $detailedP_{fds}$). The $DetailedP_{all}$ and $detailedP_{all}C_{uhash}$ setups start losing traced data (between 20M-28M *incomplete* and 51M-85M *lost* events) but impose less overhead (40%), achieving 98 Kops/s.

By capturing the file paths from syscall arguments ($detailedP_{all}$) and the content of data buffers ($detailedP_{all}C_{uhash}$), DIO needs to transfer larger events from kernel to user space. Moreover, by computing a hash sum of the buffers' content in user space, DIO further delays the events' pulling from the *ring buffer*. If no space is available on the *ring buffer* to send an event to user space, the event is discarded in the kernel and considered *lost*. Interestingly, as the number of lost events increases, the performance overhead decreases due to the reduction in the amount of data copied from the kernel to user space in the critical I/O path of requests.

The $detailedP_{all}C_{khash}$ setup reduces the amount of data to transfer to user space by computing the hash sum of buffers' content in kernel space. While it allows minimizing the loss of events (*i.e.*, no *incomplete* or *lost* events are observed), it ends up adding heavy computation to I/O requests' critical path, imposing higher overhead (66%).

A similar phenomenon is visible when using Elasticsearch as DIO's storage backend: DIO saves fewer events (48M to 52M) but imposes less overhead (14% to 51%). These results show two other phenomena. First, DIO's rate for processing and saving events to Elasticsearch is limited to ≈43K events/s, thus filling up the *ring buffer* more quickly and losing more events. Second, when writing data to disk, the performance overhead is especially dictated by the extra computation of copying data to user space. However, when writing data to Elasticsearch, the overhead imposed by the extra processing in kernel to gather more detailed information has a higher impact (*i.e.*, more detailed setups impose higher overhead). This is more noticeable for the $detailedP_{all}C_{khash}$ setup that due to computing the hash sum in kernel, reduces throughput by 51% (with a standard deviation of 4%).

4.5.1.2 Comparative Analysis

Next, we compare the three tracers regarding their imposed performance overhead, tracing accuracy, and resource usage.

Performance Overhead. Strace imposes the highest overhead over Filebench's workload, while Sysdig imposes the lowest. DIO offers better results than Strace for all setups and storage backends, resulting from using a low overhead technology, eBPF, instead of a more costly approach like `ptrace`. When configured with the Elasticsearch backend, DIO provides results closer to Sysdig. In general, all tracers impose higher overhead when collecting more detailed information.

Collected Events. By imposing the lowest performance overhead, Sysdig is the tracer that saves more events when writing these to disk. In the same way, by imposing the highest overhead, Strace saves fewer events than the others. When sending data to Elasticsearch, both Sysdig and DIO are forced to discard events. However, by including a custom implementation to communicate directly with Elasticsearch, DIO can save significantly more events than Sysdig. As in the performance overhead results, collecting more detailed information generally translates into more *incomplete* and *lost* events for DIO and Sysdig.

CPU Usage. Fig.4.14 shows the CPU usage for each tracer and setup. Due to its synchronous approach for intercepting syscalls and higher performance overhead, Strace exhibits significant CPU *idle* time, which increases when more detailed data is collected. Vanilla, Sysdig, and DIO setups have negligible *idle* time but exhibit different values for the time spent in user space (*usr*) and kernel space (*sys*). The additional *usr* time used by DIO is explained by the processing done by our solution in user space, as explained in §4.2.2). However, DIO's *usr* time reduces, while *sys* time increases, for setups capturing more detailed information, as these require extra processing at the critical path of I/O requests.

Memory Usage. Regarding memory consumption (Fig. 4.15), it is noticeable an increased usage of cache resources (*cach*) for setups writing traced data to disk, which is a consequence of using more space from the OS's page cache. This value increases further when considering more detailed setups.

DIO presents higher values for *used* memory, which is explained by the eBPF maps and the *ring buffer* used by our tracer (*e.g.,* the default size of the *ring buffer* is 1GiB in total) and by writing events in batches to Elasticsearch for increased performance (*i.e.*, the default configuration is 7MiB per thread, 28MiB in total). The latter justifies why our solution uses more memory when using the Elasticsearch backend instead of the file one. Finally, the increase in *used* memory for Sysdig, with the Elasticsearch backend, is due to Logstash's internal buffers for processing traced data.

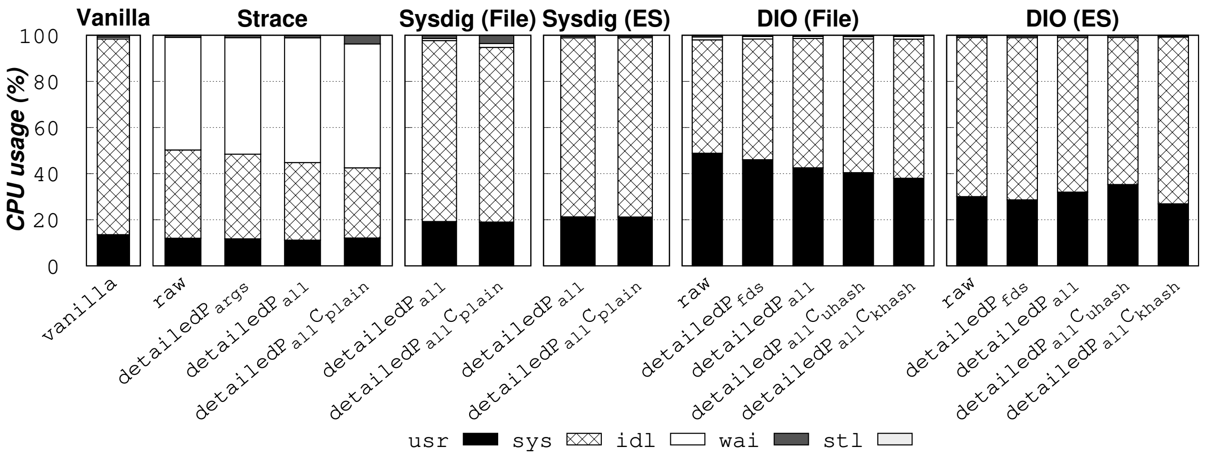


Figure 4.14. CPU usage by Strace, Sysdig, and DIO.

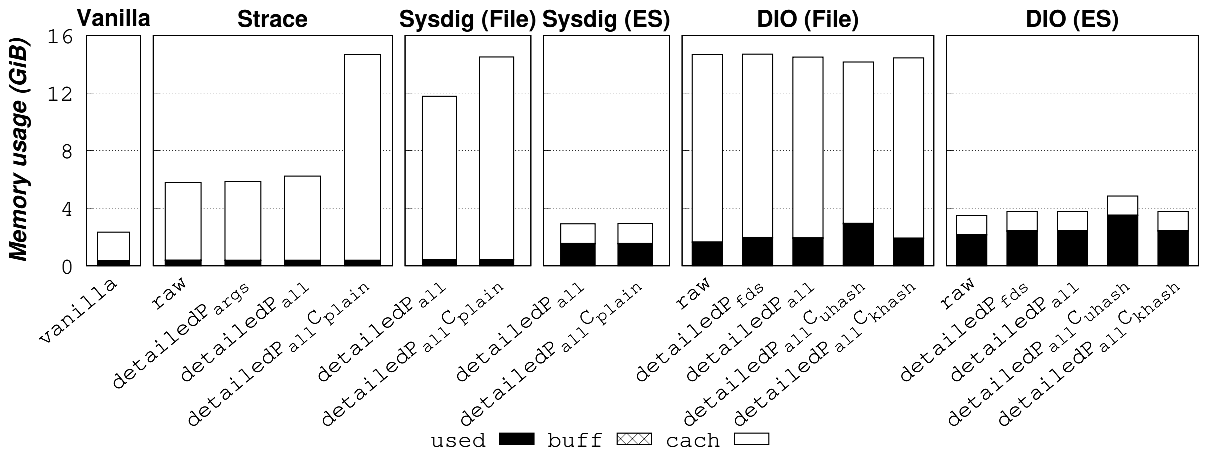


Figure 4.15. Memory usage by Strace, Sysdig, and DIO.

Storage Usage. Figs. 4.16 and 4.17 show the storage space used by each tracer when writing data to disk and when sending events to Elasticsearch, respectively. Strace generates a trace file with a smaller size (around 9GiB for *raw*, *detailedP$_{args}$*, and *detailedP$_{all}$* setups), which can be explained by the smaller number of events it collects. DIO collects more events and writes data to disk in a JSON format, thus generating larger files (from 40GiB with *detailedP$_{all}$C$_{uhash}$* to 70GiB with *detailedP$_{fds}$*).

Sysdig, on the other hand, writes data to disk in a binary format, and therefore, although being the tracer that collects more events, it can create compact files (≈31GiB with *detailedP$_{all}$* setup). Nonetheless,

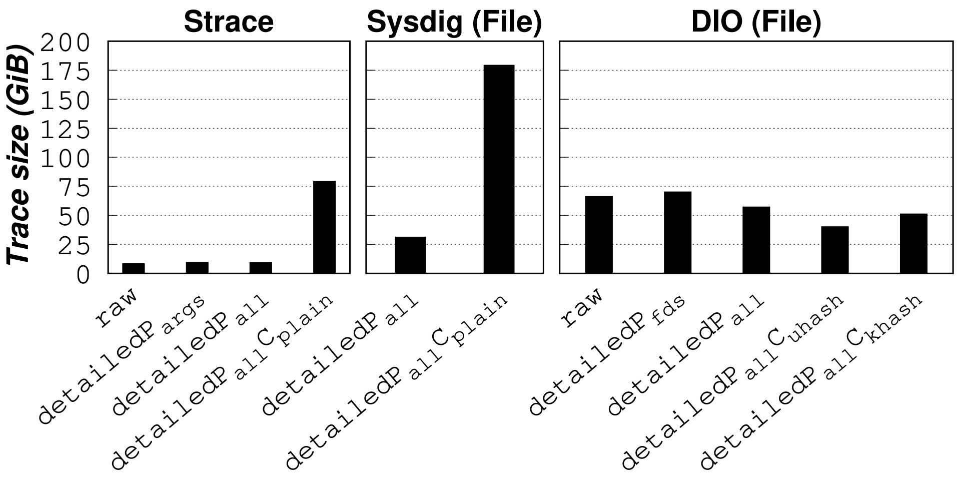


Figure 4.16. Size of the traces generated by Strace, Sysdig and DIO when writing these to disk.

both Strace and Sysdig generate larger trace files when capturing the data buffer's content (*detailedP$_{all}$C$_{plain}$* setup), generating files of 79GiB and 179GiB, respectively. By contrast, DIO minimizes the trace size (≤50GiB) by saving a hash sum of the buffers' content (*detailedP$_{all}$C$_{uhash}$* and *detailedP$_{all}$C$_{khash}$* setups).

When sending data to Elasticsearch, the resulting index size for Sysdig is around 1.4GiB, while for DIO it varies between 8.9GiB and 12.1GiB. The difference between the two is mainly dictated by the number of collected events and the amount of detailed information captured.

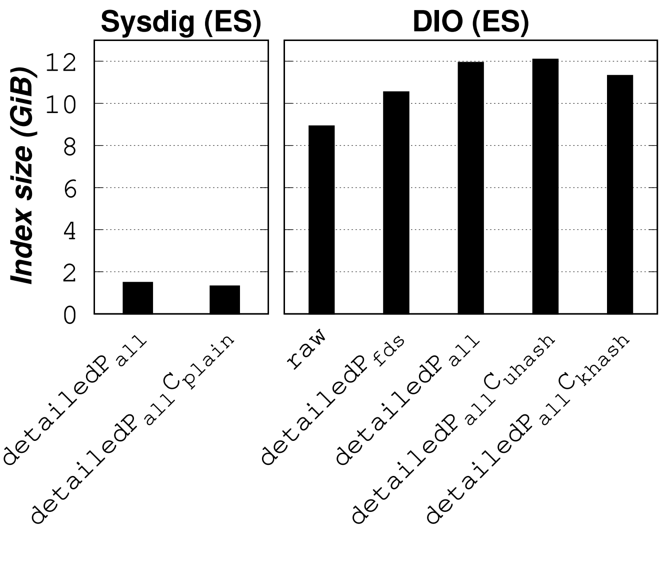


Figure 4.17. Size of the indices generated by Sysdig and DIO when using Elasticsearch as storage backend.

Network Usage. As depicted in Fig. 4.18, when considering Sysdig and DIO using Elasticsearch as the storage backend, DIO consumes more network bandwidth (*i.e.*, 2MiB/s for Sysdig and between 17MiB and 20MiB/s for DIO). Since DIO sends more MiBs per second to Elasticsearch, it can process traced data from the *ring buffer* faster and save more information at the *Backend*, as shown in Fig. 4.13.

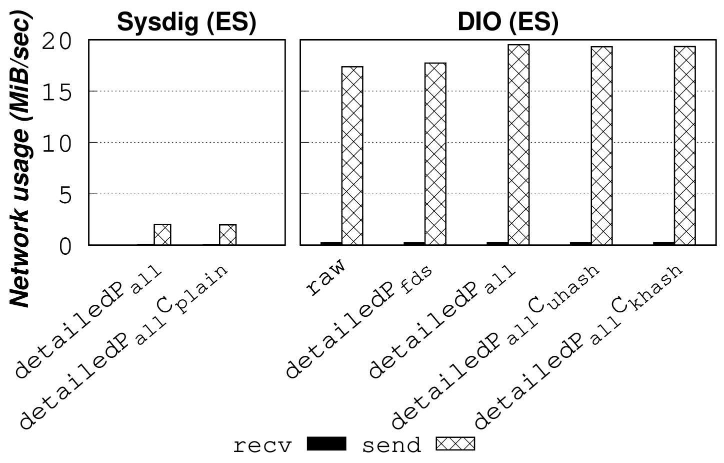


Figure 4.18. Network usage by Sysdig and DIO with the Elasticsearch backend

.

From the previous results and analysis, one can extract the following main takeaways:

Takeaway 1. *Capturing more detailed information from syscalls induces higher performance and resource usage overhead. Also, when syscalls are intercepted asynchronously (i.e., in DIO and Sysdig), it may lead to a larger number of incomplete and lost events. For synchronous approaches (i.e., in Strace), the performance overhead is more noticeable.*

Takeaway 2. *The storage backend where traced data is stored influences the number of collected events and performance overhead. A slower backend (i.e., using Elasticsearch instead of a high-performance local disk) reduces the overhead over the application but leads to a higher number of incomplete and lost events for Sysdig and DIO.*

Takeaway 3. *Tracers using the eBPF technology (i.e., Sysdig and DIO) exhibit distinct tradeoffs related to the balance between the computation done in user space and in kernel space. More computation in user space delays the collection of events from the ring buffer, which results in more discarded events and less performance overhead. More computation in kernel induces a higher performance penalty for the traced application.*

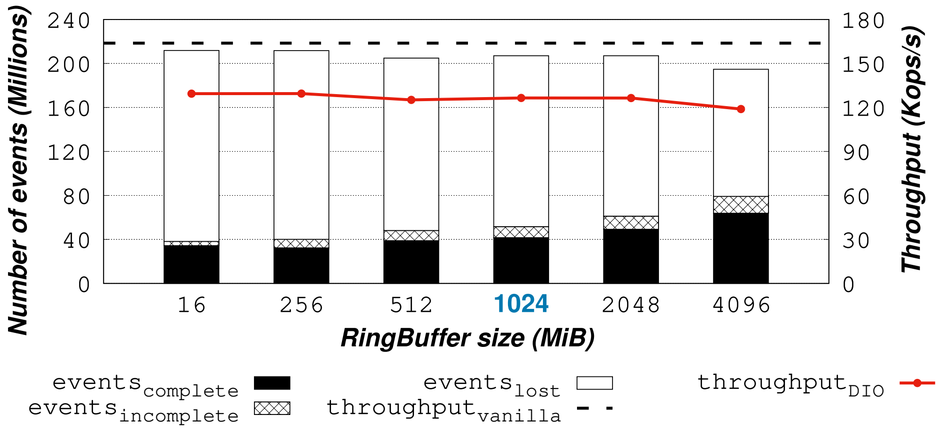
Takeaway 4. *DIO offers the best tradeoff in terms of performance overhead and collected events when considering a full inline pipeline for tracing, analyzing, and visualizing I/O syscalls (i.e., it captures between 23x to 28x more events when compared with Sysdig, while maintaining performance overhead under 51%). On the other hand, Sysdig presents the best tradeoff regarding performance overhead and collected events when considering only the tracing step for a local disk backend.*

4.5.2 Inline Analysis Pipeline

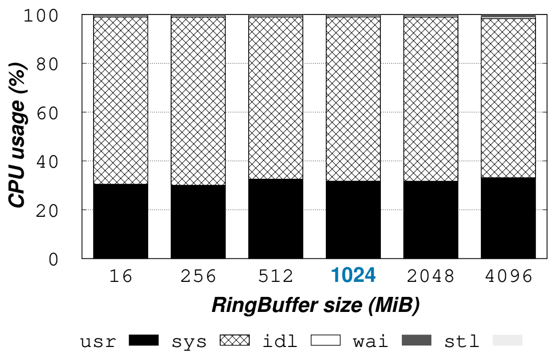
Next, we assess the best configurations to provide an efficient inline analysis pipeline (*i.e.*, in which traced data is sent directly to the Elasticseach backend). We start by studying the impact of varying the *ring buffer* size in DIO, and evaluating both Sysdig and DIO when sending batches of different sizes to Elasticsearch. Then, we discuss the advantages and drawbacks of following an inline *vs* offline approach (*i.e.*, saving data to disk and sending it posteriorly to Elasticsearch). For these experiments, both tracers are configured with the *detailed*P_{all} setup for a fair comparison.

