CaT: Content-aware Tracing and Analysis for Distributed Systems

Tânia Esteves
tania.c.araujo@inesctec.pt
INESC TEC & U. Minho
Portugal

Francisco Neves
francisco.t.neves@inesctec.pt
INESC TEC & U. Minho
Portugal

Rui Oliveira
rui.oliveira@inesctec.pt
INESC TEC & U. Minho
Portugal

João Paulo
joao.t.paulo@inesctec.pt
INESC TEC & U. Minho
Portugal

Abstract

Tracing and analyzing the interactions and exchanges between nodes is fundamental to uncover performance, correctness and dependability issues almost unavoidable in any complex distributed system. Existing monitoring tools acknowledge this importance but, so far, restrict tracing to the external attributes of I/O messages, thus missing a wealth of information in them.

We present CaT, a non-intrusive content-aware tracing and analysis framework that, through a novel similarity-based approach, is able to comprehensively trace and correlate the flow of network and storage requests from applications. By supporting multiple tracing tools, CaT can balance the coverage of captured events with the impact on applications' performance.

The conducted experimental evaluation considering two widely used applications (TensorFlow and Apache Hadoop) shows how CaT can improve the analysis of distributed systems. The results also exemplify the trade-offs that can be used to balance tracing coverage and performance impact. Interestingly, in certain cases, full coverage of events can be attained with negligible performance and storage overhead.

CCS Concepts: • **Computer systems organization → Cloud computing**; • **General and reference → Measurement**.

Keywords: Tracing, Distributed Systems, Black-box, Content-aware Analysis

ACM Reference Format:
Tânia Esteves, Francisco Neves, Rui Oliveira, and João Paulo. 2021. CaT: Content-aware Tracing and Analysis for Distributed Systems. In *22nd International Middleware Conference (Middleware '21), December 6–10, 2021, Virtual Event, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3464298.3493396

Middleware '21, December 6–10, 2021, Virtual Event, Canada

ACM ISBN 978-1-4503-8534-3/21/12…$15.00
https://doi.org/10.1145/3464298.3493396

1 Introduction

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.*

Tracing and analysis frameworks can assist these tasks enabling the observation of the applications' requests as they propagate through the distributed system and 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 [22]. However, to efficiently trace a distributed system, the following challenges must be considered:

a) Performance and storage overhead. Capturing I/O events (*e.g.*, network, storage) requires extra processing in the critical path of operations that may lead to significant performance and resource usage (*e.g.*, RAM, CPU) overheads for the observed system. Furthermore, as the number of captured events increases so does the amount of information that must be stored and analyzed [2, 16].

b) Transparency. Distributed systems are composed of heterogeneous components whose source code may be difficult or even impossible to access. Therefore, the collection of events needs to be non-intrusive, treating these components as opaque-boxes, and requiring the least possible knowledge about the target system [19].

c) Accuracy. The need to reduce tracing performance and storage overheads leads many solutions to wittingly discard sets of I/O requests, resorting to sampling [29]. However, missing important requests can have a direct impact on the analysis' accuracy.

d) Causality. In a distributed system, it is fundamental to preserve the events' causality to provide a coherent view of what is happening among the different nodes while showing the operations dependencies. As there is no global clock, and having the physical clocks on all relevant nodes synchronized precisely is often impossible, approaches must devise other ways to infer the causality between events [19].

e) Automation and Visualization. Manually analyzing traces is costly and hard due to the number of events contained in

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

Figure 1. Context vs content-aware tracing analysis.

them. Solutions must implement automatic strategies to analyze and compare distinct events while highlighting relevant details in a summarized and human-readable fashion.

Current tools either take an intrusive approach, requiring source code or binary instrumentation [5, 16, 29], or only take into account the requests' context [19, 23, 30], such as, in general, timestamps and process id, for network messages their source, destination and protocol, and for files their descriptor, name and offset.

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. 1-(a). 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. 1-(b)), 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 paper, we innovate by also exploring the network messages' payload and contents of storage accesses.

Namely, to address the challenges above, this paper proposes CaT , a novel framework for analyzing both the context and content of distributed system 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 (black-box) fashion; ii) summarization and similarity-based techniques to efficiently correlate the content of captured events and visually depict their data flows. In detail, this paper 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 (with a similarity degree of 100%), as well as near-similar 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 for detecting data modification, corruption or leakage for similar I/O messages.

Black-box tracing. The previous algorithm is integrated with two kernel-level tracing tools (Strace[14] and eBPF[17]) for capturing storage and network I/O requests in a non-intrusive fashion. These two technologies provide different trade-offs 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 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 [3]. 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.

The rest of this paper is structured as follows. Section 2 introduces relevant background. Sections 3 and 4 describe CaT's design and implementation, while Sections 5 and 6 detail its experimental evaluation. Finally, Section 7 points to relevant related work, and Section 8 concludes the paper.

2 Background

This section reviews relevant work to better understand how CaT is designed and works.

2.1 Capturing system's events

Linux tracing tools such as Ftrace, LTTng, eBPF, SystemTap, and Strace, are tracing frameworks that run at the operating system level and provide useful insights about an application behavior without requiring its modification or custom instrumentation. Depending on the tool, it is possible to trace system calls (*e.g., openat, pread*), Linux Kernel functions (*e.g., sock_sendmsg*) or user space functions (*e.g., malloc*). This paper explores the Strace and eBPF technologies to build a non-intrusive content-aware tracing framework.

