The Case for Learned Provenance Graph Storage Systems

Hailun Ding, Juan Zhai, Dong Deng, and Shiqing Ma, *Rutgers University*

https://www.usenix.org/conference/usenixsecurity23/presentation/ding-hailun-provenance

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

The Case for Learned Provenance Graph Storage Systems

Hailun Ding
Rutgers University

Juan Zhai
Rutgers University

Dong Deng
Rutgers University

Shiqing Ma
Rutgers University

Abstract

Cyberattacks are becoming more frequent and sophisticated, and investigating them becomes more challenging. Provenance graphs are the primary data source to support forensics analysis. Because of system complexity and long attack duration, provenance graphs can be huge, and efficiently storing them remains a challenging problem. Existing works typically use relational or graph databases to store provenance graphs. These solutions suffer from high storage overhead and low query efficiency. Recently, researchers leveraged Deep Neural Networks (DNNs) in storage system design and achieved promising results. We observe that DNNs can embed given inputs as context-aware numerical vector representations, which are compact and support parallel query operations. In this paper, we propose to learn a DNN as the storage system for provenance graphs to achieve storage and query efficiency. We also present novel designs that leverage domain knowledge to reduce provenance data redundancy and build fast-query processing with indexes. We built a prototype LEONARD and evaluated it on 12 datasets. Compared with the relational database Quickstep and the graph database Neo4j, LEONARD reduced the space overhead by up to 25.90x and boosted up to 99.6% query executions.

1 Introduction

In the past years, an uptick in cybercrime has emerged. The number of cybercrime in 2021 increased by 600%, and the estimated annual damage hits 6 trillion dollars [55]. In this single year, several severe attacks, e.g., Kaseya ransomware attack [57] (compromising up to 1,500 companies, $70 million loss), Saudi Aramco data breach [58] (sensitive data and technical specification leakage, $50 million payment), Accellion FTA data breach [56] (impacting over 100 organizations over the world), Pulse Secure VPN zero-day attack [15] (breach of undisclosed defense firms and government organization in the U.S. and Europe), Solarwinds supply chain attack [50] (the most significant cyberattack against the U.S. government ever), happened. On the other hand, attack investigation is becoming more and more challenging [35]. It takes 228 days to identify an attack and 80 days to contain an attack. In industries that face more attacks, such as healthcare, the detection of an attack takes 329 days on average [35]. Even worse, the *dwell-time*, i.e., the time an attack remains undetected, can be longer than a year [19], leading to millions of dollars in costs. The increasing number of attacks, the difficulty of identifying and containing attacks, and the long dwell-time call for us to rethink the security infrastructure to support effective and efficient security investigation.

1.1 Background & Existing Work

Attack Investigation. Attack investigation is one of the forensic investigation tasks, which typically leverages provenance graphs. Specifically, a provenance graph is a directed graph with nodes representing system objects (e.g., files, sockets) and system subjects (i.e., processes), and edges denoting causal relationships. At system runtime, provenance trackers [6, 40, 44, 47, 48, 49, 54, 64] collect the provenance of system objects and subjects. Investigation systems can automatically build provenance graphs by analyzing the causal relationships among them (Step A in Figure 1). To ensure the data integrity, provenance trackers can only *append* new data but cannot modify existing data. During the investigation, analysts can leverage the generated graphs to locate the root cause and understand the ramifications of an attack [6, 26, 27, 29, 32, 40, 45, 46, 51, 53, 54, 73]. Typically, investigators query the graph to extract related information based on standard algorithms. For example, to locate the root cause of an attack, investigators perform backtracking (Step B in Figure 1) to extract all system events that are causally related to the given symptom event (colored `Payload` in Figure 1) before it happens. Compared with system call sequence methods [20, 62, 65, 70, 71], provenance graphs provide a more intuitive way to present the detailed historical information of system execution and yield better results [11, 39, 68, 69, 73].