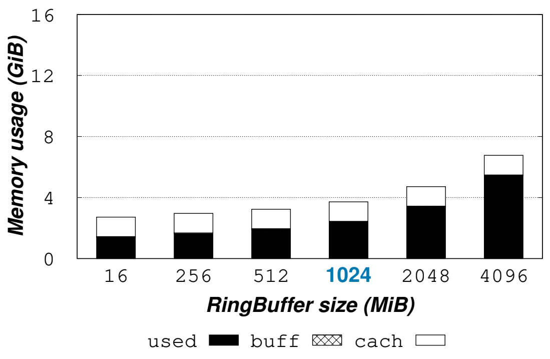
4.5.2.1 Ring Buffer's Size Impact in DIO

To further explore *Takeaway 3*, it is important to assess the impact that different *ring buffer* sizes have on DIO's performance and amount of collected events.



(a) Throughput and collected events.





(b) CPU usage.

(c) Memory usage.

Figure 4.19. Performance overhead, collected events, and resource usage for different *ring buffer* sizes in DIO. The blue color pinpoints the default configuration.

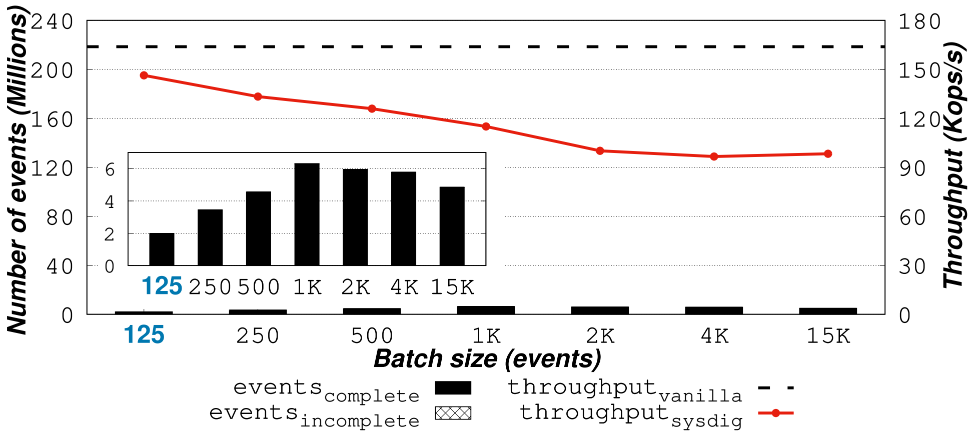
As expected and shown in Fig. 4.19a, the larger the *ring buffer*, the more events DIO collects. With a smaller configuration (16MiB), DIO saves 38M events. If a larger *ring buffer* is used (4096MiB), it saves up to 79M events. However, increasing the *ring buffer* size impacts the performance overhead imposed over Filebench. Namely, overhead ranges from 21% (16MiB configuration) to 27% (4096MiB configuration).

Regarding resource usage, varying the *ring buffer* size has minimal impact on CPU consumption, as shown in Fig. 4.19b. As for memory consumption (depicted in Fig. 4.19c), the larger the *ring buffer*, the more *used* memory DIO needs, ranging from 1.4GiB (16MiB) to 5.4GiB (4096MiB). The 1024MiB configuration offers a good tradeoff between performance overhead (23%), collected events (52M), and memory usage (3.7GiB in total), and therefore was selected as the default *ring buffer* configuration for DIO in all the experiments discussed at this section.

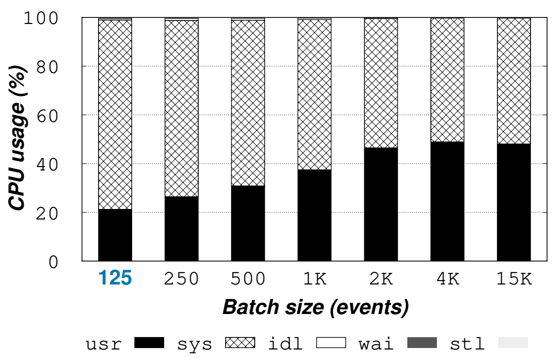
Takeaway 5. *With a larger ring buffer, DIO collects more events from kernel in user space. However, it increases performance overhead and memory usage.*

4.5.2.2 Elasticsearch's Batch Size Impact

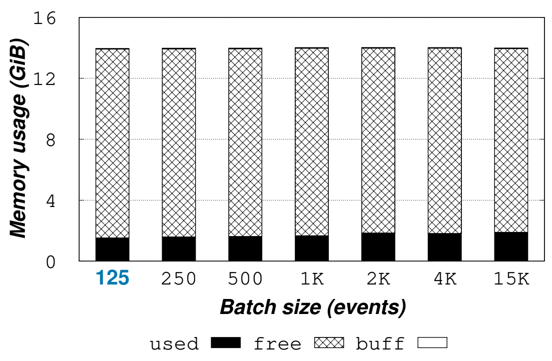
To complement the conclusions from *Takeaways 2* and *4*, we next assess the impact of using different batches sizes for transmitting traced data to Elasticsearch. Fig. 4.20a shows the throughput and number of collected events for Sysdig when Logstash sends batches of 125, 250, 500, 1K, 2K, 4K, and 15K events to Elasticsearch.



(a) Throughput and collected events.



(b) CPU usage.

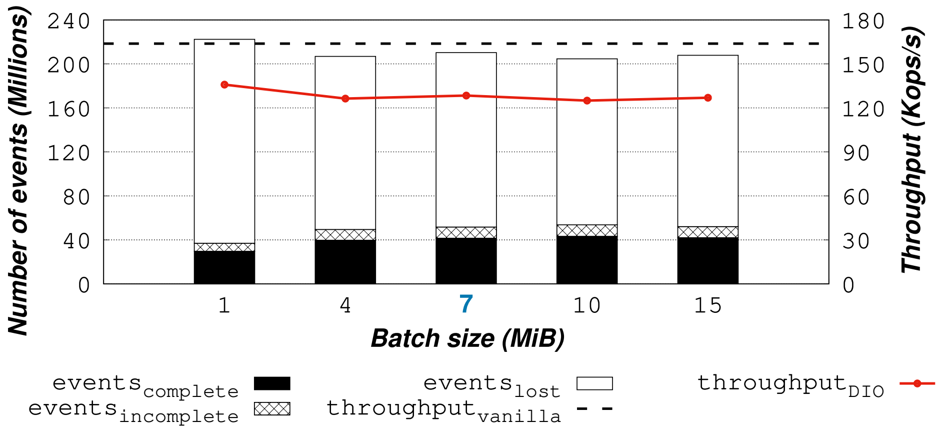


(c) Memory usage.

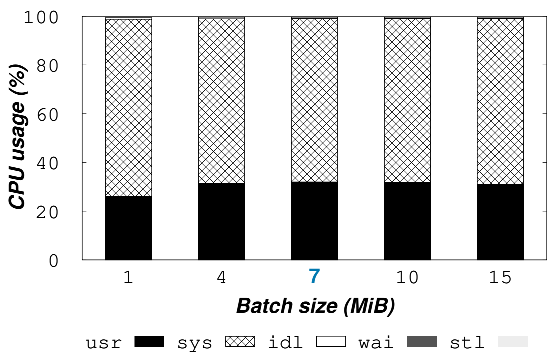
Figure 4.20. Performance overhead, collected events, and resource usage for different batch sizes in Sysdig. The blue color pinpoints the default configuration.

The default configuration used in our experiments (125 events) imposes the least overhead (12%) but saves less information (≈2M events). Increasing the batch size results in higher performance overhead, while the number of collected events increases only for sizes inferior to 1K. For the latter, Sysdig collects more events (≈6.3M) and imposes a performance overhead of 31%. For larger batches, the overhead tends to stabilize around 40%, but Sysdig starts collecting less information (*e.g.*, with a batch size of 15K, Sysdig collects 4.9M events). As depicted in Fig. 4.20b, CPU *usr* time increases with larger batches. The *used* memory (shown in Fig. 4.20c) varies between 1.6 GiB and 1.9 GiB.

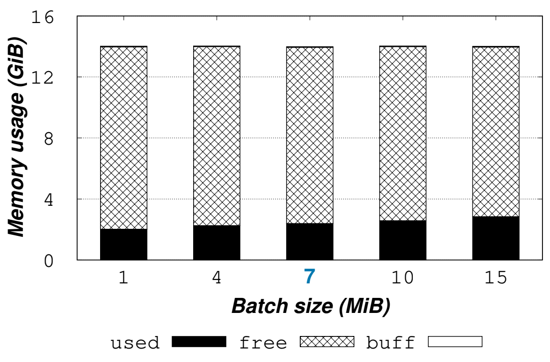
In DIO, due to a different approach in terms of design and implementation (*i.e.*, it does not resort to Logstash, as the *tracer* sends information directly to Elasticsearch to be more efficient), batch sizes must be configured in MiB instead of the number of events. Fig. 4.21a shows the throughput and number of collected events for DIO when configured with batches of 1, 4, 7, 10, and 15 MiB.



(a) Throughput and collected events.



(b) CPU usage.



(c) Memory usage.

Figure 4.21. Performance overhead, collected events, and resource usage for different batches sizes in DIO. The blue color pinpoints the default configuration.

Like in Sysdig, with larger sizes, DIO collects more events and imposes higher performance overhead. However, the variance across different size configurations is small, always capturing more than 37M events and imposing no more than 24% of overhead. In detail, a batch size of 1MiB imposes the smallest overhead (17%) but collects less information (37M events). On the other hand, a size of 10MiB allows collecting more events (54M) but imposes the highest overhead (24%).

Contrarily to Sysdig, increasing the batch size in DIO has a negligible effect on CPU usage (Fig. 4.21b), but increases *used* memory, going from 2.0GiB (batch size of 1 MiB) to 2.8GiB (batch size of 15MiB). Therefore, our experiments consider a default batch size of 7MiB for DIO.

Takeaway 6. *In Sysdig, increasing the batch size has a bigger effect in the balance between performance overhead and events captured than in DIO.*

Takeaway 7. *When considering different batch size configurations, DIO is able to capture from 6x to 27x more events than Sysdig. Also, the performance overhead in DIO is always kept below 23%, while in Sysdig, it increases up to 41% for larger batches.*

4.5.2.3 Inline vs. Offline analysis

According to the results from §4.5.1 and *Takeaway 2*, both Sysdig and DIO save more tracing information when writing data to a local disk. However, such an option requires users to, later on, parse and forward this information from disk to Elasticsearch in an offline fashion. Next, we evaluate the benefits and drawbacks of following offline and inline approaches.

For inline experiments, Sysdig is configured with a batch size of 1K since it allows collecting more events, and DIO with the default batch size of 7MiB, which offers a good tradeoff between performance overhead, collected events, and resource usage, as observed in §4.5.2.2.

For offline experiments, a custom DIO's trace processor is used to read traced data from disk, parse, and forward it to Elasticsearch. For Sysdig, since data is written in a binary format, we use Sysdig's functionality to read the tracer's binary data and redirect its output to Logstash (via a Unix pipe) for parsing and forwarding data to Elasticsearch. In both tracers, we use a batch size of 15K events (the largest size supported for our experimental setup) to optimize data transmission speed to the *Backend*.

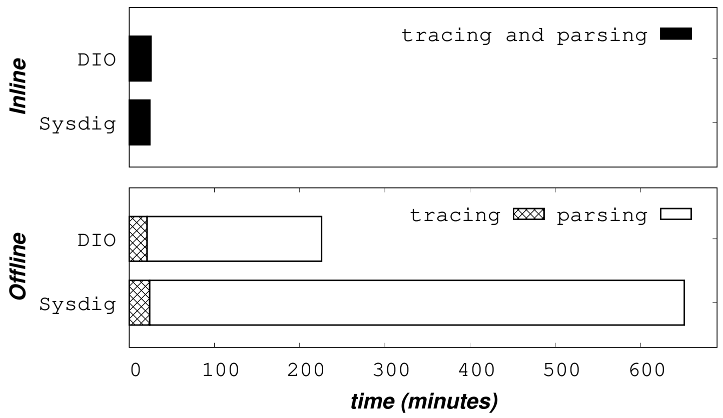


Figure 4.22. Execution times for *inline* and *offline* analysis approaches with Sysdig and DIO.

Fig. 4.22 shows the time spent by each tracer when collecting information (*tracing*), and processing and forwarding it to Elasticsearch (*parsing*). Regarding the inline approach, both tracers present similar execution times (≈24 mins for Sysdig and ≈26 mins for DIO), but Sysdig can only send 6M events to Elasticsearch, while DIO sends up to 52M. When following an offline approach, DIO sends up to 108M events to Elasticsearch, taking 21 mins to collect information (tracing phase) and 205 mins to process and forward it (parsing phase). Sysdig can collect even more information (around 230M events) during ≈24 mins but takes about ≈627 mins (*i.e.*, 10 hours and 27 min) to process and forward all these events to Elasticsearch. By further inspecting these results, we noticed that Sysdig takes only ≈32 mins reading the 230M events saved on disk, thus exposing Logstash as the main reason for the long parsing time.

Takeaway 8. *Inline approaches significantly reduce the time for users to start analyzing collected data, at the cost of discarding syscalls issued by the targeted application.*

Takeaway 9. *When following an offline approach, Sysdig captures more events but takes an impractical amount of time to parse and forward traced data to the backend. Since DIO is implemented and optimized to interact directly with Elasticsearch, it exhibits better performance.*

4.5.3 DIO's Adaptability to Different I/O Rates

In practice, data-centric applications access storage resources with different I/O rates. Fig. 4.23 shows DIO's performance overhead and collected events when Filebench is configured to generate I/O operations at specific rates (or inferior if the system cannot handle these), starting at 25 Kops/s.

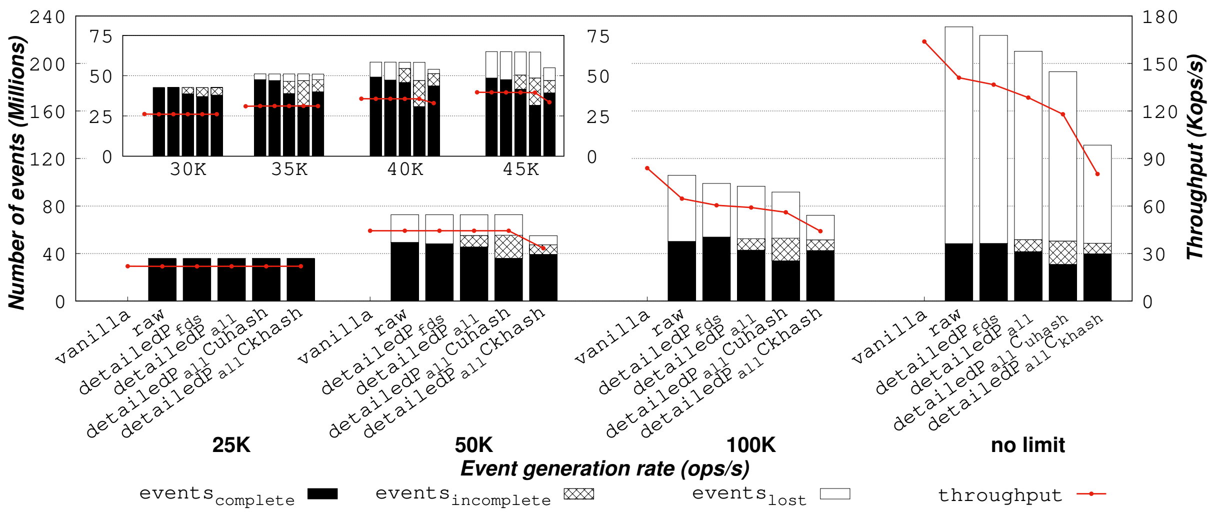


Figure 4.23. Performance overhead and collected events of DIO's setups (with Elasticsearch backend) when tracing Filebench with different I/O rates.

When Filebench issues operations at a rate ≤25 Kops/s, all setups collect the full information from issued syscalls (36M events). With a rate of 30 Kops/s, the setups capturing more detailed information (*detailedP*$_{all}$, *detailedP*$_{all}$*C*$_{uhash}$, and *detailedP*$_{all}$*C*$_{khash}$) save some incomplete events (4M to 6M). *Lost* events happen for rates ≥35 Kops/s (3M to 5M), but DIO only starts impacting Filebench's performance for rates over 40 Kops/s and mostly with *detailed$P_{all}C_{khash}$*. The difference between vanilla and DIO's setups is more noticeable for rates ≥100 Kops/s, where performance overhead ranges from 15% to 51%, and the number of *lost* events varies from 21M to 182M syscalls.