Strace [14] is a command line tool for Unix systems that intercepts and records system calls issued by a process. Its implementation relies on *ptrace* that enables tracing system calls, read and write operations to memory and registers, and manipulating signal delivery to the traced process.

Concretely, whenever the target process enters or exits a system call, it is stopped by the Linux Kernel, allowing the tracer to inspect the program and print its values (*i.e.*, system call name, arguments, and return value). Moreover, Strace can be configured to trace the threads and child processes that are created by a given process.

As a downside, Strace's trap generated by the operating system on each system call, and the context switches of processes from *blocked* to *running* state may impose considerable performance overhead on the target application [9].

eBPF stands for *extended Berkeley Packet Filter* and is a universal in-kernel virtual machine that allows user space applications to inject code in the kernel at runtime (without changing kernel source code or loading kernel modules) [18].

eBPF allows attaching small programs to specific locations in kernel and user space (*i.e., tracepoint, kprobe, uprobe*) to be executed whenever an event (*e.g.*, system call or kernel function) occurs. Moreover, it provides maps (key-value data stores) that allow sharing data between eBPF programs and between kernel and user space. Also, a *ring buffer* is usually used to submit the collected events to the user space. This buffer is a contiguous memory area that can be read (by user space) and written (by kernel space) simultaneously.

Lastly, eBPF is considered a safe option to instrument the Linux Kernel as its *Verifier* performs a sanity-check to the eBPF program before attaching it, ensuring that it cannot threaten the kernel's stability and security. However, this has implications when writing an eBPF program since its total number of instructions, and stack space are limited, only bounded loops are allowed (if the exit condition is guaranteed to be true), and map sizes must be statically defined.

2.2 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 build upon our framework.

Falcon[19] 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:

a) 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.

b) 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.

c) 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 CaT's Design and Architecture

CaT is a black-box content-aware tracing and analysis framework. It analyzes distributed systems in a non-intrusive way, highlighting how their components interact with each other and how data flows through the system. Its design enables the capture of detailed information related to I/O network and disk 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.

3.1 Design Principles

In compliance with the challenges discussed in Section 1, CaT's design considers the following design principles:

a) Kernel-level tracing. CaT resorts to kernel-level tracing to capture the context and content of network and storage

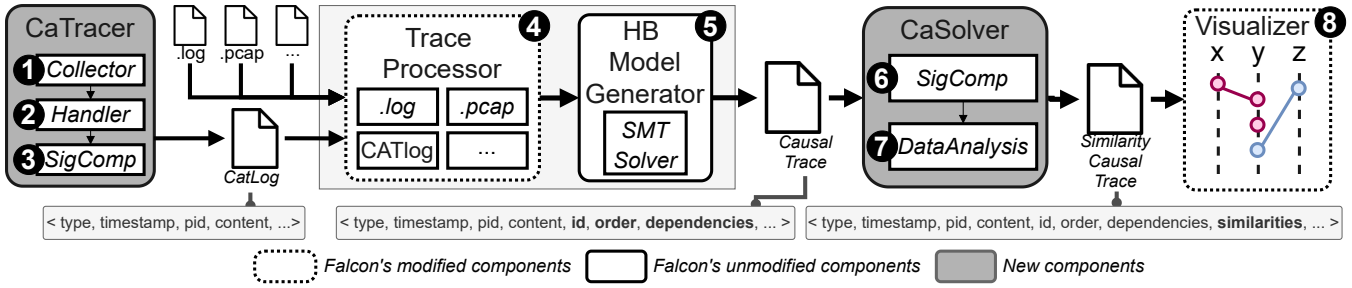


Figure 2. CAT's architecture.

I/O requests, while not requiring previous knowledge about the application nor instrumenting its source code.

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

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

d) Causality inference. CAT extends Falcon to correlate and infer the causality of distributed I/O events [19].

e) 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 Architectural Components

As depicted in Fig. 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. To this end, CAT extends Falcon's architecture for analyzing data in transit and at rest, while providing further information about the targeted system. Namely, CAT includes the following 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 (Fig. 2-❶) resorts to kernel-level tracing facilities that 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.*, OLAP and OLTP databases, analytical and machine learning frameworks).

To minimize the tracing performance and storage overheads, the *CaTracer* offers the possibility of saving only events of interest. Namely, it can 1) filter events by PID and 2) filter storage events by path. The former sets *CaTracer* to collect only the events of a given process identifier 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 (Fig. 2-❷) that parses and organizes it into the *CatLog* events format. This log file holds the events' type, context, and content. For instance, the *CatLog* for Fig. 1 example 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 (Fig. 2-❸). When this submodule is enabled, the *CatLog* file will store the corresponding hash sums instead of the full data buffers' content being intercepted. Section 4.1 further details *CaTracer*'s submodules.

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, which is performed in background and even at different servers. First, the *CatLog* file is forwarded to the *Trace Processor* (Fig. 2-❹). This component will parse and organize the events into different data structures according to their type (*e.g., SocketEvent* – with the socket type, source and destination addresses, and data buffer transmitted, *StorageEvent* – with the file path, descriptor, offset, and 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 path, file descriptor, offset).