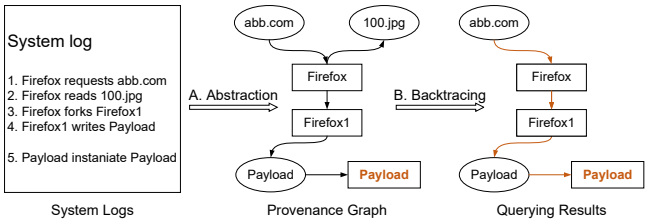


Figure 1: Example of Generating and Querying a Provenance Graph (Oval: files; Rectangle: processes; Colored vertex: attack symptoms). By backward tracing from an attack symptom process `Payload`, we can find the root cause of the attack.

Provenance graphs used in forensic tasks can be large [28, 43, 72], and the reason is two-fold. First, modern software systems are complex, which naturally leads to large provenance graphs. For example, though loading a complex webpage such as CNN in Firefox only takes a few seconds, it issues nearly 22,000 system calls, which touch and create thousands of nodes and edges in the provenance graph. As another example, reports show that browsers like Firefox can have around 10 GB of data per day written to the SSD even if the computer is idle with nothing but a few browser connections [63]. Second, the complexity of attacks (measured by victim target organizations and length of lifecycles) requires reviewing large-scale provenance graphs. For example, the 2020 U.S. federal government data breach attack affected over 10 U.S. national departments and nearly 20 private sectors [17], which made it the most significant attack against the U.S. government. The attack lasted for at least nine months without being noticed. Investigating such attacks requires inspecting massive data from various organizations and periods.

Existing Provenance Storage Systems. Existing methods either reduce the size of the graph itself (i.e., graph redundancy removal, storing less data) or utilize a better encoding (i.e., using fewer bits to store the same amount of data) to solve the high storage problem. As summarized in Table 1, graph redundancy removal methods remove the graph content or modify the graph structure to reduce the redundancy and save space. Encoding-based methods focus on the next stage (i.e., storing the provenance graphs in a more efficient format). The two methods are complementary to each other. LogGC [43], Xu et al. [72], and NodeMerge [66] remove unnecessary components in the provenance graph as long as they do not affect the provenance of attack system activities (e.g., repeated nodes and edges), which is lossy. SEAL [18] removes "unnecessary" information and provides a schema to recover them, achieving lossless decompression. The storage efficiency of these methods depends on the redundancy they identify. Because they do not change the storage format, whether they support queries or not relies on the storage format. Encoding-based compression methods optimize the storage format and also provide support for queries. Gzip [25], which compresses the whole graph as a single file, provides limited support for queries via utilities like `zcat`, `zgrep`, `zless`, and `zdiff`. These tools are mostly designed for string operations and have poor support for querying graph structures.

Modern provenance analysis solutions store provenance graphs in databases. For example, SPADE [22] supports using different databases to store provenance graphs. Different from existing graph reduction methods that focus on data reduction, databases support data storage and query [38]. These two types of methods are compatible and complementary to each other. Figure 2 simplifies and illustrates how relational databases and graph databases store the provenance graph. In Figure 2, we show a simplified provenance graph. A relational database first flattens it to vertices and edges (Step ①). It extracts the identifier and all properties associated with one vertex/edge and then uses a JSON format to represent the graph. The database stores the graph with two tables, i.e., vertex table and edge table, using the identifier as the primary key. The vertex table contains the information of a single system object or subject, such as the vertex type and metadata besides its identifier. Similarly, an entry in the edge table denotes the source and sink of one edge with vertex identifiers and related metadata (e.g., the `Type` field, which indicates what relation these two vertices have).

Analysts can query these tables to fetch vertices and edges using standard SQL language and compose the provenance graph (Step ②). As an example starting from symptom object `V1` (e.g., a malicious process detected by antivirus software), analysts first get its information from the vertex table (e.g., `PID` is `50`). Then, they backtrack its source by querying the edge table and get to know that the source is `V2` in the vertex table. This iterative process continues until reaching the end of the dependency chain (no more back edges from the edge table). The final results consist of vertices `V1` and `V2`, and the edge `E1`, indicating that the attack process `V1` is affected by `V2` according to the activity in `E1`.