Takeaway 10. *The performance overhead and number of events collected by DIO changes according to the applications' I/O rate. Under 25 Kops/s, all setups collect the full set of issued operations, while most setups only have a noticeable performance impact over Filebench when surpassing 100 Kops/s.*

4.5.4 DIO's Filters Impact

As discussed in §4.2.2, capturing only events of interest helps users reduce the overhead imposed on the target application and the volume of data to analyze. To evaluate the impact of filtering events at the tracing phase, we now compare the default configuration of DIO (*detailedP_{all}* mode with the Elasticsearch storage backend, which does not apply any filters) with four new setups:

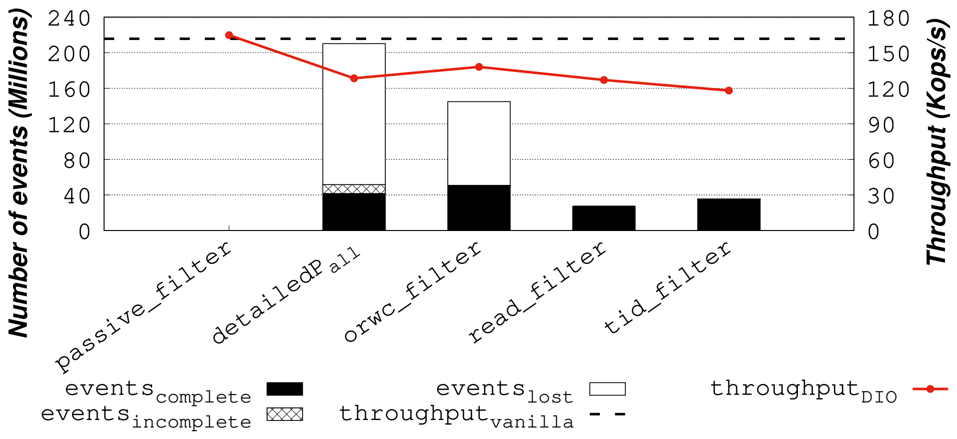
- *passive_filter* - captures only the `rename` syscall type, which is never invoked by Filebench.
- *orwc_filter* - captures a subset of syscalls (*i.e.*, `open`, `read`, `write`, and `close`).
- *read_filter* - captures only `read` syscalls.
- *tid_filter* - captures all syscalls made by a specific thread of Filebench.

The *passive_filter* evaluates the impact of having an active tracepoint that is never triggered by the targeted application, while *orwc_filter* and *read_filter* assess the impact of activating more or less tracepoints. Finally, the *tid_filter* evaluates the impact of filtering events of interest in kernel space.

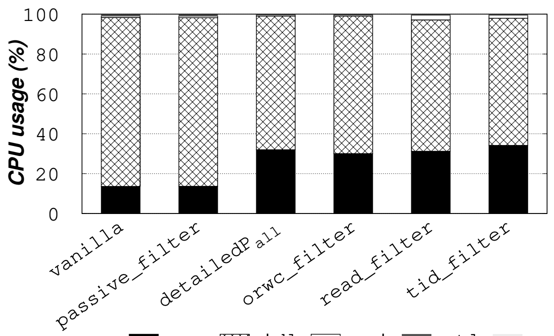
Collected Events. Fig. 4.24a shows the number of collected events and performance overhead for each setup. When capturing all events with *detailedP_{all}* (*i.e.*, without any filters), DIO intercepts 210M syscalls, saving 42M *complete* and 10M *incomplete* events while losing 158M events. The *orwc_filter* setup reduces the events of interest to 145M, which allows saving more *complete* events (≈51M) and reducing the *incomplete* and *lost* events to 0 and 94M, respectively. By capturing only *read* syscalls, the *read_filter* setup further reduces intercepted events to 27M, being able to save them all along with their complete information (*i.e.*, no *incomplete* or *lost* events). Similarly, by filtering events from a specific TID in kernel space, the *tid_filter* setup intercepts and saves 35M events of interest.

Performance Impact. As shown in Fig. 4.24a, DIO does not impose extra overhead if an active probe is never triggered (*passive_filter*), achieving a performance throughput similar to vanilla (165 Kops/s). When activating all supported tracepoints (*detailedP_{all}*), DIO introduces an overhead of 22%. By filtering events by a specific subset of syscalls (*orwc_filter*), DIO reduces the number of active tracepoints and therefore decreases the overhead to 16%.

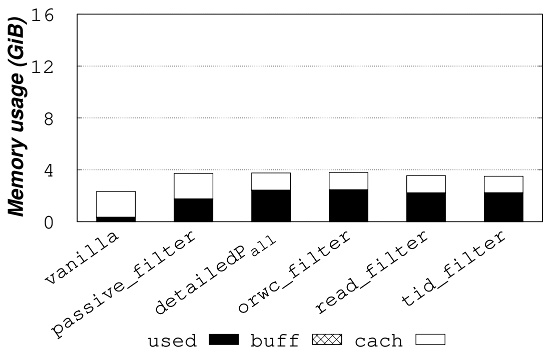
However, the *read_filter* setup, which only activates one tracepoint, imposes similar overhead as in *detailedP_{all}*. These results are explained by the number of *eventPath* events (used to map file descriptors



(a) Throughput and syscall stats.



(b) CPU usage.



(c) Memory usage.

Figure 4.24. Performance overhead and collected events of DIO (with *detailedP$_{all}$* setup and Elasticsearch storage backend) when applying different filters.

to file paths, as described in §4.3.1.3) saved by each setup. Namely, the *detailedP$_{all}$* and *orwc_filter* setups can only save between 122K to 358K *eventPaths*. On the other hand, the *read_filter* setup saves all generated *eventPaths* (≈7M), which requires copying more data from kernel to user space, imposing a higher performance overhead. The same is valid for *tid_filter*, which increases overhead to 28% because it also saves a large number of *eventPaths* (≈5M).

Resource Usage. Figs. 4.24b and 4.24c show CPU and memory usage results. The *passive_filter* setup presents similar CPU consumption as in vanilla since it does not intercept any syscall. The other setups increase *usr* time by ≈16%. As for memory consumption, all DIO's setups present similar results, which is explained by the static allocation of memory done by our system (*e.g.*, for the *ring buffer*). Regarding storage usage, by reducing the events of interest, the *read_filter* and *tid_filter* setups minimize the *detailedP$_{all}$*'s index size (12GiB) by 42% (7GiB) and 50% (6GiB), respectively.

Takeaway 11. *DIO's filters enable users to target only events of interest, which allows reducing storage overhead, improving the number of collected events, and, depending on the filter type and I/O workload, reducing performance overhead.*

4.5.5 Summary

To sum up, the results and takeaways discussed in this section show that the approach followed by DIO is key to building an integrated tracing and analysis pipeline that can offer a good tradeoff regarding performance overhead, tracing accuracy, and timely analysis for users.

First, by relying on the eBPF technology, DIO can intercept applications' syscalls without modifying their source code while minimizing tracing performance overhead.

Second, the flexibility offered by DIO's different tracing modes allows balancing the tracing accuracy with the performance and storage overheads by configuring the detail of the information being captured. Likewise, DIO's filtering capabilities enable discarding fewer I/O events of interest while reducing the storage capacity needed at the *Backend*. Specifically, the storage overhead imposed by DIO varies according to the tracing mode, filters used, and the number of events generated by the application (*e.g.*, 86 MiB for the Redis use case with ≈600K events collected with the *detailed*P_{all} mode, 90 GiB for the RocksDB use case with more than 500M events collected with the *raw* mode).

Moreover, by following an inline approach, DIO reduces the need to store traced data locally and enables users to analyze and visualize collected data in near real-time, which is not possible when following an offline design. Further, our custom design integrating the *Tracer* directly with the *Backend* component exhibits significantly better accuracy (*i.e.*, in terms of the amount of collected data at the *Backend*) than by combining different state-of-the-art tools (*i.e.*, Sysdig and Logstash).

In practice, and as shown in §4.4, data-centric applications such as RocksDB, Elasticsearch, and Redis do not fully stress the underlying storage resources and, by leveraging DIO's customized, flexible, and integrated design, users can capture the full set of syscalls or, at least, have a negligible number of events discarded that do not compromise the diagnosis of such applications.

4.6 Related Work

Table 4.4 shows a comparison between DIO and related solutions in terms of: captured tracing information, filtering capabilities, tracing and analysis integration (O-offline, I-inline), analysis customization, and predefined visualization support. While some tools are able to trace (T) the information required for the chapter's use-cases, only DIO provides users with the analysis (A) capabilities to diagnose them.

I/O Tracing. Storage I/O diagnosis is often done by capturing applications' requests in user space through source code instrumentation [62, 67, 117, 132]; through middleware libraries [81, 108] that are restricted to specific sets of applications (*e.g.*, *LD_PRELOAD* only works with dynamic libraries); or at lower kernel layers [64, 81, 102], such as the VFS, where optimizations like I/O merging make it impossible to observe the exact requests submitted by applications.

To intercept I/O operations non-intrusively and closer to the requests made by applications, other solutions rely on the syscall interface. As shown in Table 4.4, these explore distinct tracing technologies, including `ptrace` ([98, 109]), eBPF ([42, 111, 116]), LTTng ([5, 32, 68]), and auditd ([126]), which allow

Table 4.4. Comparison between DIO and related solutions regarding: *i)* tracing and (O-offline, I-inline) analysis functionalities, and *ii)* support for tracing (T) and analyzing (A) the use cases from §4.1 and §4.4.

		Strace[109]	Sysdig[111]	Re-Animator[5]	RepTrace[98]	Tracee[116]	CAT[42]	Daoud and Dagenais [32]	Kohyarnejadfard et al. [68]	LongLine[126]	DIO
Tracing	syscall info	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	f_offset	-	-	-	-	-	-	-	-	-	✓
	f_type	-	✓	-	-	-	-	-	-	-	✓
	proc_name	✓*	✓	-	-	✓	✓	-	-	✓	✓
	filters	✓	✓	-	-	✓	✓	-	-	-	✓
Analysis pipeline	integrated	-	-	-	O	-	O	O	O	I	I
	customizable	-	-	-	-	-	-	✓	✓	-	✓
	predefined visualizations	-	-	-	-	-	✓	✓	✓	✓	✓
Use cases	§4.1	-	-	-	-	-	-	-	-	-	TA
	§4.4.1	-	T	-	-	-	-	-	-	-	TA
	§4.4.2	-	-	-	-	-	-	-	-	-	TA
	§4.4.3	T*	T	-	-	T	T	-	-	T	TA

* Only supported for versions 5.15 or later.

gathering information related with the *entry* and *exit* points of syscalls, including their arguments, return value, timestamps, PIDs, *etc*. Similar to DIO, some tools enrich traced data with additional information such as the *process name* ([42, 111, 116, 126]), which is useful for observing the I/O patterns at §4.4.2, and §4.4.3. However, DIO is the only tool that collects *file offsets*, which are crucial for diagnosing the use case presented in §4.4.2.

Only CaT (Chapter 3), Tracee [116], and DIO aggregate the information contained at the *entry* and *exit* points of each syscall into a single event, thus simplifying its posterior analysis. This is done at kernel space to reduce the data transferred to user space. Further, these are the only tools, along with *strace* [109] and *Sysdig* [111], that support filtering at the tracing phase.

Integrated Analysis Pipeline. Several solutions only cover the tracing step, leaving the integration with analysis pipelines to be done by users [5, 109, 111, 116]. Other tools provide modules for automating the analysis of traced data but follow an offline approach, where this data needs to be stored first and, only later, it is parsed and provided as input to the analysis pipeline [32, 42, 68, 98]. Only DIO and LongLine [126] automatically parse and forward traced events to the analysis pipeline by following an inline (near-real-time) approach.

Syscall Analysis and Visualization. Some of the existing tools support analysis modules specialized for their concrete use cases (*e.g.*, causality [42, 98], security analysis [126]), which only consider specific information collected from traces (*e.g.*, syscall types). Therefore, these do not provide the flexibility to implement custom analysis algorithms nor enable users to access and explore other information contained in the collected I/O traces. On the other hand, solutions similar to DIO that support customizable analysis fail to capture relevant information to diagnose the use cases discussed in this chapter [32, 68].

DIO provides users access to the complete set of captured information (*e.g.*, syscall type, arguments, offsets), allowing them to build new algorithms over the data fields that are more relevant to their analysis goals. Moreover, DIO offers predefined representations that automatically summarize and allow the visualization of the I/O patterns discussed in this chapter. Further, our tool enables users to create new visualizations commonly supported by other diagnosis solutions (*e.g.*, tables, pie charts, histograms, heatmaps, time series) [32, 68, 126].

To sum up, DIO is the first solution providing an integrated inline diagnosis pipeline that is designed to be flexible and customizable, while covering a larger set of information from syscalls than other state-of-the-art solutions.

4.7 Summary and Discussion

This chapter presents DIO, a generic tool for observing and diagnosing I/O interactions between applications and in-kernel POSIX storage systems. Through a pipeline that automates the process of tracing, filtering, correlating, and visualizing millions of syscalls and by enriching the information provided by these with additional context, DIO helps users observe I/O issues while reducing the search space for finding their root cause when, for instance, source code inspection is required.

Our experiments with widely used systems show that DIO provides key information for exploring I/O requests, observing inefficient or erroneous I/O access patterns that lead to performance degradation or data loss, and identifying resource contention in multithreaded I/O that leads to high tail latency. Further, a detailed evaluation comparing DIO with state-of-the-art tracers shows that our integrated pipeline enables users to diagnose applications in a more timely fashion while providing the best balance in terms of performance overhead and tracing accuracy.

In the next chapter, we show how one can leverage and extend DIO's solution (*e.g.*, to trace network I/O requests, build custom correlation algorithms and visualizations) for a different use case, namely, the characterization of ransomware's I/O behavior. This new use case further validates the usefulness of DIO and the advantages of providing a general-purpose solution that is modular, flexible and customizable.

5 Comprehensive Analysis of Cryptographic Ransomware's I/O Behavior

Cryptographic ransomware is one of the most well-known and damaging types of malware, which acts by encrypting data at infected servers and then demanding a ransom in exchange for the cryptographic key necessary to decrypt compromised data to its original format [20, 88]. This malicious software is now spread across distinct OSs other than Windows, such as Android and Linux. As the latter OS is typically used by large institutions (*i.e.*, governments, companies) holding critical and private information, ransomware attacks on their distributed infrastructures can have devastating effects, as observed for the Colonial Pipeline, Quanta Computer, and Konica Minolta attacks in 2021 [94, 95, 120].

Current ransomware detection and prevention tools are mostly based on classification and ML algorithms that can differentiate between malign and benign applications through key features, such as unique I/O patterns exhibited by ransomware attacks (*e.g.*, targeted files, API call patterns) [27, 57, 105]. These features are distilled from information collected statically from binary inspection or dynamically from observing the I/O interaction of ransomware samples (*i.e.*, binaries) with the OS. The latter is necessary for samples using concealment techniques (*i.e.*, obfuscation, polymorphism, encryption) that make binary inspection inefficient [88, 99].

However, given the sheer amount of cryptographic ransomware families, which are constantly evolving to bypass detection mechanisms, it becomes challenging to understand what I/O information needs to be collected dynamically at runtime and used as a feature for classification purposes. For this, security analysts and engineers require deep knowledge of how ransomware families work and evolve over time.

Frameworks, such as Cuckoo [47] and Limon [71], provide sandbox environments to run ransomware samples and output to users summaries of their suspicious activities. Since these summaries are limited in terms of analysis scope, these frameworks also output detailed execution logs (traces), which are crucial for an in-depth exploration of ransomware's I/O patterns. However, we argue that this is a sub-optimal approach as it leaves to users the inspection of large traces (*i.e.*, containing thousands to millions of I/O events), which could be automated with proper analysis and visualization tools.

Therefore, while focusing on the Linux OS, the main insight of this chapter is that to further comprehend the I/O behavior of ransomware, one should leverage all the information collected during its execution and automate its analysis and visualization. Namely, by exploring the syscalls used by a ransomware sample, their arguments, and their contextual information (*e.g.*, process name, PID, TID), one would be able to obtain detailed information about the sample. For example, it would be possible to *i)* observe how many processes/threads are created and learn about their specialization (*e.g.*, encrypt files, write ransom notes); *ii)* understand how the infected file system is transversed and which files are being targeted; *iii)* learn about specific I/O patterns done over infected files (*e.g.*, encryption key generation and persistence, file's extension renaming, targeted file offsets).

While this information is useful for analysts to better understand ransomware attacks, compare distinct families and observe their I/O patterns' evolution, to implement such an idea, one must address the following challenges, as discussed in §2.4:

Challenge C1. Most of the available ransomware samples are packed binaries whose source code is undisclosed. Thus, the collection of relevant information (*tracing*) should be *dynamic* (*i.e.*, done along with the sample's execution) and *non-intrusive* (*i.e.*, without requiring modifications to its source code).

Challenge C3. To enable a comprehensive exploration and analysis of ransomware's I/O behavior, collected traces must include diverse and detailed information regarding the sample's interaction with the OS (*e.g.*, process creation, syscall types, arguments, and context). Such information can also be complemented with other system metrics (*e.g.*, CPU usage).

Challenge A1 and V1. The analysis pipeline should be integrated with the data collection phase while efficiently handling large volumes of traced data (*i.e.*, thousands to millions of I/O events), allowing its storage, processing, and visualization.

Challenge A2, A3, V2 and V3. Manually exploring traced data and finding the most appropriate queries to observe specific I/O patterns is a complex and time-consuming task, which could be automated with tailored correlation algorithms and visualizations. Further, to efficiently compare the I/O behavior of different ransomware families, one must store and index the data traced for each sample, correlate such information, and create adequate visualizations that help pinpoint their main similarities and differences.

The previous challenges are addressed with CRIBA, a tool for diagnosing the I/O behavior of Linux cryptographic ransomware. Briefly, CRIBA builds upon DIO to support dynamic collection of information about I/O syscalls issued by ransomware samples along with other system metrics. Also, it leverages DIO's integrated analysis and visualization pipeline, while extending it to build custom correlation algorithms and combining these with new tailored visual representations that enable comprehensive insights about the I/O patterns of ransomware attacks. In more detail, this work provides the following contributions:

Diagnosis Pipeline Tailored for Ransomware. An open-source tool integrating the collection, analysis, and visualization of execution traces from ransomware. CRIBA allows the collection and analysis of comprehensive information about I/O syscalls (*e.g.*, type, arguments), their contextual information (*e.g.*, PID, offset), and relation with other system metrics (*e.g.*, CPU).

Custom Correlation Algorithms. CRIBA extends DIO by providing automated analysis capabilities, through 6 new algorithms, that ease the study and comparison of ransomware samples by pinpointing their file system transversal, file access, and file extension manipulation patterns.