HB Model Generator. The next step is to find the happens-before relationships between the events, which is done at the *Happens-Before (HB) Model Generator* (Fig. 2-❺). 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 example, the *Causal Trace* of the example from Fig. 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 the Falcon system 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 (Fig. 2-❸), or the full data buffers. In the latter case, the *CaSolver* module resorts to the *SigComp* submodule to compute buffers' signatures in place (Fig. 2-❻). 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 (Fig. 2-❼) for applying data similarity estimation algorithms to find events with a high probability of operating over the same data flow. These algorithms are further detailed in section 4.2. 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. 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* (Fig. 2-❽), 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* component is provided in section 4.3.

3.3 Pipeline Usage

CAT was designed with the aim to assist developers and system administrators, particularly in analyzing their system behavior and identifying erroneous or suspected I/O flows that may lead to protocol or security flaws. To use CAT , the user must first run the targeted application with the *CaTracer* component to generate the *CatLog* file which contains the captured I/O events. Then, the *CatLog* file should be passed as an argument to the *Trace Processor* component, which will parse the events and share the information with the *HB Model Generator*. In the end, a new file is generated (*Causal Trace*) that contains the events and their corresponding causal relationships. Next, the user must run the *CaSolver* component, passing the *Causal Trace* as input, to get the *Similarity Causal Trace* file with the inferred similarity information. For visualizing the information contained in the latter file, the user can access the web page of the *Visualizer* component.

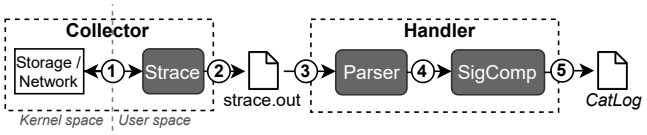


Figure 3. *CatStrace* components.

4 CAT's Algorithms and Prototype

CAT's open-source prototype[1] is based on the Falcon project (commit #997b531 [20]). As depicted in Fig. 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, similarity-based data analysis and content flow visualization.

4.1 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 tracers were chosen as they provide different trade-offs in terms of accuracy, I/O performance and resources usage, as shown in Section 6.1.

4.1.1 CatStrace. The Strace-based tracer is implemented as a Python program that executes the Strace command to trace an application's execution, capturing process, network, and storage-related system calls (*e.g., recvfrom, pwrite64*), and then parses its output into a *CatLog* file.

As depicted in Fig. 3, the *collector* module spawns a process that runs Strace for a given command or PID. Strace intercepts the system calls issued by the traced process (Fig. 3-①) and saves them to a file (strace.out) (Fig. 3-②).

The collected information is then parsed by the *handler* module (Fig. 3-③), whose implementation is based on the *strace-parser* [11] tool. The parser produces a generic JSON structure for all system calls containing the corresponding type (*e.g., pwrite64*), timestamp, PID, arguments (*e.g.*, filename, buffer, size, offset), and the result (*e.g.*, number of bytes written), which is then organized as an event.

Then, the event structure goes through the *SigComp* submodule (Fig. 3-④) that checks if it has content and resorts to the MinHash algorithm (described later in section 4.2) to compute the content's signature. The event with its content signature is then persisted into the *CatLog* file (Fig. 3-⑤).

[1] https://github.com/dsrhaslab/cat

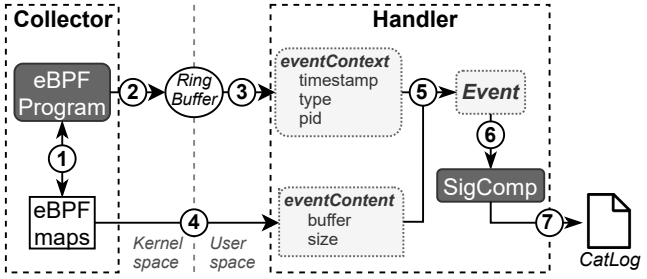


Figure 4. *CatBpf* components.

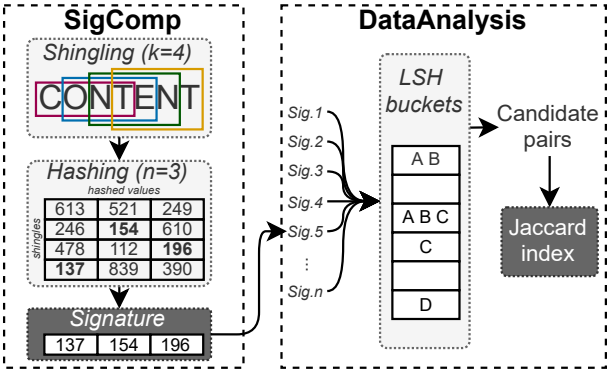


Figure 5. *CaSolver* components.

4.1.2 CatBpf.

The eBPF-based tracer is implemented in GO and resorts to the eBPF technology to capture process, network and storage requests (*e.g., fork, sock_sendmsg, write*). As pictured in Fig. 4, the *collector* module runs at kernel space while the *handler* module runs at user space.

The *collector* module has an eBPF program, written in C, that defines the code that runs when an I/O request (*e.g.*, write) is intercepted. Namely, it first checks if the request was issued by the target process and builds a structure (*eventContext*) that gathers contextual information (*e.g.*, type, timestamp, process ID). If the request is handling data (*i.e.*, has content), for instance a write request persisting a buffer to disk, the *collector* module builds another structure (*eventContent*) that gathers the data buffer and its size. The *eventContent* structures are placed in an eBPF map of type per-CPU array (Fig. 4-①) that can be accessed from user space, while the *eventContext* structures are submitted to user space via the *ring buffer* (Fig. 4-②).