Limitations of Existing Methods. Using relational databases to store provenance graphs is not storage efficient because the large graphs contain much redundant information [16, 18, 66], leading to high space overhead. Moreover, relational databases do not support graph queries well because they have a poor performance in representing relations among data points [10] and involve massive I/O operations during the query process. As shown in the previous backward tracing example, it requires querying and joining the vertex and edge tables iteratively to search the relations. The number of queries equals the longest path length of all shortest paths between any vertex and the symptom vertex. Larger graphs also lead to heavier I/O operations.

Graph databases (e.g., Neo4j [1]) and query language (e.g., Cypher [21] and Gremlin [61]) provide better support in storing and querying graphs. As illustrated in Figure 2, graph databases store individual nodes and edges in the graph structure. We can execute queries directly on the graph without

Table 1: Comparison of Provenance Graph Compression Methods

Features	Graph Redundancy Removal				Optimized Encoding-based Storage		
	LogGC [43]	Xu et al. [72]	NodeMerge [66]	SEAL [18]	Gzip	Databases [1, 52]	LEONARD
Compression Type	Lossy	Lossy	Lossy	Lossless	Lossless	Lossless	Lossless
Storage Costs	High	High	Low	Low	Low	High	Very Low
Query Support	Depending on low-level storage systems				No	Yes	Yes

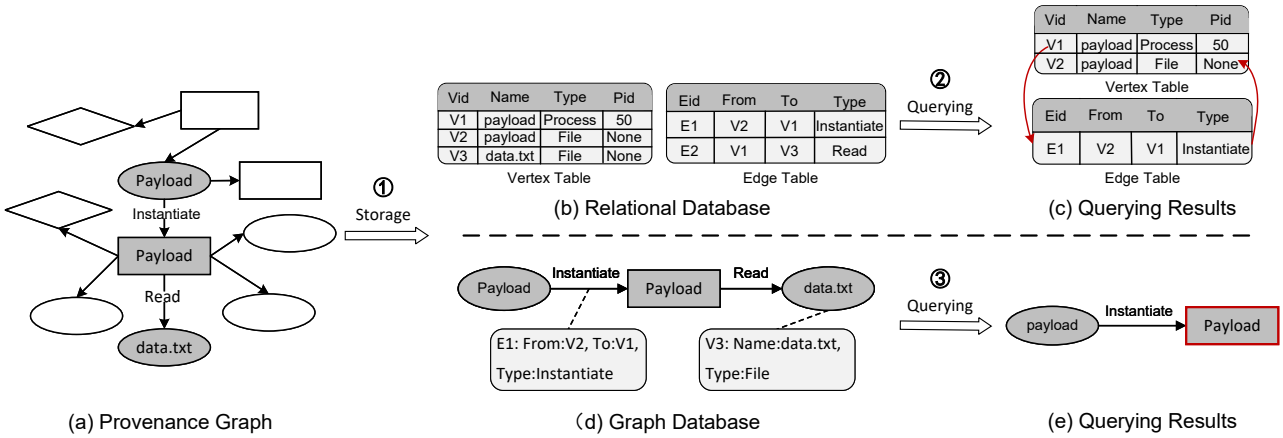


Figure 2: Storage and Querying of Different Databases. In the upper part of the figure, the relational database stores each edge and each node as a separate piece of data in different tables. When investigating the attack, the relational database needs to join and union the tables iteratively. Unlike relational databases that store provenance graphs as tables, in the lower half of the figure, graph databases directly store provenance graphs as graph structures.

table join operations, making it more intuitive and efficient. Even though graph databases provide better representation for graphs, they are not optimal in storage and query efficiency. The main reason is that graph databases store raw graphs. Because these graphs are huge, such graph storage and querying lead to high space and I/O overhead.

1.2 Proposed Solution

DNN and DNN-based Storage Systems. In recent years, DNNs have achieved excellent results in helping design storage systems. A DNN is a layered function that can describe complex data distributions. By using gradient descent methods to optimize the parameters of the DNN, we can learn the data distribution and use the learned DNN to predict the data distribution. According to the universal approximate theory, a neural network with one hidden layer can approximate any continuous function of the input in a specific range [14, 31]. DNNs can represent a complex data distribution (e.g., a text file) with a highly compact numerical vector and make predictions based on given contexts (i.e., queries), which makes it suitable for many components in storage systems. Kraska et al. [41] replaced the B+ Tree index with a DNN and achieved higher query throughput and a smaller index size. Ilkhechi et al. [37], proposed to store tabular data in vector formats and showed great storage efficiency.