Custom Visualizations. CRIBA also provides a new set of visualizations, organized into 8 distinct dashboards, for summarizing and exploring collected information as well as the outputs of the correlation algorithms in a human-readable and explainable fashion.

Ransomware Study. We provide a comprehensive analysis and comparison of 5 Linux ransomware families that shows CRIBA's capabilities.

The conducted experimental study shows that CRIBA automates the analysis and observation of generic behavior from ransomware samples (*e.g.,* the number of processes, type of syscalls, file system transversal). Further, it enables the analysis and comparison of intrinsic and complex I/O behavior (*e.g.*, file access patterns, extension manipulation) related to the creation of ransom notes, file encryption, and evasion techniques used by each family.

All artifacts discussed in this chapter, including CRIBA, datasets, scripts, and the corresponding analysis and visualization outputs, are publicly available at `https://github.com/dsrhaslab/criba`.

5.1 Ransomware Overview

We now overview the workflow of cryptographic ransomware while highlighting some of its unique I/O features.

Main Phases of Ransomware Attacks. Cryptographic ransomware typically acts in four phases [85].

First, the attacker exploits system vulnerabilities (*e.g.*, kernel bugs) or uses social engineering techniques (*e.g.*, phishing emails) to install a malicious sample at the victim's machine(s) (*Infection* phase). Once installed and running, the sample establishes a connection with its Command and Control (C&C) server to retrieve necessary information for data encryption (*e.g.*, encryption keys) and/or exfiltrate information about the infected system (*e.g.*, hostname, hardware info) to the attacker (*Communication with C&C servers* phase). Then, the ransomware transverses the files at the infected server(s) and encrypts their data, blocking access to these (*Destruction* phase). In the end, the ransomware leaves a ransom note informing the victim about the attack and disclosing payment instructions (*Extortion* phase).

Data Encryption. Cryptographic ransomware typically follows a hybrid approach combining symmetric and asymmetric encryption schemes [85]. It starts by locally creating a symmetric key, usually a different key per infected file. It then reads the file's content and encrypts it with the generated key. Symmetric key encryption schemes impose lower CPU load and have faster encryption times than asymmetric ones [16].

To prevent the victim from discovering symmetric keys and recovering the original files' content, the ransomware encrypts these with the attacker's public key obtained, for instance, during the *Communication with C&C servers* phase. The encrypted file's data and corresponding encrypted symmetric key are both written to the targeted file, which is usually renamed to include a new extension (*e.g.*, .ecrypt in Erebus). Thus, to recover the original files, the victim first needs to obtain the attacker's private key and then use it to decrypt the symmetric keys needed for the files' data decryption.

Detection Features. The ransomware actions to encrypt the victim's files result in intensive I/O patterns with several characteristics that deviate from the normal behavior of benign applications:

- *Directory search* - typically, all directories at the infected machine are transversed in search for files to encrypt [105].
- *Files access* - encryption usually requires rewriting the whole file's data in a short time window [57].
- *Number of storage operations* - encrypting several files results in a significant amount of storage I/O operations, such as opening, reading, writing, and closing each file.
- *Unknown file extensions* - by changing the extension of encrypted files, ransomware samples execute an abnormal number of rename operations. Moreover, this results in the appearance of new and unknown file extensions [105].
- *CPU usage* - by encrypting files, the sample imposes a high CPU load on the victim's machine [16].
- *Network communication* - the communication with the C&C usually translates into network operations targeting unknown network domains [27].

These are some of the features that detection tools use to identify the malicious activity of ransomware.

Evasion Techniques. Some ransomware families use evasion techniques to retard or avoid being detected. For instance, many families include the public encryption key within the binary to avoid communication with the C&C [16]. Other families reduce CPU load and encryption time, and hide I/O patterns, by encrypting only a subset of files on the infected machine. File selection can be based on *i)* its extension (*e.g.*, work-related documents such as *.pdf*, *.docx*, *.txt*; VM-related files like *.vmdk*, *.vmem*, *.vswp*); *ii)* its size (*e.g.*, larger files will most probably contain important data); or *iii)* arbitrary. Some families also limit the number of bytes to encrypt in each file, which is usually sufficient to avoid recovering their full content.

With CRIBA, we aim to assist security analysts in exploring and analyzing the I/O behavior of cryptographic ransomware samples to understand better how they operate, identify and refine key features for their detection, and learn about their techniques used to evade detection tools.

5.2 CRIBA in a Nutshell

Fig. 5.1 depicts CRIBA's architecture, which is built on top of DIO as it already provides modules for *i)* collecting information about applications' syscalls without requiring access or modification to their source code; and *ii)* storing, analyzing and visualizing the collected information.

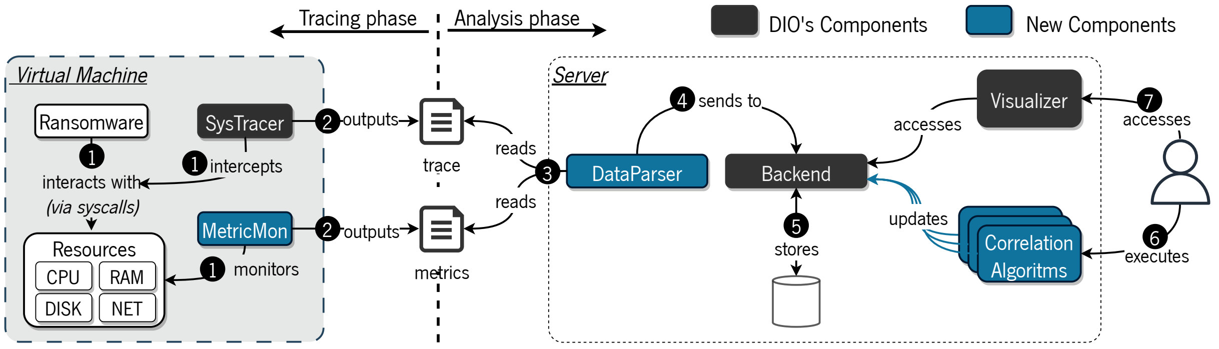


Figure 5.1. CRIBA's design and flow of events for the tracing and analysis phases.

5.2.1 System Workflow

CRIBA's components are executed in two separate phases: *i)* the tracing phase, where information from the ransomware execution is collected; and *ii)* the analysis phase, where collected data is analyzed and visualized. In the tracing phase, the ransomware sample is executed in a controlled environment along with *SysTracer* that intercepts its I/O syscalls, and *MetricsMon* that monitors system statistics (❶). When the ransomware finishes its execution, these components' output files are extracted from the controlled environment (❷) to initiate the analysis process. In the analysis phase, the *DataParser* is used to read the tracing output files (❸) and forward these to the *Backend* (❹). The latter persists and indexes collected data (❺) and provides access to it through a querying API. Meanwhile, users can execute the provided *correlation algorithms* (❻) and access the *Visualizer* dashboards (❼) to visually explore the output of these and other information contained at the indexed data.

5.2.2 Architectural Components

Although DIO's analysis pipeline is useful for observing the I/O behavior of applications, it still requires users to spend significant time exploring and manually building/customizing queries and visualizations to analyze collected traces from ransomware samples.

CRIBA extends DIO to collect more information at the tracing phase, including network operations and system's resources metrics, and to automate the analysis process by providing a set of correlation algorithms and predefined visualizations tailored for the exploration of cryptographic ransomware. Next, we highlight the modifications made to DIO's original design and describe the new components added (blue boxes at Fig. 5.1).

SysTracer. DIO's *tracer* is focused on the interception of storage-related sycalls. To consider also network-related requests, we modified this module to intercept 13 more syscalls (*e.g.*, `connect`, `accept`, `send`, `receive`). The full set of supported syscalls is shown in Table 5.1.

MetricMon. To obtain information about the system's resource usage, we introduced a new module for collecting statistics, including CPU, memory, and disk usage.

DataParser. Ransomware samples must run in a controlled environment (*e.g.*, isolated VM) to avoid infecting the experimental servers. Thus, *SysTracer* and *MetricMon* save collected information to disk (trace files) instead of sending it directly to the analysis pipeline. The *DataParser* module is responsible for parsing these trace files and forwarding their information to the analysis pipeline.

Backend and Correlation Algorithms. The *Backend* component, which may be deployed on separate server(s), is identical to the one offered in DIO and provides the functionalities of storage and exploration of collected information. However, to ease the analysis of cryptographic ransomware, we developed several new correlation algorithms that query the *Backend*, analyze and correlate queried data, and send the analysis results back to the *Backend*. As further explained in §5.3.1 and demonstrated in §5.4, these algorithms provide relevant information to understand how ransomware behaves and to find interesting and distinctive I/O patterns. Further, our design is extensible, enabling users to develop other correlation algorithms that may suit their analysis goals.

Visualizer. The *Visualizer* component, which is also based on DIO, was extended to include visual representations tailored for observing ransomware's analysis findings (*e.g.*, for the output of correlation algorithms) thus, simplifying users' exploration and making it more explainable. Particularly, we built several dashboards that allow observing:

- *Generic Overview* - general statistics about the traced execution (*e.g.*, execution time, number of processes and threads, number and type of syscalls).
- *Directory Transversal* - the type of transversal done by ransomware samples over the dataset.
- *File Name and Extensions* - the file name and extensions accessed by ransomware samples.
- *Syscall Sequences* - the sequence of syscalls done by samples over dataset files.
- *File Ngrams* - bigrams, trigrams, and quadgrams of file accesses done by the ransomware sample.
- *File Offsets* - file offsets accessed by samples.
- *Resource Usage* - metrics about the utilization of resources (*e.g.*, CPU, RAM) at the infected host.
- *Families Comparison* - heatmaps comparing multiple ransomware families regarding their issued syscalls, accessed file extensions and names.

Each dashboard is mentioned explicitly in §5.4 while discussing the analysis findings contained in it. Also, our design is extensible as users can create new visualizations based on their analysis needs.

Table 5.1. List of syscalls supported by CRIBA (and corresponding tags used in §5.3.1).

Tag	Syscall	Tag	Syscall
AC	`accept, accept4`	**RD**	`read, pread64, readv`
BD	`bind`	**RH**	`readahead`
CL	`close`	**RL**	`readlink, readlinkat`
CN	`connect`	**RN**	`rename, renameat, renameat2`
CR	`creat`	**RX**	`removexattr, lremovexattr, fremovexattr`
FS	`fsync, fdatasync`	**SD**	`sendto, sendmsg`
GS	`getsockopt`	**SK**	`socket, socketpair`
GX	`getxattr, lgetxattr, fgetxattr`	**SS**	`setsockopt`
LS	`lseek`	**ST**	`stat, lstat, fstat, fstatfs, fstatat`
LT	`listen`	**SX**	`setxattr, lsetxattr, fsetxattr`
LX	`listxattr, llistxattr, flistxattr`	**TR**	`truncate, ftruncate`
MK	`mknod, mknodat`	**UN**	`unlink, unlinkat`
OP	`open, openat`	**WR**	`write, pwrite64, writev`
RC	`recvfrom, recvmsg`		

5.3 Algorithms and Prototype

CRIBA's open-source prototype includes six correlation algorithms that aim to ease the analysis process and provide more detailed insights on how ransomware behaves. We next explain each correlation algorithm in detail (§5.3.1) and describe the implementation detais of CRIBA's prototype (§5.3.2).

5.3.1 Correlation Algorithms

The correlation algorithms provided by CRIBA include three main phases: *i)* the querying phase, where one or more queries are performed to the *Backend* to obtain the required data; *ii)* the correlation phase, where the acquired data is processed and correlated to obtain addition information; and *iii)* the updating phase, where the correlation results are sent back to the *Backend* to be more easily accessed and visualized by users at the *Visualizer* component.

UNExt. The *UNExt* algorithm extracts the file name and file extension from the file paths targeted by traced syscall events. Specifically, for every event accessing[1] a file path (*e.g.*, `read`, `write`, `stat`), the algorithm splits the full path (*e.g.*, `/files/example.txt`) to obtain the file name (*e.g.*, `example.txt`) and the file extension (*e.g.*, `.txt`). This algorithm is implemented as a search and update query that is fully executed at the *Backend*. With its output, users can explore, at the *Visualizer*, the file names and extensions accessed by ransomware samples.

DsetU. The *DsetU* algorithm compares the list of file paths accessed by the ransomware sample with the full list of file paths contained in a given dataset collection, provided as input by the user (*e.g.*, the experimental dataset described in §5.4). The output sent to the *Backend* includes which dataset's files

[1] In the chapter, file access means at least one syscall is done over the file.

and extensions were accessed by the ransomware. This output can then be explored with the *Visualizer* to uncover samples targeting specific files and extensions.

Transversals. The *Transversals* algorithm determines the order in which ransomware threads transverse the file system, namely Breadth First Search (BFS), Depth First Search (DFS), or unknown. Alg. 5.1 shows the algorithm for identifying DFS. By analyzing opened files, it builds a file tree (L2) and obtains an order – $dfsOrder$ – in which the files would have been visited with DFS (L3). Then it correlates the actual file opening order done by the thread with $dfsOrder$ (L4-L9). Next, it builds a SegmentTree structure [53] (L10) to efficiently verify if external files were visited while transversing a given subtree (L11-L16), meaning that the thread is not doing DFS. Note that the algorithm tolerates different access orders to files on the same directory (*e.g.*, alphabetical, creation time, *etc.*). The search type done by each thread and the information containing the opened files and folders are sent to the *Backend* to be explored at the *Visualizer*.

Algorithm 5.1: CRIBA's algorithm for identifying a Depth First Search.

```
Input: Files opened over time ($fpaths$)
Output: True if DFS was observed
1  Function isDFS($fpaths$) is
2      $tree \leftarrow$ buildFileTree($fpaths$)
3      $dfsOrder \leftarrow$ transverseDFS($tree$)
4      $list \leftarrow []$
5      $i \leftarrow 0$
6      for $file \in fpaths$ do
7          $idx \leftarrow dfsOrder[file]$
8          $list[idx] = i$
9          $i += 1$
10     $st \leftarrow$ buildSegmentTree($list$)
11     for $file \in fpaths$ do
12         $a \leftarrow st$.getMaxElemInsideSubtree()
13         $b \leftarrow st$.getMinElemOusideSubtree()
14         $st$.remove($file$)
15         if $a > b$ then
16             return $false$
17     return $true$
```

FnGram. The *FnGram* algorithm computes the n-grams collocations for the files accessed by a ransomware sample over time. Specifically, it queries the *Backend* to obtain a list of the file paths accessed by each thread. Then, it computes and sends back to the *Backend* the *bigrams*, *trigrams*, and *quadgrams* for that input list. With this information, it is possible to depict dependencies between file accesses (*e.g.*, the trigram (`A.txt`, `B.txt`, `C.txt`) shows that `B.txt` was accessed after `A.txt` and before `C.txt`).

FSysSeq. The *FSysSeq* algorithm computes the sequence of consecutive unique syscalls done by ransomware threads to all files. First, for each file, the algorithm queries the *Backend* to obtain the list of syscalls (sorted by time). As depicted in Alg. 5.2, each syscall in the list is translated to a tag (L5) of two letters (according to Table 5.1) to reduce the length of the final sequence, while subsequent syscalls of the same type are reduced to a single one (L6-L8). For example, the list [`open`, `read`, `read`, `lseek`, `write`, `write`, `close`, `rename`] is reduced to "`OP`→`RD`→`LS`→`WR`→`CL`→`RN`". With these sequences, one is able to learn how ransomware threads access files and observe similar patterns between files (*e.g.*, identical sequences for ransom notes as shown in §5.4.2).

Algorithm 5.2: FSysSeq correlation algorithm provided by CRIBA.

```
  Input: Syscalls for a given file over time ($syscalls$)
  Output: Syscall sequence ($sequence$)
1 Function ComputeSysSeq($syscalls$) is
2     $sequence \leftarrow []$
3     $prev\_tag \leftarrow$ NULL
4     for $s \leftarrow syscalls$ do
5         $tag \leftarrow$ get_tag($s$)
6         if $prev\_tag = NULL$ or $prev\_tag \neq tag$ then
7             $sequence$.append($tag$)
8         $prev\_tag \leftarrow tag$
9     return $sequence$
```

TfidfFam. The *tfidfFam* algorithm eases the comparison between distinct ransomware families. Specifically, for each family, the algorithm requests from the *Backend* all the values observed for a given category (*e.g.*, syscall type). These values are passed as input to the Term Frequency-Inverse Document Frequency (TF-IDF) algorithm. The latter is a feature selection technique, often used by ransomware detection tools [25, 26, 130], that outputs a numerical statistic showing the relevance of each value (*e.g.*, relevance of `read` syscalls for the Erebus sample). Then, the cosine similarity is applied to the output of TF-IDF to have a single metric of comparison between families according to their most relevant values. By sending back to the *Backend* the TF-IDF output, users can observe the most relevant values per family and for a given category. By sending the cosine similarity results, users are able to understand how similar/distinct the ransomware families are. We apply this algorithm to three categories: syscall type, file paths, and file extensions.

5.3.2 Implementation

SysTracer extends DIO's tracer with 7 new eBPF programs, written in restricted C, that are attached to 13 *entry* and 13 *exit* tracepoints for intercepting network-related syscalls. The *Backend*, *Visualizer* and *MetricMon* components are provided by instances of Elasticsearch [8], Kibana [9], and Metricbeat [11], respectively. The *DataParser* and the *correlation algorithms* are implemented in ≈2K lines of Python code and interact directly with the Elasticsearch instance (*i.e.*, index, update, and query). The dashboards with predefined visualizations are provided along with CRIBA and include representations (*e.g.*, Fig. 5.5) developed using the Vega-Lite grammar.