At user space, the *handler* is continually polling the events context from the *ring buffer* (Fig. 4-③) and, when applicable, gets their content from the per-CPU array (Fig. 4-④). It then merges all the collected data into an *Event* structure (Fig. 4-⑤), computes its signature (Fig. 4-⑥), and persists it to the *CatLog* file (Fig. 4-⑦). CatBpf's *SigComp* submodule is similar to the one from CatStrace, but it is implemented in GO.

Our strategy of splitting the context and content of events into two different structures was based on that of unixdump[7] 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 events' 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.

4.2 Similarity-based analysis

The similarity-based analysis of events' content is performed in two phases: 1) the signatures computation (at the *SigComp* submodule), and 2) the processing of events' signatures (at the *DataAnalysis* submodule), as depicted in Fig. 5.

In the first phase, we use the *min-wise hashing (MinHash)* algorithm [4] to summarize the content of each I/O event into a small set of signatures. In a nutshell, the MinHash algorithm applies n different hash functions to each shingle (consecutive overlapping sequences of k bytes) of the intercepted content buffer. Then, for each hash function output it is selected the smallest value obtained, resulting in a signature with n values.

To assess the similarity between the content of two events, at the second phase, we can now calculate the Jaccard index [10] of their signatures, which determines the percentage of identical values present in them. 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.

Therefore, to efficiently compare all MinHash signatures and find the pairs with a similarity greater than a given threshold, we resort to the *locality-sensitive hashing (LSH)* [12] approach. This mechanism uses several hash functions to group MinHash signatures referring to similar content into the same bucket. Then, the Jaccard index is only computed for strong candidate pairs, namely signatures that have been 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 visually represent the

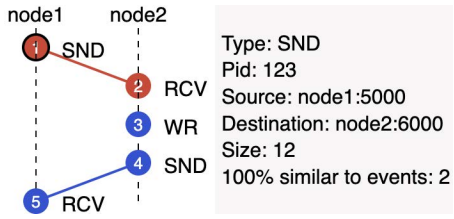


Figure 6. *Visualizer* output example.

events data flow and dependencies. Namely, by looking at the Jaccard index, the *Visualizer* will be able to 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).

4.3 Visual representation

The *Visualizer* generates a space-time diagram depicting the events executed in each host and the inter-host causal relationships. Moreover, by resorting to the similarity information found by the *CaSolver* module, the *Visualizer* employs a color-based scheme to depict the events' content similarities.

Fig. 6 shows the *Visualizer* output for the example from Fig. 1. Each host's events are represented as circles positioned along a dashed line according to the order in which they occurred. For instance, on host *node1* occurred two events, a network *send* (event 1) followed by a network *receive* request (event 5). Each event is accompanied by its ID and type.

Causal relationships are represented by a line linking two events (*e.g.*, events 1 and 2). Data dependencies, *i.e.*, events whose content is similar, are presented with the same color (*e.g.*, events 1 and 2, and events 3, 4 and 5). Events whose content is unique are assigned with the black color.

When selecting a specific event or relationship it is then possible to consult additional information about it. In Fig. 6 we show such information for event 1. It is a *send* request of 12 bytes from *node1* (port 5000) to *node2* (port 6000), from the process with PID 123. Moreover, it summarizes the event's similarities. In this case, it states that the content from event 1 is 100% similar to the one from event 2.

5 Experimental Methodology

The experimental evaluation of CAT was planned to validate the following questions:

- What is the performance, resource usage, and storage overhead of CAT, namely of the two supported tracers[2], at the application's critical I/O path?
- How do the two different tracers vary in terms of accuracy (number of captured events)?
- What novel insights can CAT's content-aware approach provide?

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

Next, we further detail the experimental methodology followed to address these questions.

5.1 Use cases and Workloads

We selected two Big Data applications for our evaluation, namely TensorFlow [1] and Apache Hadoop [3].

TensorFlow is a machine learning platform used for the training and inference of deep neural networks. During the training phase, TensorFlow performs disk I/O operations to read the dataset being used to build the deep-learning model. CAT prototype was used to capture TensorFlow's interactions with the storage medium while reading the ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC2012) dataset[26], which is often used for computer vision research and includes 1.28 million images (≈ 138 GiB) for training and 50,000 images (≈ 6 GiB) for validation, distributed across 1000 classes. The dataset was previously converted to the TFRecord format, resulting in 1152 TFRecords files (1024 for training and 128 for validation), occupying approximately 144 GiB. The conducted tests consisted of running the training workload for 20 epochs, with a batch size of 64, while using a single GPU. Also, the LeNet CNN model[15] was chosen given its disk I/O-bound nature, thus providing a scenario where multiple disk operations must be captured by CAT's tracers [28].

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. 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 prototype was used to intercept network and disk I/O calls across HDFS client, NameNode, and DataNodes.

For these experiments we resorted to BigDataBench 5.0[32], a Big Data benchmark suite that provides representative real-world datasets. BigDataBench ran the Naive Bayes algorithm (a classification algorithm used in data mining) with the Amazon movie review dataset. The experiments also considered the loading phase of this dataset into the HDFS store with two different dataset sizes (16 and 32 GiB).

5.2 Experimental Setups

Experiments included three distinct deployments:

- *Vanilla*: The application running without tracing tools.
- *CatBpf*: The eBPF-based tracer running simultaneously with the target application and intercepting its events. To optimize the number of I/O events handled, CatBpf was configured to capture only the first 4KiB 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.

Table 1. TensorFlow results. '—' indicates that it is not applicable, while '*' means that the values could not be measured.