To the best of our knowledge, AI-based graph storage systems remain unexplored. Note that DNNs can be Turing-complete and are capable of expressing complex logic. Using an AI model as the storage system has unique benefits. First, compared with traditional logic-based systems, DNNs can learn a more compact data representation [16, 37], which reduces the storage size for the same data. Thus, AI-based storage can be storage efficient. Second, querying DNNs has better performance than logic-based systems. Processing small-sized data (vectors rather than graphs) avoids frequent I/O operations for data movements between different storage devices (e.g., memory and disks). Modern AI models like DNNs naturally support batch queries (i.e., parallelism), which can boost system performance. Operations (e.g., matching) on vectors are more lightweight than string or graph operations. Moreover, the learned vectors embed contexts, making the query more efficient. For example, a linear classifier can test whether a word exists in a sentence or not by only querying the `[CLS]` vector from BERT. Similarly, the vector representation of an edge can reflect the information of its connected vertices. This further speeds up many operations, e.g., edge/vertex filtering and matching. Lastly, the learned vector representation is naturally obfuscated, making it harder to interpret leaked data. Motivated by this, we designed the first AI-based storage system LEONARD, for provenance graphs. The basic idea of LEONARD is converting provenance graph into numerical vectors and then store them using DNNs. LEONARD also supports queries to interact with humans.

Challenges. Using DNNs to store provenance graphs has many challenges. First, existing work [16, 37] shows that storing texts and tabular data in DNN compressed formats are feasible, but how to adapt this knowledge to graph structures is unknown. From the AI perspective, texts and tabular data are in DNN model's Euclidean space while graphs are not, making it harder to train. Thus, how to process the graph structure data and design a proper AI model to train remains challenging. Moreover, using DNNs as a storage system is new, and how to interact with these systems has not been well-explored. For example, traditional solutions optimize query executions via indexes, which provides shortcuts to data access and dramatically improves the performance [67]. The index is essential for efficient query processing. How to design similar mechanisms remains to be explored. Lastly, provenance graphs have domain-specific characteristics, and how to leverage them in system design is unknown. For example, provenance graphs are redundant, e.g., paths sharing prefix directories. Leveraging the domain knowledge can benefit the system's effectiveness and efficiency.

In this paper, we present LEONARD, a novel DNN-based provenance graph storage system. LEONARD first flattens graphs into vertices and edges to avoid training models on complex graph structures and leverages domain knowledge to reduce the redundancy in vertices and edges. Then, we build indexes for vertices and edges to support efficient query execution. Afterward, we use a DNN model to memorize the details of vertices and edges. To alleviate the problem of model misprediction, we leverage a calibration table that records fixes to guarantee lossless data storage. LEONARD provides interfaces that allow users to query and interact with the data. We evaluated LEONARD on 12 datasets. The experiment results show that compared with the relational database Quickstep and the graph database Neo4j, LEONARD can reduce the storage overhead by up to 25.90 times and boost the execution of up to 99.6% queries. LEONARD takes more time than other baseline systems. This is acceptable because LEONARD targets storing provenance graphs that are append-only and cold-data. In typical scenarios, provenance data will be locally stored in databases like Redis as hot data for other applications (e.g., auditing) and pushed to LEONARD for long-term storage (typically once per auditor business cycle). The update operations are append-only writes without modifications to ensure data integrity. As the only reliable source of many security analysis tasks, provenance data is essential. Compared with the cost of storing graphs and support for typical usage patterns (write-once-read-multiple times), the cost is acceptable. We also want to mention that LEONARD is not a general database due to its high overhead and envision that LEONARD potentially can be used as a general graph database with the development of AI acceleration techniques.