5.4 CRIBA in Action

Our experimental evaluation shows how CRIBA automates and eases the work for users when: *i)* exploring and understanding both general and specific behaviors exhibited by ransomware samples; and *ii)* comparing different families to find common and distinct patterns across them.

Ransomware Families. The experiments consider 5 Linux ransomware families, which were chosen based on their popularity and distinct traits.

- Erebus emerged in 2016 and is known for infecting thousands of computers and servers. A notorious example is the attack on the Linux infrastructure of a South Korean web hosting company in 2017 [40].
- REvil is a ransomware family discovered in 2019 that reached its peak activity in 2021. It targeted both widely known public figures and companies like Quanta Computer, a supplier of Apple [95].
- RansomEXX is a recent ransomware targeting Linux infrastructures. The Texas Department of Transportation, Konica Minolta, and Scottish Mental Health Charity were attacked by this malware between 2020 and 2022 [94].
- Darkside emerged in 2020 and was used to launch a global campaign infecting targets in 15 countries and multiple industry sectors [33]. It is known for, in 2021, targeting the Colonial Pipeline, a company responsible for half of the fuel supply of the US East Coast [120].
- AvosLocker released a Linux variant in 2021. This family has been targeting critical infrastructures in countries such as the US, Canada, and UK [93].

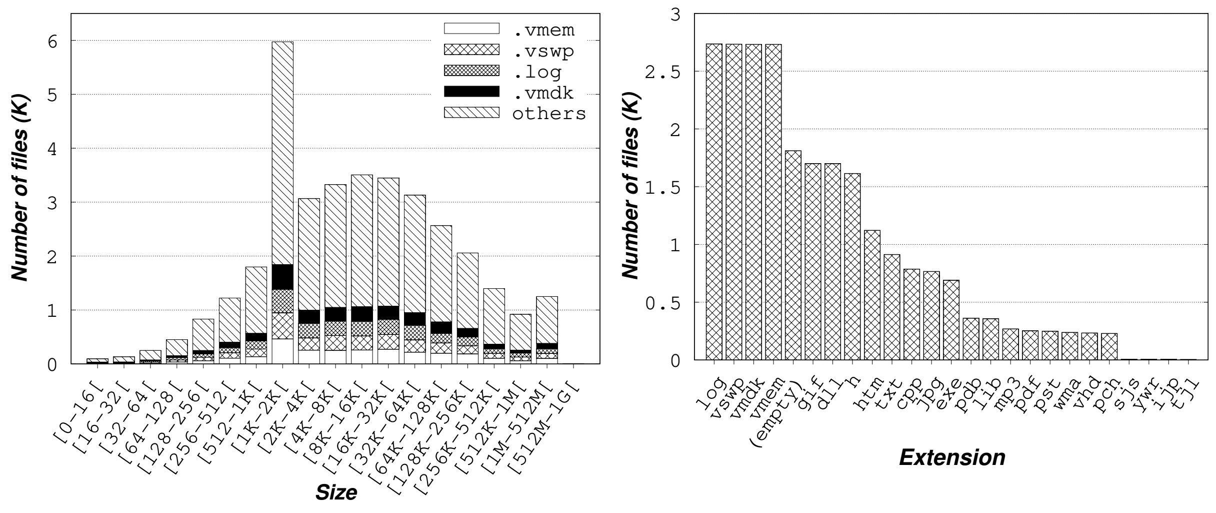
As some ransomware samples require defining the number of encryption threads and the targeted file system directory path, we configured all samples to use 1 encryption thread and to target the dataset discussed next. For Erebus, which does not allow specifying these two arguments, and for Darkside, which does not allow changing the number of threads, we used their default configurations. Table 5.2 shows the SHA256 hashes and execution commands used for each sample.

File Dataset. As in previous work [17, 80], the *Impressions* framework [2] was used to generate a synthetic dataset exhibiting a statistically accurate file system image with realistic metadata and content. The dataset, with 9.4 GiB, includes 35,418 files, with sizes ranging from 0 B to 800 MiB. Files are spread across 3,510 directories with an average tree depth of 12 levels.

Table 5.2. SHA256 hashes and execution commands for the 5 ransomware samples analyzed with CRIBA.

Family	*SHA256*	*Command*
AvosLocker	d7112a1e1c68c366c05bbede9dbe782b b434231f84e5a72a724cc8345d8d9d13	`./avos.elf 1 /app/files`
RansomEXX	08113ca015468d6c29af4e4e4754c003 dacc194ce4a254e15f38060854f18867	`./ransomexx.elf --threads 1 --path /app/files`
REvil	3d375d0ead2b63168de86ca2649360d9 dcff75b3e0ffa2cf1e50816ec92b3b7d	`./revil.elf --path /app/files --threads 1`
Erebus	0b7996bca486575be15e68dba7cbd802 b1e5f90436ba23f802da66292c8a055f	`./erebus.elf`
Darkside	c93e6237abf041bc2530ccb510dd016e f1cc6847d43bf023351dce2a96fdc33b	`./darkside.elf --path /app/files`

Darkside encrypts only specific file extensions (*.vmem*, *.vswp*, *.log* and *.vmdk*) that are not generated with the *Impressions* framework. Thus, we developed a script to change the file names of some dataset files, considering both small and large-sized files, to include these. The final dataset used in the experiments has 8,267 unique file extensions. Fig. 5.2 shows the distribution of the files' sizes and extensions for the dataset.



(a) Files' sizes.

(b) Files' extensions.

Figure 5.2. Distribution of files' size and extensions for the dataset used in CRIBA's experiments.

Testbed Configuration. Our testbed includes two environments. The ransomware samples and CRIBA's *SysTracer* and *MetricsMon* are executed in a controlled environment. Namely, they run inside a VM configured with 2 GiB of RAM, 2 CPU cores, and a disk partition of 64 GiB. The VM is deployed on a server equipped with an 8-core Intel i9-9880H, 16 GiB of memory, and a 500 GiB SSD NVMe. The host OS runs *macOS Big Sur 11.7.6* while the guest OS runs *Ubuntu 22.04 LTS* with kernel *5.15.0*. The VM image is reverted to a previous (and clean) snapshot every time a ransomware sample is executed. CRIBA's *Backend* and *Visualizer* components, as well as the correlation algorithms, run at the analysis environment, which consists of a separate server equipped with a 6-core Intel i5-9500, 16 GiB of memory and a 250 GiB NVMe SSD, and running *Ubuntu 20.04 LTS* with kernel *5.4.0*.

The execution time for the full analysis workflow, including the tracing, preprocessing, and loading of traced data (using CRIBA's *DataParser*), and execution of all correlation algorithms, took, on average, ≈19 mins per family.

5.4.1 General Statistics

Tables 5.3 and 5.4 show general statistics provided by CRIBA for the 5 ransomware families considered in our experiments.

Table 5.3. Execution time, process creation, accessed files and issued syscalls statistics for the ransomware families.

Family	*Execution*	*Process*		*Accesses*			*Syscalls*			
	time (min)	*PIDs*	*TIDs*	*Paths*	*Extensions*	*Transversal*	*Types*	*Events*	*Data / Metadata*	*Storage / Network*
AvosLocker	1.481	1	2	11 646	3 044	DFS	8	134 985	34.13% / 65.87%	100% / 0%
RansomEXX	3.126	1	5	85 583	19 341	DFS	9	703 575	31.99% / 68.01%	100% / 0%
REvil	8.719	12	13	39 384	8 275	DFS	9	774 007	41.83% / 58.17%	100% / 0%
Erebus	10.361	3	12	107 307	8 482	-	17	1 257 238	26.86% / 73.14%	99.96% / 0.04%
Darkside	0.386	1	6	11 244	12	DFS	19	21 070	25.06% / 74.94%	99.79% / 0.21%

Table 5.4. Top 3 syscall types issued per ransomware family.

Syscall	*Family*				
	AvosLocker	RansomEXX	REvil	Erebus	Darkside
#1	`lseek` (23.942%)	`lseek` (20.280%)	`read` (28.385%)	`read` (19.013%)	`stat` (53.711%)
#2	`read` (20.589%)	`write` (16.340%)	`lseek` (20.084%)	`stat` (15.871%)	`read` (10.332%)
#3	`fstat` (16.755%)	`read` (15.648%)	`close` (14.226%)	`openat` (14.671%)	`writev` (10.190%)

Observation 1. Families exhibit significantly different execution times, with Darkside running in less than 1 min and Erebus taking more than 10 mins.

Observation 2. Most families use a single process, except for Erebus (3 processes) and REvil (12 processes). The number of threads ranges from 2 in AvosLocker to 13 in REvil.

Observation 3. AvosLocker and Darkside access fewer unique file system paths and file extensions than the other samples. Erebus has the highest number of file accesses, while RansomEXX accesses the highest number of distinct file extensions. When considering only accesses to the dataset's files, AvosLocker and Darkside only access ≈30% of these, while RansomEXX accesses almost 93%. REvil and Erebus are the only samples accessing all files in the dataset.

Observation 4. Except for Erebus, all families perform a DFS to transverse the dataset's directory tree. This search is done by 2 threads in AvosLocker, REvil and Darkside, and 5 threads in RansomEXX. Also, all samples access system directories besides the dataset's ones (*e.g.*, `/usr`, `/proc`, `/dev`).

Observation 5. Darkside uses a wider range of different syscall types (*e.g.*, `stat`, `read`, `writev`) but performs fewer operations in total than the other families. AvosLocker issues fewer types of syscalls, while Erebus performs more than 1M I/O operations.

Observation 6. Most of issued syscalls are metadata-related. Namely, `lseek`, `stat`, and `fstat` are widely used by all families. In Darkside, half of the total issued I/O requests correspond to `stat` syscalls.

Observation 7. Erebus and Darkside are the only samples issuing network-related syscalls (e.g., `connect` or `recvfrom`). This indicates that these samples may be the only ones communicating with C&C servers.

Observation 8. For all families, the distribution of I/O load (*i.e.*, amount of requests) per thread varies. For instance, when considering the 13 threads created by REvil, only two do syscalls throughout the whole execution. Moreover, as depicted in Fig. 5.3, most I/O requests are done by a single thread (TID 1814 issues 98.027% of the syscalls, while TID 1777 executes only 1.822%). The remaining threads perform fewer I/O requests and only at the beginning of the execution (*e.g.*, TID 1809 does 0.132% of the syscalls).

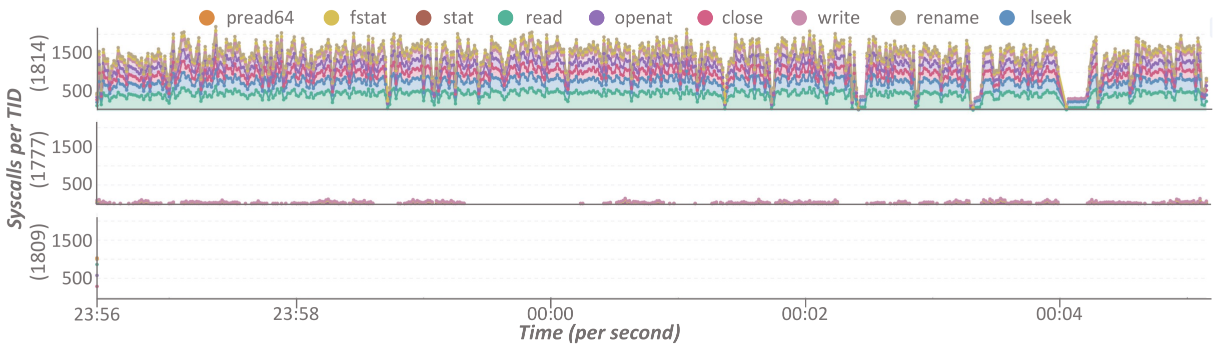


Figure 5.3. Aggregated number of operations, per syscall, for three distinct threads launched by REvil.

Takeaways. *General statistics show that ransomware families exhibit different patterns in terms of execution time, process and thread creation, and accessed files and extensions. Metadata-related storage operations are the most predominant type of issued syscall, while network-related calls are only issued by a few families. Most families transverse the file system in a DFS fashion.*

CRIBA's role. *The aforementioned statistics were automatically computed by CRIBA, with the UN-Ext, DsetU and Transversals algorithms, and explored through its Generic Overview and Directory Transversal dashboards.*

5.4.2 Ransom Notes

Table 5.5 shows the sequences of syscalls done for ransom notes by each ransomware family.

Observation 9. While the file name used for ransom notes changes across families, each sample reuses the same name for all written notes, most of which use the `.txt` extension. In AvosLocker, no extension is used, while Erebus creates an additional note with an `.html` extension.

Table 5.5. Syscall sequences for ransom notes per family.

Family	*Syscall sequence*
AvosLocker	*README_FOR_RESTORE*: `OP→ST→WR→CL`
RansomEXX	*!NEWS_FOR_STJ!.txt*: `ST→OP→ST→WR→CL`
REvil	*qoxaq-readme.txt*: `OP→ST→WR→CL`
Erebus	*_DECRYPT_FILE.html*: `OP→WR→CL→RN→OP→WR→CL` *_DECRYPT_FILE.txt*: `OP→WR→CL`
Darkside	*darkside_readme.txt*: *(1):* `ST` *(2):* `ST→OP→WR→CL` *(3):* `ST→OP→WR→CL→ST`

Observation 10. Darkside creates 274 ransom notes, while RansomEXX, REvil and Erebus create more than 3,500. Erebus creates more ransom notes with the `.html` extension (8,430) than with the `.txt` one (4,000).

Observation 11. While Darkside delegates the creation of ransom notes to two separate threads, the others use only one.

Observation 12. Most families exhibit a unique sequence of syscalls for ransom notes. Contrarily, Darkside does 3 different sequences, which is caused by having multiple threads creating these. Namely, each thread starts by performing a `stat` syscall to verify if the targeted directory already contains a note, writing a new one only if this is not the case.

Observation 13. All families perform `open`, `write` and `close` syscalls over each ransom note file. `Stat` is also significantly used (except for Erebus), but its amount and placement, in the sequence of syscalls issued per ransom note, varies (*i.e.*, AvosLocker and REvil only perform `stat` before opening the file, while RansomEXX performs it before and after opening the file).

Observation 14. The syscall sequence done for Erebus's `.html` ransom notes includes a rename (`RN`) operation. By further analyzing the arguments of the syscalls issued to these files, one observes that Erebus first creates and writes a file named `_DECRYPT_FILE.html`. Later, it renames it to `index.html`, and creates another `_DECRYPT_FILE.html` file, in the same folder, with the same content.

Takeaways. *The 5 ransomware families share similarities regarding the creation of ransom notes, such as reusing the same name for files placed across different directories and using almost the same set of syscalls. The observations also show distinct patterns, such as the number of ransom notes created by each family, the use of two distinct ransom notes by Erebus, and the different syscall sequences performed by Darkside.*

CRIBA's role. *The previous findings were obtained through CRIBA's UNext, FSysSeq, FnGrams, and Transversals algorithms, and by exploring their output with the File Name and Extensions, Syscall Sequences, File Ngrams and Directory Transversal dashboards.*

5.4.3 Dataset's Files Access and Encryption

Table 5.6 shows syscall sequences done over a small (*F31277.jsd.vswp*) and large (*F10573.bqt.vmdk*) dataset file.

Table 5.6. Syscall sequences issued per family over a small and large dataset file.

Family	*Syscall sequence*	
	F31277.jsd.vswp (≈217 KiB)	*F10573.bqt.vmdk* (≈32 MiB)
AvosLocker	OP→ST→LS→RD→ LS→RD→LS→WR→CL→RN	OP→ST→LS→RD→(LS→RD→LS→WR)$_{x4}$→CL→RN
RansomEXX	OP→ST→LS→RD→WR→LS→RD→LS→WR→CL→RN	OP→ST→LS→RD→WR→LS→RD→LS→WR→CL→RN
REvil	OP→LS→ST→RD→LS→ WR→LS→RD→WR→CL→RN	OP→LS→ST→RD→LS→WR→(LS→RD→LS→WR)$_{x32}$→ LS→RD→WR→CL→RN
Erebus	ST	*original file*: ST→RN *renamed file*: ST→OP→RD→LS→RD→WR→LS→WR→ (LS→RD)$_{x3}$→LS→WR→(RD→WR)$_{x64}$→CL
Darkside	ST	ST→OP→(RD→WR)$_{x32}$→CL→OP→LS→WR→CL→RN

Observation 15. The number of unique sequences of syscalls, considering all dataset files, varies from 7 (REvil) to 28 (RansomEXX). These differ depending on specific file characteristics (*e.g.*, file size). For instance, RansomEXX uses the same sequence for the small and the large file examples, while the other families use different sequences. For AvosLocker and REvil, the difference is mostly on the amount of `lseek`, `read`, and `write` operations done to each file.

Observation 16. By inspecting the TID associated with each sequence, one can conclude that AvosLocker, REvil, and Erebus use 1 thread for accessing files, while Darkside uses 2 and RansomEXX uses 4. Further, except for Erebus and Darkside, the creation of ransom notes and the access to dataset files are delegated to distinct threads.

Observation 17. Some of the observed sequences only include the `stat` syscall. This suggests that some files are not being processed (*e.g.*, encrypted). In fact, through CRIBA, we can conclude that Erebus accesses all dataset's files but only processes 33.82% of these. Also, from the 30% of dataset's files accessed by Darkside, only 3.48% are being processed.

Observation 18. Erebus is the only sample accessing and encrypting files from all directories of the file system, which explains the high number of accessed files (*Observation 3*).