	Vanilla	*CatBpf*	*CatStrace*
Elapsed time (min)	169.86	173.83	610.56
Images per second	2 527.75	2 495.92	703.07
Events handled	—	11 836 041	*
Events incomplete	—	0	—
Events truncated	—	11 788 963	*
Events lost	—	0	*

- *CatStrace*: The Strace-based tracer intercepting application's events, while capturing 256KiB of requests content, which allowed obtaining the full content buffers for most events.

5.3 Collected Metrics and Experimental Environment

Besides measuring the elapsed time and throughput metrics, the Dstat[24] tool was used to observe the CPU and memory usage on each cluster node. Also, we collected the events statistics reported by each tracer, including the total of events saved into *CatLog*, incomplete events (*i.e.*, *CatLog* events including only context information), truncated events (*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. We performed 3 runs for each Hadoop experiment, and 2 runs for the TensorFlow experiments.

TensorFlow ran on a server equipped with a Intel Core i9-9900K CPU (8 physical and 16 logical cores); 16 GiB of DDR4 RAM; one 1 TiB Micron 2200S NVMe; and NVIDIA GeForce RTX 2070 GPU with CUDA version 10.2.

Hadoop ran in five servers with one Intel Core i5-9500 CPU (6 physical and logical cores); 16 GiB of DDR4 RAM; one 500 GiB, SATA III, Seagate ST500DM009-2F110 HDD; and one 250 GiB, Samsung SSD 970 EVO Plus. The five servers ran Hadoop 2.7.1, while three were configured as DataNodes, one as the NameNode, and the last one as the client. Servers were interconnected by a switched 10 Gigabit Ethernet network.

6 Experimental Evaluation

Next, we discuss the performance and accuracy results for the different tracers and show the usefulness of CaT for better understanding the data flow across complex real systems.

6.1 Content-aware Tracers Evaluation

TensorFlow. Table 1 presents TensorFlow's results, namely the elapsed training time, the number of images processed per second, and the events statistics. As expected, the *Vanilla* setup processed the highest number of images per second (2527.75) and was executed in the shortest time (169.86 min).

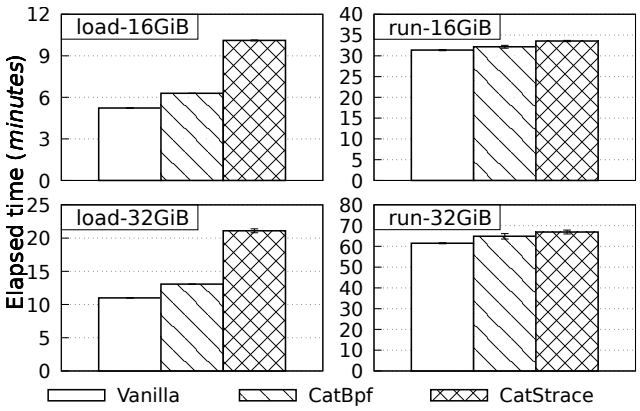


Figure 7. BigDataBench elapsed times.

Comparing to the *Vanilla* setup, the *CatBpf* deployment decreases the images processed per second by 1.26% and increases the elapsed training time by 2.34%. *CatBpf* collected 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 for different files of the ImageNet dataset. As each read operation has approximately 256 KiB, the collected content was truncated to the first 4 KiB, resulting in 99.60% of events truncated. The resulting *CatLog* file, with all the collected events and corresponding context metadata and content signatures, occupied approximately 5.1GiB.

The performance impact imposed by *CatStrace* was higher, achieving only 703.07 images processed per second (27.81% of Vanilla results), with an elapsed time of 610.56 minutes, almost 3.6 times more than the *Vanilla* results. Moreover, the Strace command invoked by *CatStrace* produced a file (*strace.out*) with 7.6TiB 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 1).

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 4KiB, it imposes negligible performance overhead and collects all events. On the other hand, *CatStrace* can collect the full content of requests but imposes high performance and storage overheads.

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. Fig. 7 depicts BigDataBench elapsed times for each phase and dataset size (16 and 32 GiB).

The elapsed times for the *Vanilla* deployment were around 5.23 minutes for *load-16GiB* and 11 minutes for *load-32GiB*. The *CatBpf* setup increased the elapsed time by almost 1.20 times, taking about 6.29 and 13.07 minutes for *load-16GiB* and

Table 2. Collected events for the BigDataBench experiments. '—' indicates that it is 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	—

load-32GiB, respectively. The *CatStrace* deployment lasted around 10.11 minutes for *load-16GiB* and 21.10 minutes for *load-32GiB* (almost 1.93 times more than the *Vanilla* setup).

Concerning the *run-16GiB* test, the *Vanilla* deployment ran in 31.37 minutes, while *CatBpf* and *CatStrace* executions lasted for 32.18 and 33.56 minutes, respectively. As for *run-32GiB*, the elapsed times were 61.52, 64.86 and 66.91 minutes for *Vanilla*, *CatBpf* and *CatStrace* setups, respectively.

The loading phase generated more I/O requests in a shorter time span when compared to the running phase, explaining why the performance impact was more significant in the former. As shown in Table 2, at the loading phase, *CatBpf* captured all the network and storage requests (around 8 million on the *load-16GiB* test and 17 million on the *load-32GiB* test), with approximately 9.18% of *incomplete* events and had to truncate the captured content of 42% of *handled* events. The *CatStrace* deployment collected about 16M and 32M requests for loading phases of 16GiB and 32GiB, respectively. *CatStrace* saved all relevant events to the *CatLog* file and only truncated 2 content buffers that were larger than 256KiB.