In summary, we make the following contributions:

- To the best of our knowledge, we are the first to propose the idea of using DNNs as provenance graph storage systems. We present several novel designs to realize this idea, including domain-specific graph reduction, index construction, and misprediction calibration. The design overcomes the limitations of existing solutions and provides high storage and query efficiency.
- We built a prototype LEONARD and evaluated it on 12 datasets. Our results show that compared with relational database Quickstep and graph database Neo4j, LEONARD respectively takes 17.43 and 25.90 times less storage space on average, and is faster than other databases on 95.2% and 99.6% queries. Our code is available at https://github.com/dhl123/Leonard.

2 Design

The workflow of LEONARD is shown in Figure 3. It consists of three major components: data preparation (§2.1), model training and calibration (§2.2), and query engine (§2.3). After receiving the provenance graph, LEONARD flattens the graph and conducts the preprocessing to prepare the data for training (Component A). Specifically, LEONARD converts vertices and edges in the graph into individual records (similar as ① in Figure 2). This flattening makes it more flexible to extend the graph with new logs. It also converts graphs to Euclidean space, which is easier to train. Then, LEONARD conducts content reduction to remove redundancies caused by monotonous values and repeated long strings in records (A.2). This step helps reduce the storage space. Afterward, we perform content indexing to support fast data querying (A.3). The second component is to train the DNN model and calibrate model mispredictions (Component B). In LEONARD, we use LSTM models because, after data preparation, the data is in Euclidean space and has a similar structure to Natural Language Processing (NLP) tasks. Considering that texts for each node/edge are not long and large models require much space to store, we use lightweight LSTM models rather than Transformer-based models that are harder to train. In case of prediction errors, we create a calibration table to record and fix them when needed. The final artifacts of our storage system contain the reference table, the content index, the DNN model, and the calibration table. Given user queries, the query engine can leverage these files to rebuild the provenance graph (Component C). The engine provides needed primitives to interact with the stored information. A complete example of this process is presented in §A.2.

Scope and Assumptions. LEONARD targets forensics scenarios where analyzing the data happens after the attack is discovered. Due to the long dwell-time, the data has been stored for a long time and the data size is large. In such cold data scenarios, data storage overhead is more of a concern than processing them for storage. LEONARD can store large-scale provenance graphs with minimal space and efficient

operations on the stored graphs. We assume the integrity of the source (i.e., provenance graph) is guaranteed. LEONARD ensures the integrity of stored data during its processing and supports necessary query operations on provenance graphs. As a storage system, LEONARD does not make assumptions about the data content.

2.1 Data Preparation

Raw provenance graphs have low information density. Storing them in the graph format is storage inefficient and makes it hard to support fast querying. LEONARD processes these graphs to reduce redundancies and prepares them for DNN training. This section explains how LEONARD does this.

2.1.1 Step 1: Graph Decoupling

First, LEONARD decouples the graph connection into vertex records and edge records (A.1 in Figure 3). Graph decoupling is a standard processing inspired by how relational databases store provenance graphs (Figure 2). The main reason we decouple the graph is to break the graph structures and flatten them. This decoupling makes our system scalable to larger graphs and converting graphs to sequential data in the Euclidean space, which allows us to leverage existing DNN solutions such as LSTM.

2.1.2 Step 2: Redundant Content Removal

The second step is to remove redundant contents (Step A.2). Specifically, there are two main types of redundant contents in such records: monotonous values and redundant strings.

• **Monotonous Values Reduction.** In the edge and vertex records, some fields have monotonous values. For example, provenance graphs typically represent events that happen within a continuous period, and timestamps of events grow monotonically. Most timestamps have identical prefix digits. For example, in UNIX timestamp, the first a few digits represent the year, month, and day. The prefix digits of records on the same day are the same, leading to high redundancy. For such monotonous values, LEONARD replaces them with base plus offset formats to save space. For example, if the timestamp of an edge is `159.83` and the base timestamp is `159.0`, LEONARD replaces `159.83` with `0.83` (159.83 − 159.0). We want to point out that numbers like timestamps are stored as the floating-point type in traditional databases, making such reduction less useful. By contrast, DNNs do not have data types, and in LEONARD, all numbers are characters. Thus, reducing such monotonous values is helpful in LEONARD.