Observation 19. When inspecting the offset of `lseek` and `write` requests for other sequences, one can observe interesting patterns associated with the writing of encryption keys to infected files. Namely, RansomEXX starts by jumping to the end of the infected file (ST→LS→RD), writing the key (WR), and then jumping back to the beginning of the file to initiate the encryption process (LS). AvosLocker, REvil, and Darkside only write the key after encrypting the file's content. The latter reopens the file, jumps to its end, and then writes the key (OP→LS→WR→CL). Erebus does not exhibit the previous file offset access patterns, as it writes the encryption key at the beginning of files before encrypting their content [40].

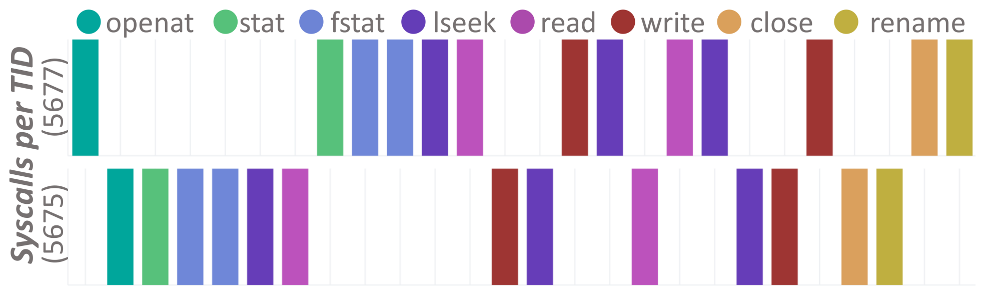


Figure 5.4. Syscalls issued over time by RansomEXX's encryption threads to file F10573.bqt.vmdk.

Observation 20. REvil and Erebus always read content from the `/dev/urandom` file before, or in between, consecutive accesses to each dataset file. The n-grams (`/dev/urandom`, …`/F3737.vva.vmem`, `/dev/urandom`, …`/F10573.bqt.vmdk`) from REvil and (…`/F10573.bqt.vmdk`, `/dev/urandom`, …`/F10573.bqt.vmdk`) from Erebus show these two patterns. Darkside also reads from the `/dev/urandom` file multiple times. These accesses are probably due to the generation of randomness for creating symmetric encryption keys.

Observation 21. Most of the ransomware families operate directly over the original dataset file and, after encrypting it, rename the file to add their own extension (note the `RN` operation at the end of most sequences). However, Erebus starts by first renaming the original file to a random name (*e.g.*, `F10573.bqt.vmdk` → `CA2065AE397D85C1.ecrypt`) and only then encrypts its content.

Observation 22. While most families add a constant file extension to encrypted files (*e.g.*, `.avoslinux` for AvosLocker, `.ecrypt` for Erebus), RansomEXX generates a different extension for each file, composed of a constant prefix (`.stj888-`) concatenated with a random suffix. Interestingly, some files have two distinct extensions (*e.g.*, the large file has the extensions: `.stj888-36acf3f1` and `.stj888-40aa97db`). By observing in more detail the syscalls issued for the large file by thread, as depicted in Fig. 5.4, it is possible to see two threads simultaneously opening, reading, writing, and renaming the same file. This concurrent pattern can lead to data corruption and irrecoverable files.

Takeaways. *The sequences of syscalls change according to the targeted files and the ransomware family. Different patterns are also observed regarding the timing and placement of encryption keys at infected files and for the extensions chosen by each family when renaming encrypted files. Interestingly, REvil, Erebus, and Darkside use `/dev/urandom` to generate randomness. In RansomEXX, two threads are concurrently encrypting the same file, a pattern that may lead to corrupted files.*

CRIBA's role. *Results were obtained with the UNext, DsetU, FSysSeq, FnGram and Transversals algorithms, and observed with the Syscall Sequences, File Ngrams, File Offsets and Directory Transversal dashboards.*

5.4.4 Dataset's Files Selection and Evasion Techniques

Fig. 5.5 shows the offsets accessed when reading and writing to the large file (*i.e.*, `F10573.bqt.vmdk`).

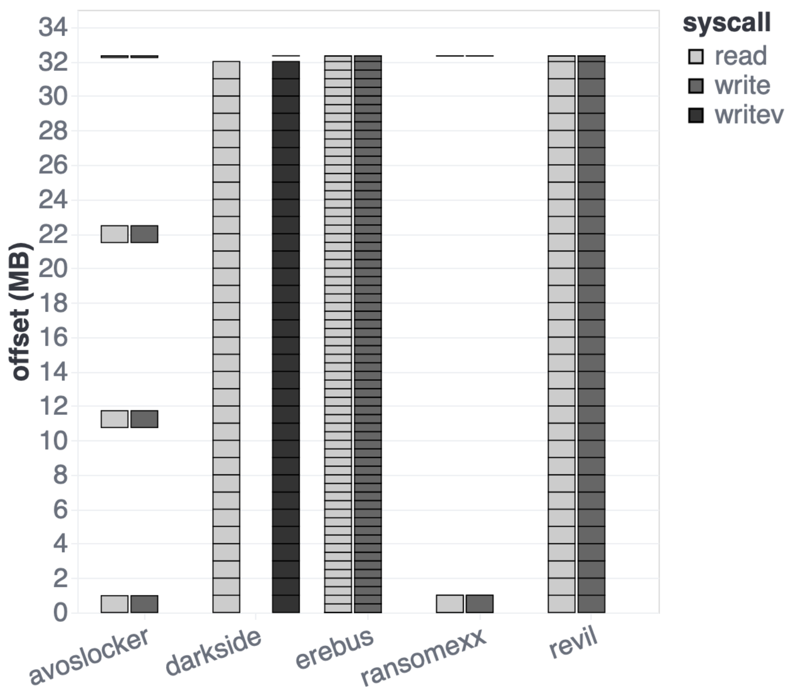


Figure 5.5. File offsets accessed per family when reading and writing file F10573.bqt.vmdk.

Observation 23. REvil and Erebus read and overwrite the full file's content (*i.e.*, offsets are fully accessed across the whole file ≈32.33 MiB). Darkside encrypts most of the file's content but leaves the last (incomplete) block in plaintext. Also, REvil and Darkside use blocks of 1 MiB to read and write the file, while Erebus uses blocks of 512 KiB.

Observation 24. For every 10.78 MiB of content belonging to a given file, AvosLocker only encrypts 0.98 MiB. For the `F10573.bqt.vmdk` file, RansomEXX only encrypts the first 1 MiB block. However, CRIBA shows that the latter behavior changes across files and, for other files, this sample sparsely encrypts multiple 1 MiB blocks across the entire file.

Observation 25. *Observations 3 and 17* show that, except for REvil, all other families avoid encrypting the full dataset. When correlating the files accessed by Darkside with the dataset's file sizes and extensions, we observe that it only processes 4 types of extensions (*i.e.*, `.vmem`, `.vswp`, `.log` and `.vmdk`) and that only considers files with a size larger than 1 MiB, amounting to 381 files.

Observation 26. As depicted in Fig. 5.6, Darkside and RansomEXX have the highest bursts of CPU usage (94% and 86%, respectively), but for a short time, given the multiple threads doing data encryption. REvil exhibits a CPU usage of ≈55%, for a larger time period, given its single thread encrypting the full dataset's content.

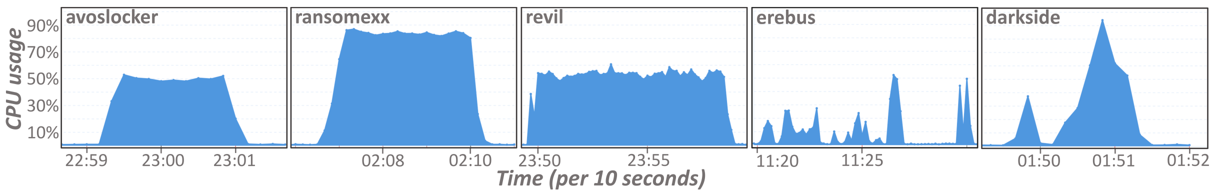


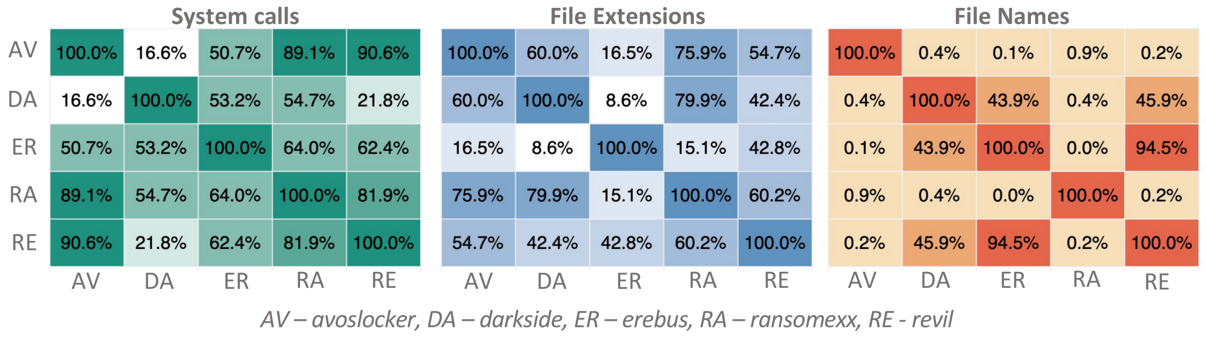
Figure 5.6. CPU usage per ransomware family.

Takeaways. *Only REvil accesses all dataset files and overwrites their full content. The other families target specific file extensions (e.g., Darkside) and/or do not process the full content of files (e.g., AvosLocker). These patterns enable faster execution and lower CPU usage and are used to deceive detection tools (see §5.1).*

CRIBA's role. *DsetU and UNExt algorithms, along with the File Offsets, Directory Transversals, and Resource Usage dashboards were used for the previous observations.*

5.4.5 Families Similarity and Summary

Fig. 5.7 highlights the similarity across families for the type of syscalls done and the file extensions and names accessed. These observations were obtained through CRIBA's *TfidfFam* algorithm and the *Families Comparison* dashboard.



System calls

	AV	DA	ER	RA	RE
AV	100.0%	16.6%	50.7%	89.1%	90.6%
DA	16.6%	100.0%	53.2%	54.7%	21.8%
ER	50.7%	53.2%	100.0%	64.0%	62.4%
RA	89.1%	54.7%	64.0%	100.0%	81.9%
RE	90.6%	21.8%	62.4%	81.9%	100.0%

File Extensions

	AV	DA	ER	RA	RE
AV	100.0%	60.0%	16.5%	75.9%	54.7%
DA	60.0%	100.0%	8.6%	79.9%	42.4%
ER	16.5%	8.6%	100.0%	15.1%	42.8%
RA	75.9%	79.9%	15.1%	100.0%	60.2%
RE	54.7%	42.4%	42.8%	60.2%	100.0%

File Names

	AV	DA	ER	RA	RE
AV	100.0%	0.4%	0.1%	0.9%	0.2%
DA	0.4%	100.0%	43.9%	0.4%	45.9%
ER	0.1%	43.9%	100.0%	0.0%	94.5%
RA	0.9%	0.4%	0.0%	100.0%	0.2%
RE	0.2%	45.9%	94.5%	0.2%	100.0%

AV – avoslocker, DA – darkside, ER – erebus, RA – ransomexx, RE - revil

Figure 5.7. Heatmaps comparing the families regarding the type of issued syscalls, and accessed file extensions and names.

For the type of syscalls, Darkside is the most unique sample, sharing less than 22% of similarity with AvosLocker and REvil, and less than 55% with Erebus and RansomEXX. This is due to Darkside using more types of syscalls, including network calls (*Observations 5 and 7*).

When looking at accessed file extensions, Erebus is the sample with the highest deviation, being only 42.8% similar to REvil. This is explained by its distinctive behavior of encrypting files only after adding the *.ecrypt* extension (*Observation 21*), and by the number of accesses to `/dev/urandom`, which does not have an extension (*Observation 20*).

As for file names, the families are very dissimilar, with only Erebus, REvil, and Darkside sharing similarities (94.5% between the first two and up to 45.9% with the latter). The similarity between these 3 families results from their accesses to */dev/urandom* (*Observation 20*).

The results discussed in this section show that CRIBA provides comprehensive analysis and comparison of cryptographic ransomware samples. Also, it shows that ransomware analyses must consider different features to provide a clear understanding of the samples' intrinsic and complex behaviors.

5.5 Related Work

This section discusses and compares existing work on ransomware analysis with CRIBA.

Behavior Analysis Sandboxes. Some solutions provide a controlled environment for running malware samples and extracting their behavioral information [14, 47, 71]. These tools monitor the memory state, network traffic, and API calls done by samples and generate a report highlighting their main (and suspicious) activities. Raw logs are also outputted so that users can further analyze specific features not included in the report.

Leaving the parsing, analysis, and visualization of information contained in raw logs for users is a complex and time-consuming endeavor. Thus, CRIBA can complement these tools with the mechanisms for automating the aforementioned tasks. For instance, security analysts can use tools like Cuckoo Sandbox [47] to perform a first analysis of multiple malware and benign applications. Based on their reports, they can select the samples that need a more in-depth analysis and use our tool to ease such a process.

Ransomware Detection Tools. Solutions focused on ransomware detection also rely on dynamic analysis for understanding the key features that identify ransomware samples exhibiting malicious activity [7, 25, 27, 57, 65, 104, 105, 113]. These tools typically resort to the aforementioned analysis sandboxes to collect information from the samples' execution and then use feature selection techniques to extract the most relevant features for malware detection [6, 66].

As discussed in VizMal [34], since the purpose of detection tools is to classify applications as malign or benign, they do not provide further information for exploring and understanding the internal behavior of ransomware samples. VizMal introduces a new visualization to highlight potential malicious behavior at specific portions of the execution traces of Android ransomware samples. Nonetheless, as VizMal is limited to a single visualization, it cannot provide comprehensive information about ransomware's I/O behavior.

CRIBA is not a tool for ransomware detection, instead, its main goal is to provide an integrated tracing and analysis pipeline optimized for collecting, exploring, and visualizing a vast amount of I/O information about ransomware samples. With this information and the aid of custom correlation algorithms and visualizations, CRIBA automates the observation of interesting I/O patterns for ransom note creation (§5.4.2), data encryption (§5.4.3) and evasion techniques (§5.4.4) used by ransomware samples. Also, it enables the comparison of different samples from either the same or different families while pinpointing their main similarities and differences (§5.4.5).

5.6 Summary and Discussion

We present CRIBA, a tool for simplifying and automating the exploration, analysis, and comparison of I/O patterns for Linux cryptographic ransomware. CRIBA supports non-intrusive and comprehensive collection of I/O information from ransomware samples and combines it with an integrated analysis and visualization pipeline. The latter is enhanced with 6 custom correlation algorithms and different predefined dashboards. As shown in our experimental study, these features are key to: *i)* automate the analysis of ransomware families; *ii)* understand complex and intrinsic behavior from each sample; *iii)* and pinpoint common and distinct traits across families.

Notably, CRIBA also highlights the usefulness of DIO. By building upon the latter, while including new tailored algorithms and visualizations, it shows that DIO's modular and flexible design is key to efficiently diagnosing a wide range of use cases.

6 Conclusion

I/O diagnosis tools play a key role in the development and maintenance of data-centric applications and storage systems. With these, one can design better solutions, *i.e.*, that are more dependable, secure and efficient. In this thesis, we defend that existing tools can be further improved to: *i)* simplify the amount of manual work done by users; *ii)* provide comprehensive details about I/O requests; *iii)* be adaptable to the different analysis requirements that users may have. To achieve such a goal, we propose three novel solutions that advance the current state of the art on this research topic.

When diagnosing distributed systems, one must consider their multiple and heterogeneous components that may be deployed across different servers. These components communicate with each other through the network and issue requests to local or remote storage systems to persist data. Knowing these I/O interactions is key to better understanding complex distributed systems. This need is addressed by the work described in Chapter 3, where we introduce CaT, a novel framework for collecting and analyzing storage and network I/O requests of distributed systems. Contrary to existing solutions, CaT follows a content-aware approach that allows observing how data flows across components, which we show to be useful for uncovering data corruption and adulteration issues. Further, our solution addresses multiple challenges linked to the diagnosis process. To non-intrusively intercept applications' I/O requests, CaT relies on two low-level tracing technologies (Strace and eBPF) that provide different tradeoffs between tracing accuracy, imposed performance overhead and resource usage. Moreover, it employs summarization techniques to persist digests of requests' content instead of their full data to reduce the amount of storage space required for persisting collected data. Finally, it leverages Falcon [82] for inferring the causality of distributed I/O events while proposing a novel algorithm based on similarity estimation techniques to automatically correlate the content of these events and depict their similarities.

Although CaT advances the state of the art for distributed systems' diagnosis, it is still purpose-specific. When exploring multiple I/O behavior aspects (*e.g.*, correctness, dependability and performance) of data-centric applications, one requires general-purpose and flexible diagnosis tools that support a broader range of analyses and are suitable for detecting multiple issues (*e.g.*, I/O contention, data loss). We address this challenge in Chapter 4 with DIO, a generic tool for observing and diagnosing the I/O interactions between applications and in-kernel POSIX storage systems. DIO further explores eBPF's potential for non-intrusively intercepting applications' I/O requests, and provides a set of tracing functionalities that enable users to narrow or broaden the amount and detail of collected data. Such flexibility allows users to configure our tool according to their diagnosis requirements and better balance the data collection's accuracy, performance overhead and resource usage tradeoffs. Moreover, DIO's diagnosis pipeline includes customizable components for data analysis and visualization, allowing users to explore, query and execute different types of analysis over the same set of collected data. The analysis's outputs are presented through visual representations in near real-time. We showcase the applicability and capabilities of DIO, for debugging, validating and exploring both known and unknown I/O patterns through the diagnosis of four distinct production-level applications.