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

Tracers were configured to only capture HDFS's data and metadata operations, while requests to third-party libraries and applications (*e.g.*, java) were ignored. While the requests that reach the *CatBpf handler* (*handled* events) no longer include ignored operations, as these were 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 processes that were already created by the target application at the tracing time. That is why the number of *saved* events, specially for the running phases, is higher than the one from *CatStrace*.

The results show that *CatStrace* can capture all the events, truncating almost none of them, but generates a significant performance overhead and a large output trace log. For example, Strace created a file of 120GiB when tracing only the network and storage events for one of the HDFS DataNodes during the running phase of 32 GiB. Once more, the *CatBpf* deployment shows to be the one with the best trade-offs, 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 captured the context of almost all the events while being the tracer that incurs the least performance overhead.

Dstat results. For the TensorFlow tests, the *Vanilla* deployment used 5.6GiB of RAM and 43% of CPU. The *CatBpf* setup increased those values up to 12.1GiB 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 4.8GiB of RAM and 14.5% of CPU. As *CatStrace* delays I/O requests and generates less load on the system, resources utilization is also lower.

Regarding BigDataBench load experiments, the *Vanilla* deployment used 3.7GiB of RAM on the client node, 0.8GiB on the NameNode, and 0.6GiB on DataNodes. *CatBpf* required additional 3GiB, for each type of node, while *CatStrace* reduced RAM consumption by 1GiB at the client node. For the remaining nodes, *CatStrace* used around the same amount of RAM as the *Vanilla* setup.

For the same experiments, the *Vanilla* setup used 1.5% of CPU on the NameNode, 10.3% on the DataNodes, and 98.2% on the client node. *CatBpf* increased CPU usage by 15% for the NameNode and 30% for the DataNodes. The values for the *CatStrace* are similar to the *Vanilla* ones, except for the CPU usage on the client machine that required 70% of CPU.

As for BigDataBench run experiments, the *Vanilla* setup used 1.9GiB of RAM on the client node, 1GiB on the NameNode, and 2.4GiB on the DataNodes. *CatBpf* imposed an increase of 2GiB on each server while *CatStrace* used 1GiB less at the client node. CPU usage was similar for the *Vanilla*

and *CatStrace* setups, approximately 0.4% for the client node, 1.3% for the NameNode and 46% for DataNodes. *CatBpf* increased CPU usage by 15% across all nodes.

Discussion. 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 4KiB 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 ~ 1147 events/s), *CatBpf* can collect the content and context of all requests. When tracing a more I/O intensive application (*e.g.*, BigDataBench ~ 22308 events/s for *load-16GiB*), *CatBpf* starts losing information (high percentage of incomplete events and loss of some requests). 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. If 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.

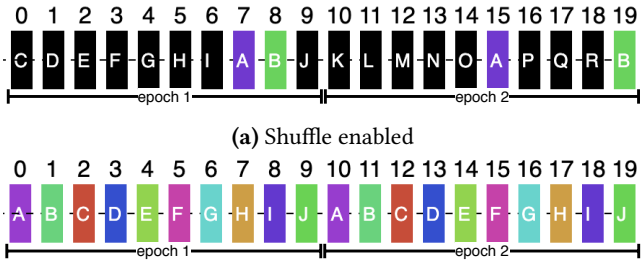
One aspect to take into account is the implications of lost information at the analysis phase. Namely, when truncating the events' content thus capturing only the first X bytes of their payload, events with the same first X bytes but with a different payload for the remaining content will be matched 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 resource 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 and storage space overhead, *CatStrace* should be used.

6.2 CaT Framework in Action

Two possible purposes for using tracing frameworks are to analyze data access patterns and the correction / adulteration of protocols. Next, we show how CaT can be useful for both cases.

TensorFlow Dataset shuffle. CaT was used to analyze TensorFlow's training phase and observe the access pattern used to read the dataset from disk. Typically, a dataset is 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). On each epoch, it is usual to randomly shuffle the data records that are going to be read to keep the model general while avoiding it from overfitting and decreasing its accuracy. If shuffling is disabled, all epochs will fetch (read) data records in the same order.



(a) Shuffle enabled

(b) Shuffle disabled

Figure 8. Disk access pattern for TensorFlow's dataset shuffle.