LEONARD can automatically discover monotonous values and reduce them. It works by first scanning all properties and directly testing if the value change is monotonous. If so, it will automatically pick the minimum value as the base. The reference table will keep records of these rules. When returning results to users, it will automatically do the reverse computation to get the original value (in the query engine).

• **Redundant Strings Reduction.** Provenance graphs also contain many redundant strings, such as keywords of entities, process names, and paths. Replacing them with short annotations is a typical method of reducing such redundancy. For example, Gzip replaces repeated strings after their first occurrence with an annotation. The annotation will include information to reference its last appearance. LEONARD solves this problem by replacing repeated strings with shorter numerical values. Specifically, for each field in the edges and vertices, LEONARD counts the frequency of all values and then sorts them. Similar to Huffman encoding [34], LEONARD replaces values with higher frequency by shorter numerical codes. For example, for the field of `type`, `Process` and `IP Address` are the most frequent strings. LEONARD respectively assigns `0` and `1` to them. Notice that our reduction performs on individual fields instead of the entire content because provenance graphs are well-structured. Such fine-grained reduction can give us better results without confusing users.

2.1.3 Step 3: Content Indexing

A key to support efficient querying is building indexes [67]. Inspired by this idea, we propose to build *indexes* in LEONARD. Forensic analysis frequently accesses both vertices and edges, and we build indexes for both. The format of node index is $\{c : k\}$ where c contains frequently used information of a node, e.g., file name, and k is an input string to the DNN model that can get all information related to the node, i.e., a prompt. For edges, we use $\{(s,t) : k\}$ as our indexes, where s and t respectively represent the source and destination of this edge and k is the prompt (the indexes are necessary IDs to support efficient queries). In general, storing the ids of records as indexes is enough to support all queries (when querying with some specific information like PID, we can decompress each record and find the matched node). Adding some more indexes, such as PID, can accelerate the querying by making the identification process faster. When executing queries, LEONARD first checks indexes to locate strings and then queries the DNN to get detailed information. More details are in §2.2 and §2.3.

2.2 Model Training & Calibration

As discussed before, the DNN in LEONARD is a storage system that is similar to a database: when the user inputs a search query token `r`, the DNN predicts a string that contains all related information. It is a typical sequence-to-sequence task in natural language processing (NLP), such as code completion, text generation, and machine translation. We can use autoregressive language models to solve this problem [7, 59].

In LEONARD, we use LSTM models. In data preparation, LEONARD converts provenance graphs into sequential data. Compared with large sequential models such as Transformers,