To further demonstrate how one can leverage and extend DIO to perform specialized I/O diagnosis, in Chapter 5 we introduce CRIBA, a practical framework for studying common and distinct I/O patterns of cryptographic ransomware families. The original pipeline from DIO provides the means to collect and index comprehensive data from ransomware samples' execution. Then, through novel correlation algorithms and visual dashboards, CRIBA automates the analysis and visualization of specific behaviors of cryptographic ransomware, such as the creation of ransom notes, the encryption of files, and the evasion techniques used by these to avoid detection mechanisms. Through a study of the I/O behavior of five cryptographic ransomware families, we show that CRIBA provides insightful details about each sample. Such knowledge can be leveraged by security analysts to better understand ransomware attacks and enhance detection tools to accompany their constant evolution.

With the contributions presented in this thesis, we demonstrate that it is possible to further ease the diagnosis of applications' I/O for users. We believe that comprehensive, flexible and customizable diagnosis pipelines, such as the ones proposed in this work, are key for building applications and systems that are more robust and efficient.

6.1 Future Work

The work done under this thesis opens up several research directions that can be addressed in future work.

Explore new diagnosis scopes. The use cases presented in Chapters 4 and 5 demonstrate DIO's capabilities for uncovering unknown patterns and I/O issues. It would be interesting to extend DIO's scope and explore other applications. For instance, it would be interesting to merge CaT's distributed analysis capabilities (*e.g.,* causal order and content similarity inference) into the near-real-time analysis pipeline from DIO and study the latter's applicability for diagnosing distributed applications and systems.

Leverage traced data for improving ML models. Given the comprehensive data collected by DIO and CRIBA, it would be interesting to provide an automated framework that enables developers to take advantage of this information for designing, training and testing new ML-based detection models.

Taking the ransomware detection goal as example, it would be useful to leverage the knowledge provided by CRIBA to build, test, and enhance new detection models and even compare the accuracy of different models. Further, the same idea could be applied to other fields such as failure detection and performance modeling.

DaaS: Diagnosis-as-a-Service. By building upon the flexible and customizable design of DIO, one could further extend its modularity by creating common APIs that would ease the integration of various technologies and strategies for data collection, analysis, and visualization.

The first step would be to support distinct data collection modules that, as in CaT's design, use specific technologies and capture varied data but are all integrated within the same pipeline. These modules could use different tracing technologies (*e.g.*, Strace, eBPF, LTTng, kernel modules), intercept requests at different levels (*e.g.*, application, middleware, OS), and capture varied information (*e.g.*, network requests, storage requests, resource consumption statistics). To efficiently store, index and query the heterogeneous data (*e.g.,* system metrics, timestamped syscalls and corresponding arguments) collected by the aforementioned modules, the analysis pipeline would require support for multiple storage backends. These multiple backends would also ease the integration of different analysis algorithms (*e.g.*, ML-based, graph-based, solvers), and visualization frameworks (*e.g.*, Kibana, Grafana, ShiViz) in the same pipeline.

A general-purpose service, allowing users to choose the best option(s) for their data collection, analysis, and visualization tasks, would eliminate the need for combining several tools and running the diagnosis process multiple times. However, to avoid overwhelming users with extensive configurations, it would be important to automate the latter process with, for instance, ML-based approaches that would aid in the choice of the best configurations according to the desired purpose(s) of diagnosis.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX, 2016, pp. 265–283. url: `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi`.

[2] Nitin Agrawal, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. "Generating Realistic Impressions for File-System Benchmarking". In: *ACM Transactions on Storage* 5.4 (2009), pp. 1–30. doi: `10.1145/1629080.1629086`.

[3] Waseem Ahmed and Yong Wei Wu. "A Survey on Reliability in Distributed Systems". In: *Journal of Computer and System Sciences* 79.8 (2013), pp. 1243–1255. doi: `10.1016/j.jcss.2013.02.006`.

[4] Rohit Aich. "Efficient Audit Data Collection for Linux". PhD thesis. Stony Brook University, 2021.

[5] Ibrahim Umit Akgun, Geoff Kuenning, and Erez Zadok. "Re-Animator: Versatile High-Fidelity Storage-System Tracing and Replaying". In: *13th ACM International Systems and Storage Conference*. ACM, 2020, pp. 61–74. doi: `10.1145/3383669.3398276`.

[6] P Mohan Anand, PV Sai Charan, and Sandeep K Shukla. "A Comprehensive API Call Analysis for Detecting Windows-Based Ransomware". In: *IEEE International Conference on Cyber Security and Resilience*. IEEE, 2022, pp. 337–344. doi: `10.1109/CSR54599.2022.9850320`.

[7] Niccolò Andronio, Stefano Zanero, and Federico Maggi. "HelDroid: Dissecting and Detecting Mobile Ransomware". In: *International Symposium on Research in Attacks, Intrusions and Defenses*. Vol. 9404. Springer, 2015, pp. 382–404. doi: `10.1007/978-3-319-26362-5_18`.

[8] Elasticsearch B.V. *Elasticsearch: The Heart of the Free and Open Elastic Stack*. Accessed on October, 2023. url: `https://www.elastic.co/elasticsearch/`.

[9] Elasticsearch B.V. *Kibana: Your Window Into the Elastic Stack*. Accessed on October, 2023. url: https://www.elastic.co/kibana/.

[10] Elasticsearch B.V. *Logstash: Centralize, Transform & Stash Your Data*. Accessed on October, 2023. url: https://www.elastic.co/logstash/.

[11] Elasticsearch B.V. *Metricbeat: Lightweight Shipper for Metrics*. Accessed on October, 2023. url: https://www.elastic.co/beats/metricbeat.

[12] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. "SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores". In: *USENIX Annual Technical Conference*. USENIX, 2019, pp. 753–766. url: https://www.usenix.org/conference/atc19/presentation/balmau.

[13] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. "Using Magpie for Request Extraction and Workload Modelling". In: *USENIX Symposium on Operating Systems Design and Implementation*. Vol. 4. 2004, pp. 18–18. url: https://www.usenix.org/legacy/event/osdi04/tech/full_papers/barham/barham_html/.

[14] Brian Baskin. *Noriben Malware Analysis Sandbox*. Accessed on October, 2023. url: https://github.com/Rurik/Noriben.

[15] Mick Bauer. "Paranoid Penguin: An Introduction to Novell AppArmor". In: *Linux Journal* 2006.148 (2006), p. 13. url: https://dl.acm.org/doi/fullHtml/10.5555/1149826.1149839.

[16] Eduardo Berrueta, Daniel Morato, Eduardo Magaña, and Mikel Izal. "A Survey on Detection Techniques for Cryptographic Ransomware". In: *IEEE Access* 7 (2019), pp. 144925–144944. doi: 10.1109/ACCESS.2019.2945839.

[17] Eduardo Berrueta, Daniel Morato, Eduardo Magaña, and Mikel Izal. "Open Repository for the Evaluation of Ransomware Detection Tools". In: *IEEE Access* 8 (2020), pp. 65658–65669. doi: 10.1109/ACCESS.2020.2984187.

[18] Jean Luca Bez, Houjun Tang, Bing Xie, David Williams-Young, Rob Latham, Rob Ross, Sarp Oral, and Suren Byna. "I/O Bottleneck Detection and Tuning: Connecting the Dots using Interactive Log Analysis". In: *2021 IEEE/ACM Sixth International Parallel Data Systems Workshop*. IEEE, 2021, pp. 15–22. doi: 10.1109/PDSW54622.2021.00008.

[19] Ashish Bijlani and Umakishore Ramachandran. "Extension Framework for File Systems in User space". In: *USENIX Annual Technical Conference*. USENIX, 2019, pp. 121–134. url: https://www.usenix.org/conference/atc19/presentation/bijlani.

[20] Ross Brewer. "Ransomware Attacks: Detection, Prevention and Cure". In: *Network Security* 2016.9 (2016), pp. 5–9. doi: 10.1016/S1353-4858(16)30086-1.

[21] Andrei Z. Broder. "On the Resemblance and Containment of Documents". In: *Compression and Complexity of Sequences 1997*. IEEE, 1997, pp. 21–29. doi: `10.1109/SEQUEN.1997.666900`.

[22] E Burgener and J Rydning. "High Data Growth and Modern Applications Drive New Storage Requirements in Digitally Transformed Enterprises". In: *IDC Report: A White Paper, sponsored by Dell Technologies and NVIDIA* (2022). url: `https://www.delltechnologies.com/asset/en-my/products/storage/industry-market/h19267-wp-idc-storage-reqs-digital-enterprise.pdf`.

[23] "Chapter 4 - Source code Analysis and Instrumentation". In: *Embedded Computing for High Performance*. Ed. by João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz. Morgan Kaufmann, 2017, pp. 99–135. isbn: 978-0-12-804189-5. doi: `10.1016/B978-0-12-804189-5.00004-1`.

[24] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. "Pinpoint: Problem Determination in Large, Dynamic Internet Services". In: *2002 International Conference on Dependable Systems and Networks*. IEEE, 2002, pp. 595–604. doi: `10.1109/DSN.2002.1029005`.

[25] Qian Chen and Robert A Bridges. "Automated Behavioral Analysis of Malware: A Case Study of WannaCry Ransomware". In: *16th IEEE International Conference on Machine Learning and Applications*. IEEE, 2017, pp. 454–460. doi: `10.1109/ICMLA.2017.0-119`.

[26] Qian Chen, Sheikh Rabiul Islam, Henry Haswell, and Robert A Bridges. "Automated Ransomware Behavior Analysis: Pattern Extraction and Early Detection". In: *2nd International Conference on Science of Cyber Security*. Springer, 2019, pp. 199–214. doi: `10.1007/978-3-030-34637-9_15`.

[27] Christopher Jun-Wen Chew, Vimal Kumar, Panos Patros, and Robi Malik. "ESCAPADE: Encryption-Type-Ransomware: System Call Based Pattern Detection". In: *14th International Conference on Network and System Security*. Springer, 2020, pp. 388–407. doi: `10.1007/978-3-030-65745-1_23`.

[28] Shigeru Chiba. "Javassist: Java Bytecode Engineering Made Simple". In: *Java Developer's Journal* 9.1 (2004).

[29] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. "Benchmarking Cloud Serving Systems with YCSB". In: *1st ACM Symposium on Cloud Computing*. ACM, 2010, pp. 143–154. doi: `10.1145/1807128.1807152`.

[30] Victor Costan and Srinivas Devadas. "Intel SGX Explained". In: *Cryptology ePrint Archive* (2016). url: `https://eprint.iacr.org/2016/086`.

[31] Marco Dantas, Diogo Leitão, Cláudia Correia, Ricardo Macedo, Weijia Xu, and João Paulo. "MONARCH: Hierarchical Storage Management for Deep Learning Frameworks". In: *2021 IEEE International Conference on Cluster Computing*. IEEE, 2021, pp. 657–663. doi: `10.1109/Cluster48925.2021.00097`.

[32] Houssem Daoud and Michel R Dagenais. "Performance Analysis of Distributed Storage Clusters Based on Kernel and Userspace Traces". In: *Software Practice and Experience* 51.1 (2021), pp. 5–24. doi: `10.1002/spe.2889`.

[33] *DarkSide Ransomware as a Service (RaaS)*. Accessed on October, 2023. url: `https://www.state.gov/darkside-ransomware-as-a-service-raas/`.

[34] Andrea De Lorenzo, Fabio Martinelli, Eric Medvet, Francesco Mercaldo, and Antonella Santone. "Visualizing the Outcome of Dynamic Analysis of Android Malware with VizMal". In: *Journal of Information Security and Applications* 50 (2020), p. 102423. doi: `10.1016/j.jisa.2019.102423`.

[35] Jeffrey Dean and Luiz André Barroso. "The Tail at Scale". In: *Communications of the ACM* 56.2 (2013), pp. 74–80. doi: `10.1145/2408776.2408794`.

[36] Biplob Debnath, Mohiuddin Solaimani, Muhammad Ali Gulzar Gulzar, Nipun Arora, Cristian Lumezanu, Jianwu Xu, Bo Zong, Hui Zhang, Guofei Jiang, and Latifur Khan. "LogLens: A Real-Time Log Analysis System". In: *38th International Conference on Distributed Computing Systems*. IEEE, 2018, pp. 1052–1062. doi: `10.1109/ICDCS.2018.00105`.

[37] Mathieu Desnoyers and Michel R Dagenais. "The LTTng Tracer: A Low Impact Performance and Behavior Monitor for GNU/Linux". In: *Ottawa Linux Symposium*. Vol. 2006. Citeseer, 2006, pp. 209–224. url: `https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=355ac65470c38333b26d55b7c8493d93c419cd2e#page=209`.

[38] Jeff Dileo and Andy Olsen. *eBPF Adventures: Fiddling with the Linux Kernel and Unix Domain Sockets*. Accessed on October, 2023. url: `https://www.nccgroup.com/us/about-us/newsroom-and-events/blog/2019/march/ebpf-adventures-fiddling-with-the-linux-kernel-and-unix-domain-sockets/#case-study-sniffing-frida-traffic`.

[39] *eBPF Safety*. Accessed on October, 2023. url: `https://ebpf.io/what-is-ebpf/#ebpf-safety`.

[40] *Erebus Resurfaces as Linux Ransomware*. Accessed on October, 2023. url: `https://www.trendmicro.com/en%5C%5Fse/research/17/f/erebus-resurfaces-as-linux-ransomware.html`.

[41] Tânia Esteves, Ricardo Macedo, Alberto Faria, Bernardo Portela, João Paulo, José Pereira, and Danny Harnik. “TrustFS: An SGX-Enabled Stackable File System Framework”. In: *38th International Symposium on Reliable Distributed Systems Workshops*. IEEE, 2019, pp. 25–30. doi: 10.1109/SRDSW49218.2019.00012.

[42] Tânia Esteves, Francisco Neves, Rui Oliveira, and João Paulo. “CAT: Content-Aware Tracing and Analysis for Distributed Systems”. In: *22nd International Middleware Conference*. ACM, 2021, pp. 223–235. doi: 10.1145/3464298.3493396.

[43] Facebook. *RocksDB: A Persistent Key-value Store for Fast Storage Environments*. Accessed on October, 2023. url: https://rocksdb.org.

[44] *Fluent Bit: An End to End Observability Pipeline*. Accessed on October, 2023. url: https://fluentbit.io.

[45] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. “X-Trace: A Pervasive Network Tracing Framework”. In: *4th Symposium on Networked Systems Design & Implementation*. USENIX, 2007, pp. 271–284. url: http://www.usenix.org/events/nsdi07/tech/fonseca.html.

[46] Apache Software Foundation. *Apache Hadoop*. Accessed on October, 2023. url: https://hadoop.apache.org.

[47] Cuckoo Foundation. *Cuckoo: Automated Malware Analysis*. Accessed on October, 2023. url: https://cuckoosandbox.org.

[48] Guillaume Fournier, Sylvain Afchain, and Sylvain Baubeau. “Runtime Security Monitoring with eBPF”. In: *17th SSTIC Symposium sur la Sécurité des Technologies de l’Information et de la Communication*. 2021. url: https://www.sstic.org/media/SSTIC2021/SSTIC-actes/runtime_security_with_ebpf/SSTIC2021-Article-runtime_security_with_ebpf-fournier_afchain_baubeau.pdf.

[49] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. “Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to File-System Faults”. In: *ACM Transactions on Storage* 13.3 (2017), pp. 1–33. doi: 10.1145/3125497.

[50] Robin Gassais, Naser Ezzati-Jivan, Jose M Fernandez, Daniel Aloise, and Michel R Dagenais. “Multi-level Host-based Intrusion Detection System for Internet of Things”. In: *Journal of Cloud Computing* 9 (2020), pp. 1–16. doi: 10.1186/s13677-020-00206-6.

[51] Mohamad Gebai and Michel R. Dagenais. “Survey and Analysis of Kernel and Userspace Tracers on Linux: Design, Implementation, and Overhead”. In: *ACM Computing Surveys* 51.2 (2018), pp. 1–33. doi: 10.1145/3158644.

[52] Wael H. Gomaa and Aly A. Fahmy. “A Survey of Text Similarity Approaches”. In: *International Journal of Computer Applications* 68.13 (2013), pp. 13–18. doi: `10.5120/11638-7118`.

[53] S. Halim and F. Halim. *Competitive Programming 3: The New Lower Bound of Programming Contests*. 3rd. Lulu.com, 2013. doi: `10.15388/ioi.2020.14`.

[54] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. “Logmine: Fast pattern recognition for log analytics”. In: *25th ACM International on Conference on Information and Knowledge Management*. ACM, 2016, pp. 1573–1582. doi: `10.1145/2983323.2983358`.

[55] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. “Towards Automated Log Parsing for Large-Scale Log Data Analysis”. In: *IEEE Transactions on Dependable and Secure Computing* 15.6 (2017), pp. 931–944. doi: `10.1109/TDSC.2017.2762673`.

[56] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R Lyu. “A Survey on Automated Log Analysis for Reliability Engineering”. In: *ACM Computing Surveys* 54.6 (2021), pp. 1–37. doi: `10.1145/3460345`.

[57] Dorjan Hitaj, Giulio Pagnotta, Fabio De Gaspari, Lorenzo De Carli, and Luigi V Mancini. “Minerva: A File-Based Ransomware Detector”. In: *arXiv preprint* (2023). doi: `10.48550/arXiv.2301.11050`.

[58] Chris Hunt. *chrahunt/strace-parser*. Accessed on October, 2023. url: `https://github.com/chrahunt/strace-parser`.

[59] Steve Huss-Lederman, Bill Gropp, Anthony Skjellum, Andrew Lumsdaine, Bill Saphir, Jeff Squyres, et al. “MPI-2: Extensions to the message passing interface”. In: *University of Tennessee* (1997). url: `https://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-2.0-sf/mpi2-report.htm`.