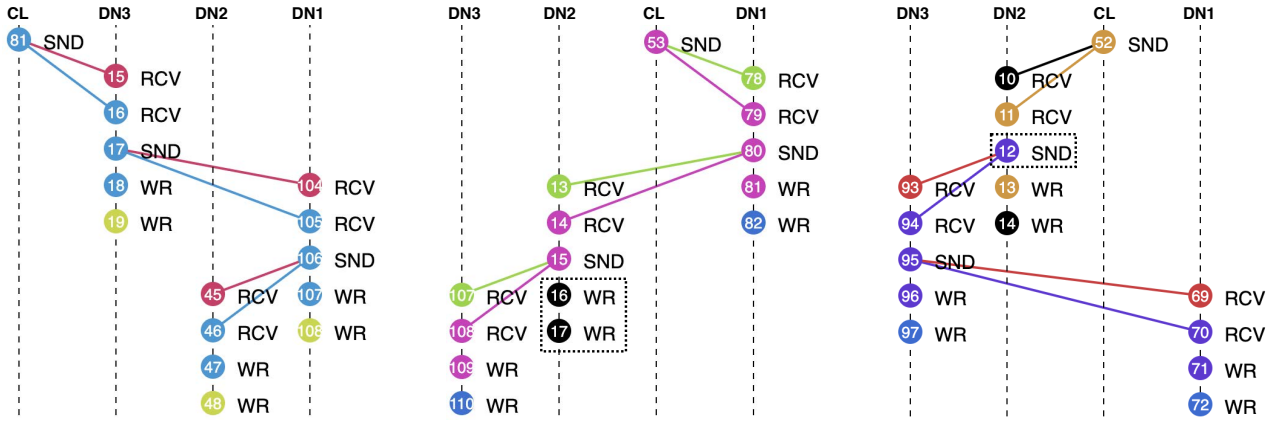
We ran LeNet model for two training epochs with shuffling enabled and disabled. A sample of the ImageNet training set including 64 tfrecords with a total of 64 images was chosen. *CatBpf* was used to capture events and the resulting *CatLog* file was provided to the remaining CaT's pipeline.

Fig. 8 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. 8-(a)), TensorFlow accesses disk records (ImageNet images) in random order. Therefore the order in which data is read differs between epochs. The only similarities found were between events 7 and 15 and events 8 and 19. While on the first epoch, event 7 was the eighth operation, on the second epoch, the same data was read in sixth place (event 15). The same happened 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. 8-(b)), TensorFlow reads the train set files in the same order (deterministic access pattern) at each epoch, as depicted by CaT's output.

HDFS File replication. As another example, CaT traced the send of a file to HDFS to verify if the replication protocol was being correctly applied. More precisely, we instantiated *CatBpf* on the client machine to capture the events issued by the HDFS `copyFromLocal` command, and we ran another



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

Figure 9. HDFS replication of a file.

instance of *CatBpf* for each of the three DataNodes. The resulting *CatLog* files were 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.

Fig. 9-(a) depicts the visual output from CAT's pipeline. Event 81 corresponds to the sending (SND) of the file content by the client to the DataNode 3 (DN3). In turn, DN3 received the data in two receive (RCV) events (15 and 16), forward it to DN1 (event 17), and then saved the corresponding data (event 18) and metadata (event 19) on disk. DN1 did the same thing, 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 (blue color) and the metadata (green color) 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: 1) storage corruption, *i.e.*, DN2 alters the file content before persisting it on disk (Fig. 9-(b)), and 2) network corruption, *i.e.*, DN2 sends the wrong data content to another DataNode (Fig. 9-(c)).

For the first case (Fig. 9-(b)), DataNodes 1 and 3 have a *write* event (events 81 and 109) with the same color as the *send* event from the client (event 53), indicating that their content is similar. However, the *write* events from DN2 (events 16 and 17) have a black color, as the data and metadata persisted is no longer equal to the one DN2 received (event 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. 9-(c)), the client sent a file (event 52) to DN2 (event 11). Then, DN2 forwarded the data to DN3 (event 12) and persisted it to disk (event 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.

Discussion. 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 the 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.

Moreover, when choosing the appropriate set of tracing tools (Section 6.1), we show that CAT can be used over real systems while imposing a balanced trade-off 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.

Finally, for automation purposes, CaT's pipeline could be improved to: 1) save the captured events directly to the analysis components, allowing for near real-time analysis and avoiding the cost of saving all the events to disk; 2) use a database to save the pipeline outputs, enabling queries and automated procedures over the results; and 3) use the visualization component to depict only a query of interest instead of the whole captured data. However, the automation and optimization of these components are an interesting future research direction which is orthogonal to this paper.

7 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.

A common approach is to use static analysis or machine learning algorithms to extract information from application logs [6, 33–35]. However, the typical information available at these logs makes it hard, if not impossible, to correlate events across heterogeneous and distributed components.

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 [5, 8, 13, 16, 27, 29, 31]. However, this approach requires prior knowledge or access to the source code of targeted systems, thus making it less transparent and less applicable to a wider range of scenarios.

Non-intrusive approaches resort to kernel-level tools (*e.g.*, Strace, LSM, NetFilter) to capture applications requests [19, 21, 23, 25, 30]. 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. 1) or content flows such as those depicted for HDFS in section 6.2. These can only be revealed when observing the content of requests.

Unixdump[7] and Re-Animator[2] 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 requests simultaneously, while being restricted to request tracing, thus not providing any analysis mechanism.

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 black-box tracing, correlation, analysis, and visualization of distributed I/O events.

8 Conclusion

This paper 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, depending on the target workload, it is possible to capture most of the I/O events while incurring negligible performance overhead. Moreover, it showcases 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 find performance, correctness and dependability issues in today's complex systems.

9 Acknowledgments

We would like to thank our shepherd Christine Julien and the anonymous reviewers for their insightful comments that helped us improve this paper. We also thank Nuno Machado, Ricardo Macedo, Cláudia Brito and Mariana Miranda for their comments and suggestions.