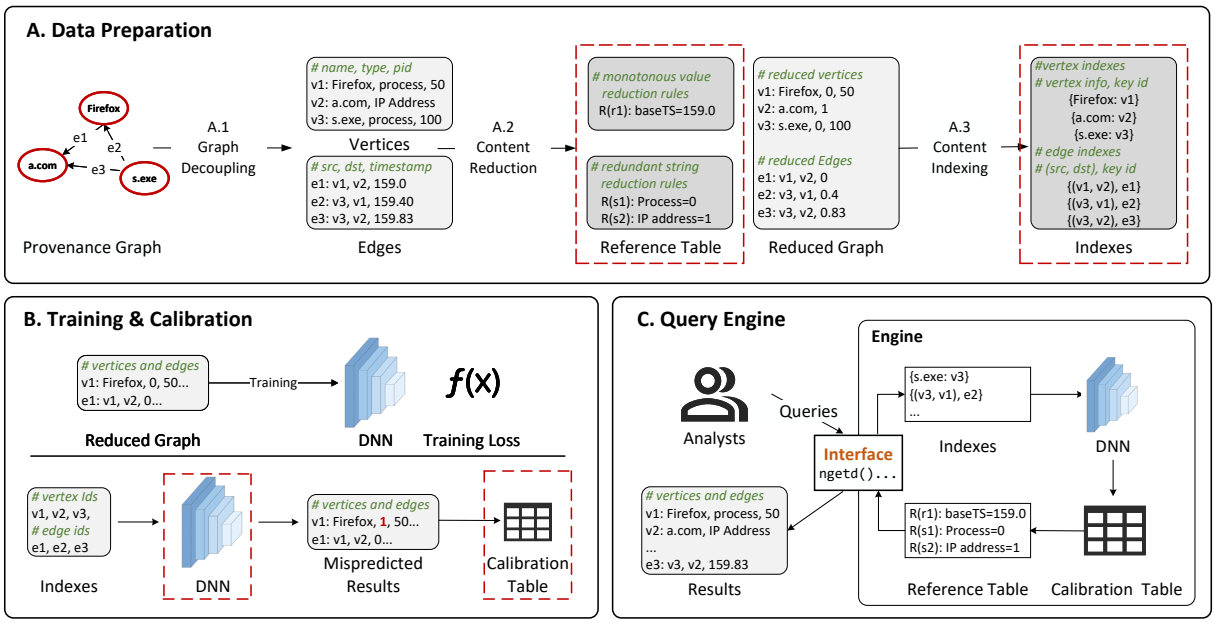


Figure 3: Overview of LEONARD. LEONARD prepares the graph data (Component A) by decoupling the graph into text representation (A.1) and compressing the redundancies in the graph (A.2). Then, LEONARD creates indexes. When training and calibrating the system (Component B), LEONARD trains a DNN model to memorize the data also vertices and edges, and uses a calibration table to store mispredictions. LEONARD provides user interfaces to process general queries (Component C) and return the results.

LSTM models are smaller and can reduce the final storage overhead (models are part of the compressed artifact). Also, LSTM models are easier to train compared with Transformers.

• **Training.** To train this LSTM model, we first embed all vertex/edge records (records of the reduced graph after A.2) into numerical vectors by directly using the char2vec method [9]. LSTM models will memorize the numerical representation of such records. Then, we extract a reference token k for each record. Notice that this token has to be unique. Otherwise, it will confuse the model. Luckily, all records in provenance graphs can be well separated. Otherwise, the graph construction phase should merge them already. Training in LEONARD is just like training other sequence-to-sequence models: we use the token k to predict the first character and use k and all existing characters to predict the next one. During training, we use the cross-entropy loss. We also leverage beam search to increase its handling of long sequences. We stop the model training when the training accuracy stabilizes, which is a common practice in machine learning community. All techniques are standard in NLP.

• **Handling Mispredictions.** In practice, training a model with high training accuracy for a single file requires an excessively large model to memorize the data (which takes much space to store) and more training epochs (causing high time costs). To alleviate this problem, we use a calibration table. Each record in the table has a $(k, [p : c])$ format, where k is the reference token, p is the position offset, and c is the correct character. Namely, after training LEONARD on provenance graphs, LEONARD makes predictions for each character of the graph and memorizes correct characters for each misprediction in the calibration table (the prediction results can be obtained from the last epoch of training without introducing additional scanning). For example, when storing a vertex vector `<v1: Firefox, 0, 50>` in Step B of Figure 3, LEONARD predicts each character in the vector and obtains `<v1: Firefox, 1, 50>`. LEONARD compare the predicted vector with the original one, find the misprediction pair `(v1,[9,0])` (the ninth character of `v1` should be `0`) and stores the pair in the calibration table. Notice that this calibration table is also part of the final artifacts and affects the storage efficiency. The more accurate the model is, the smaller the calibration size is, and vice versa. In all scenarios, we can use the calibration table to fix all mispredictions on the data[1] and guarantee the integrity by design. Existing compression techniques like Gzip can further compress the calibration table. According to §3, the size of calibration tables is not significant. For a 6.49 GB log file, the calibration table is 90.47 MB.

Model Reusing and Ensembling. LEONARD can be more efficient by using transfer learning. We adapt fine-tuning that initializes a model with well-trained weights, allowing faster training convergence. Our results in Figure 11 show that this method speed up the training by 15 times. LEONARD handles new data by ensemble models. That is, LEONARD caches new data until it is large enough or a data cycle is completed. Then, LEONARD trains a new model with the cached data and

[1] Notice that in our task, the training and test data are the same.

updates the system by integrating the new indexes, calibration tables, and trained models.