[60] Piotr Indyk and Rajeev Motwani. “Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality”. In: *30th Symposium on Theory of Computing*. ACM, 1998, pp. 604–613. doi: `10.1145/276698.276876`.

[61] DPDK Intel. *Data Plane Development Kit*. Accessed on October, 2023. url: `https://www.dpdk.org`.

[62] *Jaeger: Open Source, End-to-End Distributed Tracing*. Accessed on October, 2023. url: `https://www.jaegertracing.io`.

[63] Nishtha Jatana, Sahil Puri, Mehak Ahuja, Ishita Kathuria, and Dishant Gosain. “A Survey and Comparison of Relational and Non-relational Database”. In: *International Journal of Engineering Research & Technology* 1.6 (2012), pp. 1–5. issn: 2278-0181.

[64] Devki Nandan Jha, Graham Lenton, James Asker, David Blundell, and David Wallom. "Holistic Runtime Performance and Security-aware Monitoring in Public Cloud Environment". In: *22nd IEEE International Symposium on Cluster, Cloud and Internet Computing*. IEEE. 2022, pp. 1052–1059. doi: `10.1109/CCGrid54584.2022.00128`.

[65] Amin Kharraz, Sajjad Arshad, Collin Mulliner, William K Robertson, and Engin Kirda. "UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware". In: *25th USENIX Security Symposium*. USENIX, 2016. url: `https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kharaz`.

[66] Amin Kharraz, William Robertson, Davide Balzarotti, Leyla Bilge, and Engin Kirda. "Cutting the Gordian Knot: A Look Under the Hood of Ransomware Attacks". In: *12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015, pp. 3–24. doi: `10.1007/978-3-319-20550-2_1`.

[67] Seong Jo Kim, Seung Woo Son, Wei-keng Liao, Mahmut Kandemir, Rajeev Thakur, and Alok Choudhary. "IOPin: Runtime Profiling of Parallel I/O in HPC Systems". In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE. 2012, pp. 18–23. doi: `10.1109/SC.Companion.2012.14`.

[68] Iman Kohyarnejadfard, Daniel Aloise, Michel R Dagenais, and Mahsa Shakeri. "A Framework for Detecting System Performance Anomalies Using Tracing Data Analysis". In: *Entropy* 23.8 (2021), p. 1011. doi: `10.3390/e23081011`.

[69] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based Learning Applied to Document Recognition". In: *IEEE* 86.11 (1998), pp. 2278–2324. doi: `10.1109/5.726791`.

[70] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. "Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency". In: *ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–14. doi: `10.1145/2670979.2670988`.

[71] *Limon - Sandbox for Analyzing Linux Malwares*. Accessed on October, 2023. url: `https://github.com/monnappa22/Limon`.

[72] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *ACM SIGPLAN Notices* 40.6 (2005), pp. 190–200. doi: `10.1145/1064978.1065034`.

[73] Matti Lyra. *mattilyra/lsh*. Accessed on October, 2023. url: `https://github.com/mattilyra/lsh`.

[74] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. "Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems". In: *25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 378–393. doi: `10.1145/2815400.2815415`.

[75] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. “Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems”. In: *ACM Transactions on Computer Systems* 35.4 (2018), pp. 1–28. doi: `10.1145/3208104`.

[76] Steven McCanne and Van Jacobson. “The BSD Packet Filter: A New Architecture for User-level Packet Capture”. In: *Winter 1993 USENIX Conference*. Vol. 46. USENIX, 1993, pp. 259–269. url: `https://vodun.org/papers/net-papers/van_jacobson_the_bpf_packet_filter.pdf`.

[77] Steven McCanne and Van Jacobson. “The BSD Packet Filter: A New Architecture for User-level Packet Capture”. In: *Winter 1993 USENIX Conference*. Vol. 46. USENIX, 1993, pp. 259–269. url: `https://www.usenix.org/legacy/publications/library/proceedings/sd93/mccanne.pdf`.

[78] Alexandre Montplaisir, Naser Ezzati-Jivan, Florian Wininger, and Michel Dagenais. “Efficient Model to Query and Visualize the System States Extracted from Trace Data”. In: *4th International Conference on Runtime Verification*. Springer, 2013, pp. 219–234. doi: `10.1007/978-3-642-40787-1_13`.

[79] Alexandre Montplaisir-Gonçalves, Naser Ezzati-Jivan, Florian Wininger, and Michel R Dagenais. “State History Tree: An Incremental Disk-Based Data Structure for Very Large Interval Data”. In: *2013 International Conference on Social Computing*. IEEE, 2013, pp. 716–724. doi: `10.1109/SocialCom.2013.107`.

[80] Daniel Morato, Eduardo Berrueta, Eduardo Magaña, and Mikel Izal. “Ransomware Early Detection by the Analysis of File Sharing Traffic”. In: *Journal of Network and Computer Applications* 124 (2018), pp. 14–32. doi: `10.1016/j.jnca.2018.09.013`.

[81] Mohammed Islam Naas, François Trahay, Alexis Colin, Pierre Olivier, Stéphane Rubini, Frank Singhoff, and Jalil Boukhobza. “EZIOTracer: Unifying Kernel and User Space I/O Tracing for Data-intensive Applications”. In: *Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*. ACM. 2021, pp. 1–11. doi: `10.1145/3439839.3458731`.

[82] Francisco Neves, Nuno Machado, and José Pereira. “Falcon: A Practical Log-based Analysis Tool for Distributed Systems”. In: *48th International Conference on Dependable Systems and Networks*. IEEE, 2018, pp. 534–541. doi: `10.1109/DSN.2018.00061`.

[83] Francisco Neves, Nuno Machado, and José Pereira. *fntneves/falcon*. Accessed on October, 2023. 2019. url: `https://github.com/fntneves/falcon`.

[84] Francisco Neves, Nuno Machado, Ricardo Vilaça, and José Pereira. “Horus: Non-Intrusive Causal Analysis of Distributed Systems Logs”. In: *51st International Conference on Dependable Systems and Networks*. IEEE, 2021, pp. 212–223. doi: `10.1109/DSN48987.2021.00035`.

[85] Philip O'Kane, Sakir Sezer, and Domhnall Carlin. "Evolution of Ransomware". In: *IET Networks* 7.5 (2018), pp. 321–327. doi: 10.1049/iet-net.2017.0207.

[86] Adam Oliner, Archana Ganapathi, and Wei Xu. "Advances and Challenges in Log Analysis". In: *Communications of the ACM* 55.2 (2012), pp. 55–61. doi: 10.1145/2076450.2076466.

[87] *OpenTelemetry: High-quality, Ubiquitous, and Portable Telemetry to Enable Effective Observability*. Accessed on October, 2023. url: https://opentelemetry.io.

[88] Harun Oz, Ahmet Aris, Albert Levi, and A Selcuk Uluagac. "A Survey on Ransomware: Evolution, Taxonomy, and Defense Solutions". In: *ACM Computing Surveys* 54.11s (2022), pp. 1–37. doi: 10.1145/3514229.

[89] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. "Practical Whole-System Provenance Capture". In: *2017 Symposium on Cloud Computing*. ACM, 2017, pp. 405–418. doi: 10.1145/3127479.3129249.

[90] Andrew Pollock. *Dstat: Versatile Tool for Generating System Resource Statistics*. Accessed on October, 2023. url: https://linux.die.net/man/1/dstat.

[91] Rogério Pontes, Dorian Burihabwa, Francisco Maia, João Paulo, Valerio Schiavoni, Pascal Felber, Hugues Mercier, and Rui Oliveira. "SafeFS: A Modular Architecture for Secure User-Space File Systems: One FUSE to Rule Them All". In: *10th ACM International Systems and Storage Conference*. ACM, 2017, 9:1–9:12. doi: 10.1145/3078468.3078480.

[92] *ptrace – Linux Manual Page*. Accessed on October, 2023. url: https://man7.org/linux/man-pages/man2/ptrace.2.html.

[93] *Ransomware Spotlight: AvosLocker*. Accessed on October, 2023. url: https://www.trendmicro.com/vinfo/us/security/news/ransomware-spotlight/ransomware-spotlight-avoslocker.

[94] *Ransomware Spotlight: RansomEXX*. Accessed on October, 2023. url: https://www.trendmicro.com/vinfo/us/security/news/ransomware-spotlight/ransomware-spotlight-ransomexx.

[95] *Ransomware Spotlight: REvil*. Accessed on October, 2023. url: https://www.trendmicro.com/vinfo/us/security/news/ransomware-spotlight/ransomware-spotlight-revil.

[96] Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "Can Applications Recover from Fsync Failures?" In: *ACM Transactions on Storage* (2021), pp. 1–30. doi: 10.1145/3450338.

[97] Redis Ltd. *Redis*. Accessed on October, 2023. url: https://redis.io.

[98] Zhilei Ren, Changlin Liu, Xusheng Xiao, He Jiang, and Tao Xie. “Root Cause Localization for Unreproducible Builds via Causality Analysis Over System Call Tracing”. In: *34th International Conference on Automated Software Engineering*. ASE. IEEE, 2019, pp. 527–538. doi: 10.1109/ASE.2019.00056.

[99] Bander Ali Saleh Al-rimy, Mohd Aizaini Maarof, and Syed Zainudeen Mohd Shaid. “Ransomware Threat Success Factors, Taxonomy, and Countermeasures: A Survey and Research Directions”. In: *Computers & Security* 74 (2018), pp. 144–166. doi: 10.1016/j.cose.2018.01.001.

[100] Drew Roselli, Jacob R Lorch, and Thomas E Anderson. “A Comparison of File System Workloads”. In: *2000 USENIX Annual Technical Conference*. USENIX, 2000, pp. 41–54. url: https://www.usenix.org/legacy/event/usenix2000/general/full_papers/roselli/roselli_html/.

[101] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision* 115.3 (2015), pp. 211–252. doi: 10.1007/s11263-015-0816-y.

[102] Abdulqawi Saif, Lucas Nussbaum, and Ye-Qiong Song. “IOscope: A Flexible I/O Tracer for Workloads’ I/O Pattern Characterization”. In: *International Conference on High Performance Computing*. Springer. 2018, pp. 103–116. doi: 10.1007/978-3-030-02465-9_7.

[103] Sanhita Sarkar. “A Scalable Artificial Intelligence Data Pipeline for Accelerating Time to Insight”. Storage Developer Conference. 2019. url: https://www.snia.org/sites/default/files/SDC/2019/presentations/Machine_Learning/Sarkar_Sanhita_A_Scalable_Artificial_Intelligence_Data_Pipeline_for_Accelerating_Time_to_Insight.pdf.

[104] Nolen Scaife, Henry Carter, Patrick Traynor, and Kevin RB Butler. “CryptoLock (and Drop It): Stopping Ransomware Attacks on User Data”. In: *36th International Conference on Distributed Computing Systems*. IEEE, 2016, pp. 303–312. doi: 10.1109/ICDCS.2016.46.

[105] Saiyed Kashif Shaukat and Vinay J Ribeiro. “RansomWall: A Layered Defense System against Cryptographic Ransomware Attacks using Machine Learning”. In: *10th International Conference on Communication Systems & Networks*. IEEE, 2018, pp. 356–363. doi: 10.1109/COMSNETS.2018.8328219.

[106] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep. Google, Inc., 2010. url: https://research.google.com/archive/papers/dapper-2010-1.pdf.

[107] Stephen Smalley, Chris Vance, and Wayne Salamon. “Implementing SELinux as a Linux Security Module”. In: *NAI Labs Report* 1.43 (2001), p. 139. url: http://www.cs.unibo.it/~sacerdot/doc/so/slm/selinux-module.pdf.

[108] Shane Snyder, Philip Carns, Kevin Harms, Robert Ross, Glenn K Lockwood, and Nicholas J Wright. "Modular HPC I/O Characterization with Darshan". In: *5th Workshop on Extreme-Scale Programming Tools*. IEEE. 2016, pp. 9–17. doi: `10.1109/ESPT.2016.006`.

[109] *Strace: Linux Syscall Tracer*. Accessed on October, 2023. url: `https://strace.io`.

[110] Chun Hui Suen, Ryan KL Ko, Yu Shyang Tan, Peter Jagadpramana, and Bu Sung Lee. "S2Logger: End-to-End Data Tracking Mechanism for Cloud Data Provenance". In: *12th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2013, pp. 594–602. doi: `10.1109/TrustCom.2013.73`.

[111] *Sysdig*. Accessed on October, 2023. url: `https://github.com/draios/sysdig/`.

[112] *Sysdig and Falco Now Powered by eBPF*. Accessed on October, 2023. url: `https://sysdig.com/blog/sysdig-and-falco-now-powered-by-ebpf/`.

[113] Kimberly Tam, Aristide Fattori, Salahuddin Khan, and Lorenzo Cavallaro. "CopperDroid: Automatic Reconstruction of Android Malware Behaviors". In: *Network and Distributed System Security Symposium*. 2015, pp. 1–15. url: `http://dx.doi.org/10.14722/ndss.2015.23145`.

[114] Vasily Tarasov, Erez Zadok, and Spencer Shepler. "Filebench: A Flexible Framework for File System Benchmarking". In: *The USENIX Magazine* 41.1 (2016), pp. 6–12. url: `https://www.usenix.org/system/files/login/articles/login_spring16_02_tarasov.pdf`.

[115] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R Ganger. "Stardust: Tracking Activity in a Distributed Storage System". In: *SIGMETRICS Performance Evaluation Review* 34.1 (2006), pp. 3–14. doi: `10.1145/1140103.1140280`.

[116] *Tracee: Linux Runtime Security and Forensics using eBPF*. Accessed on October, 2023. url: `https://github.com/aquasecurity/tracee`.

[117] Karthik Vijayakumar, Frank Mueller, Xiaosong Ma, and Philip C Roth. "Scalable I/O Tracing and Analysis". In: *4th Annual Workshop on Petascale Data Storage*. ACM, 2009, pp. 26–31. doi: `10.1145/1713072.1713080`.

[118] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Chen Zheng, Gang Lu, Kent Zhan, Xiaona Li, and Bizhu Qiu. "Bigdatabench: a Big Data Benchmark Suite from Internet Services". In: *20th International Symposium on High Performance Computer Architecture*. IEEE, 2014, pp. 488–499. doi: `10.1109/HPCA.2014.6835958`.

[119] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. "Ceph: A Scalable, High-Performance Distributed File System". In: *7th symposium on Operating Systems Design and Implementation*. USENIX, 2006, pp. 307–320. url: `https://www.usenix.org/legacy/events/osdi06/tech/full_papers/weil/weil_html/`.

[120] *What We Know About the DarkSide Ransomware and the US Pipeline Attack*. Accessed on October, 2023. url: `https://www.trendmicro.com/en%5C%5Fus/research/21/e/what-we-know-about-darkside-ransomware-and-the-us-pipeline-attac.html`.

[121] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. "Linux Security Modules: General Security Support for the Linux Kernel". In: *11th USENIX Security Symposium*. USENIX, 2002. url: `https://www.usenix.org/legacy/event/sec02/full_papers/wright/wright_html/`.

[122] Cong Xu, Shane Snyder, Vishwanath Venkatesan, Philip Carns, Omkar Kulkarni, Suren Byna, Roberto Sisneros, and Kalyana Chadalavada. *Dxt: Darshan Extended Tracing*. Tech. rep. Argonne National Lab, 2017. url: `https://www.osti.gov/servlets/purl/1392598`.

[123] Wei Xu. "System Problem Detection by Mining Console Logs". PhD thesis. University of California, Berkeley, 2010.

[124] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. "Detecting Large-Scale System Problems by Mining Console Logs". In: *22nd Symposium on Operating Systems Principles*. ACM, 2009, pp. 117–132. doi: `10.1145/1629575.1629587`.

[125] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. "SPDK: A Development Kit to Build High Performance Storage Applications". In: *2017 IEEE International Conference on Cloud Computing Technology and Science*. IEEE, 2017, pp. 154–161. doi: `10.1109/CloudCom.2017.14`.

[126] Seunghoon Yoo, Jaemin Jo, Bohyoung Kim, and Jinwook Seo. "LongLine: Visual analytics system for large-scale audit logs". In: *Visual Informatics* 2.1 (2018), pp. 82–97. doi: `10.1016/j.visinf.2018.04.009`.

[127] Young Jin Yu, Dong In Shin, Woong Shin, Nae Young Song, Jae Woo Choi, Hyeong Seog Kim, Hyeonsang Eom, and Heon Young Yeom. "Optimizing the Block I/O Subsystem for Fast Storage Devices". In: *ACM Transactions on Computer Systems* 32.2 (2014), pp. 1–48. doi: `10.1145/2619092`.

[128] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. "SherLog: Error Diagnosis by Connecting Clues from Run-Time Logs". In: *15th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2010, pp. 143–154. doi: `10.1145/1736020.1736038`.

[129] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. “Improving Software Diagnosability via Log Enhancement”. In: *ACM Transactions on Computer Systems* 30.1 (2012), pp. 1–28. doi: `10.1145/2110356.2110360`.

[130] Hanqi Zhang, Xi Xiao, Francesco Mercaldo, Shiguang Ni, Fabio Martinelli, and Arun Kumar Sangaiah. “Classification of Ransomware Families with Machine Learning Based on N-gram of Opcode”. In: *Future Generation Computer Systems* 90 (2019), pp. 211–221. doi: `10.1016/j.future.2018.07.052`.

[131] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. “Iprof: A Non-intrusive Request Flow Profiler for Distributed Systems”. In: *11th Symposium on Operating Systems Design and Implementation*. USENIX, 2014, pp. 629–644. url: `https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zhao`.

[132] *Zipkin*. Accessed on October, 2023. url: `https://zipkin.io`.