This work was financed by the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through Ph.D. grant DFA/BD/5881/2020, and realized within the scope of the project BigHPC – POCI-01-0247-FEDER-045924, funded by the ERDF - European Regional Development Fund, through the Operational Programme for Competitiveness and Internationalization (COMPETE 2020 Programme) and by National Funds through FCT, I.P. within the scope of the UT Austin Portugal Program.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Ibrahim Umit Akgun, Geoff Kuenning, and Erez Zadok. 2020. Re-Animator: Versatile High-Fidelity Storage-System Tracing and Replaying. In *Proceedings of the 13th ACM International Systems and Storage Conference (SYSTOR)*. ACM, 61–74. https://doi.org/10.1145/3383669.3398276

[3] Apache Software Foundation. 2015. Hadoop. Retrieved April, 2021 from https://hadoop.apache.org

[4] Andrei Z. Broder. 1997. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences 1997*. IEEE, 21–29. https://doi.org/10.1109/SEQUEN.1997.666900

[5] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. 2002. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN)*. IEEE, 595–604. https://doi.org/10.1109/DSN.2002.1029005

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

[7] Jeff Dileo and Andy Olsen. 2019. eBPF Adventures: Fiddling with the Linux Kernel and Unix Domain Sockets. Retrieved April, 2021 from 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

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

[9] Mohamad Gebai and Michel R. Dagenais. 2018. Survey and Analysis of Kernel and Userspace Tracers on Linux: Design, Implementation, and Overhead. *Comput. Surveys* 51, 2 (2018), 1–33. https://doi.org/10.1145/3158644

[10] Wael H. Gomaa and Aly A. Fahmy. 2013. A Survey of Text Similarity Approaches. *International Journal of Computer Applications* 68, 13 (2013), 13–18. https://doi.org/10.5120/11638-7118

[11] Chris Hunt. 2019. chrahunt/strace-parser. Retrieved April, 2021 from https://github.com/chrahunt/strace-parser

[12] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the 30th Symposium on Theory of Computing (STOC)*. ACM, 604–613. https://doi.org/10.1145/276698.276876

[13] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS)*. The Internet Society. http://bitblaze.cs.berkeley.edu/papers/dta++-ndss11.pdf

[14] Paul Kranenburg, Branko Lankester, Rick Sladkey, et al. 2021. strace: Linux syscall tracer. Retrieved April, 2021 from https://strace.io

[15] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324. https://doi.org/10.1109/5.726791

[16] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*. ACM, 378–393. https://doi.org/10.1145/2815400.2815415

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

[18] Q. Monnet. 2016. Dive into BPF: a list of reading material. Retrieved April, 2021 from https://qmonnet.github.io/whirl-offload/2016/09/01/dive-into-bpf/

[19] Francisco Neves, Nuno Machado, and José Pereira. 2018. Falcon: A practical log-based analysis tool for distributed systems. In *Proceedings of the 48th International Conference on Dependable Systems and Networks (DSN)*. IEEE, 534–541. https://doi.org/10.1109/DSN.2018.00061

[20] Francisco Neves, Nuno Machado, and José Pereira. 2019. fntneves/falcon. Retrieved April, 2021 from https://github.com/fntneves/falcon

[21] Francisco Neves, Nuno Machado, Ricardo Vilaça, and José Pereira. 2021. Horus: Non-Intrusive Causal Analysis of Distributed Systems Logs. In *Proceedings of the 51st International Conference on Dependable Systems and Networks (DSN)*. 212–223. https://doi.org/10.1109/DSN48987.2021.00035

[22] Adam Oliner, Archana Ganapathi, and Wei Xu. 2012. Advances and Challenges in Log Analysis. *Commun. ACM* 55, 2 (2012), 55–61. https://doi.org/10.1145/2076450.2076466

[23] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. 2017. Practical Whole-System Provenance Capture. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC)*. ACM, 405–418. https://doi.org/10.1145/3127479.3129249

[24] Andrew Pollock. 2020. dstat(1) - Linux man page. Retrieved April, 2021 from https://linux.die.net/man/1/dstat

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

[26] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, et al. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252.

[27] Nikolaos Sapountzis, Ruimin Sun, Xuetao Wei, Yier Jin, Jedidiah Crandall, and Daniela Oliveira. 2020. MITOS: Optimal Decisioning for the Indirect Flow Propagation Dilemma in Dynamic Information Flow Tracking Systems. In *Proceedings of the 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1090–1100. https://doi.org/10.1109/ICDCS47774.2020.00093

[28] Sanhita Sarkar. 2019. A Scalable Artificial Intelligence Data Pipeline for Accelerating Time to Insight. (2019). Storage Developer Conference.

[29] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google, Inc. https://research.google.com/archive/papers/dapper-2010-1.pdf

[30] Chun Hui Suen, Ryan KL Ko, Yu Shyang Tan, Peter Jagadpramana, and Bu Sung Lee. 2013. S2Logger: End-to-End Data Tracking Mechanism for Cloud Data Provenance. In *Proceedings of the 12th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 594–602. https://doi.org/10.1109/TrustCom.2013.73

[31] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R Ganger. 2006. Stardust: Tracking Activity in a Distributed Storage System. *SIGMETRICS Performance Evaluation Review* 34, 1 (2006), 3–14. https://doi.org/10.1145/1140103.1140280

[32] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, et al. 2014. Bigdatabench: a Big Data Benchmark Suite from Internet Services. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 488–499. https://doi.org/10.1109/HPCA.2014.6835958

[33] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. 2009. Detecting Large-Scale System Problems by Mining Console Logs. In *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP)*. ACM, 117–132. https://doi.org/10.1145/1629575.1629587

[34] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. SherLog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 143–154. https://doi.org/10.1145/1736020.1736038

[35] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *Prooceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 629–644. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zhao