2.3 Query Engine

We provide primitives for queries in LEONARD (§A.1). For a given query, LEONARD first parses it to get the target vertex/edge (e.g., `s.exe` in Figure 3). Then, it searches the corresponding index to look for keys and asks the DNN to make predictions on the key. After getting prediction results, it leverages the calibration table to fix mispredicted characters. This process gets the reduced graph information. Lastly, LEONARD decodes the result to recover original information by reversely applying the rules in the reference table. A detailed algorithm is provided in §A.1.

3 Evaluation

We first introduce our experiment setup (§3.1). Then, we evaluate the costs of storing provenance graphs (§3.2) and querying (§3.3). Moreover, we include an ablation study to understand how LEONARD perform under different settings (§3.4). Finally, we give a case study to show the integrity of data stored in LEONARD and demonstrate how LEONARD support provenance-based forensics analysis (§3.6).

3.1 Experiment Setup

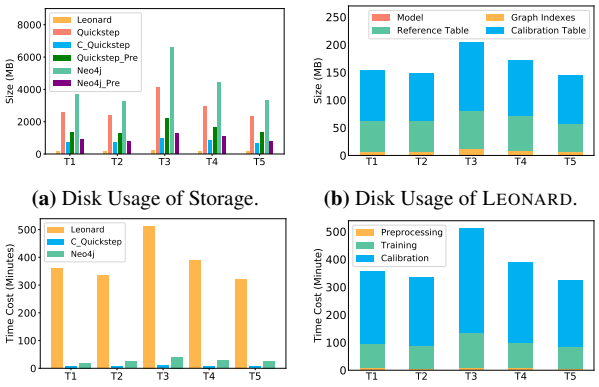
LEONARD is implemented in python 3.6.9 using Keras 2.4.3 [12] with TensorFlow 2.4.1 [4] as the backend. All experiments are done in a Ubuntu 18.04 machine equipped with an RTX 6000 GPU, 64 CPUs, and 376 GB memory.

Datasets and Model Settings. Our datasets consist of 12 provenance graphs from five publicly available log files (i.e., T1 to T5) collected by the trace group in transparent computing engagement #5[1] and seven Linux log files collected by ourselves on an Ubuntu 16.04 machine (i.e., L1 to L7). The details are included in §A.3. Specifically, graph sizes in L1 to L7 grow almost linearly to show how their performance changes when the size of the provenance graph increases.

If not specified, we use the LSTM [30] model described in §A.3 as the default model. The default training settings for trace datasets are batch size 4096, learning rate 0.001, and maximal training epoch 5. The default maximal epoch for the Linux datasets is 15, considering the sizes of Linux datasets are much smaller than trace datasets. Other settings of Linux datasets are the same as those of trace datasets.

3.2 Storing Provenance Graphs

Disk Usage. We evaluate the disk usage of LEONARD and compare LEONARD with Quickstep, compressed Quickstep (we use its built-in compression tools), Quickstep with preprocessing (i.e., we integrate preprocessing in LEONARD), Neo4j and Neo4j with preprocessing. The total disk usage of LEONARD includes the costs of storing reference tables, indexes, DNN models, and calibration tables. For databases, we measure the total size of all data volumes used to store the graph and omit the log files to ensure the fairness of comparison. The overall disk space used by LEONARD and databases trace datasets are in Figure 4(a). The disk space used to store each component of LEONARD is shown in Figure 4(b). The *x*-axis of each figure shows the dataset names, and the *y*-axis is the space used to store provenance graphs.



(a) Disk Usage of Storage. **(b)** Disk Usage of LEONARD.

(c) Time Costs of Storage. **(d)** Time Cost of LEONARD.

Figure 4: Disk Usage and Time Costs of Storing Graphs with Different Systems (C_Quickstep and Pre are shorts for compressed quickstep databases and the preprocessing)

From Figure 4(a), we observe that LEONARD always requires less space to store the same provenance graphs compared to other databases. Compared with Neo4j, Neo4j with preprocessing, Quickstep, compressed Quickstep and Quickstep with preprocessing, LEONARD only uses 3.85%, 17.11%, 5.74%, 21.07% and 10.62% space, respectively. Compared to databases without compression, the preprocessing optimizes redundant fields such as repeated strings. Compared with databases that use preprocessing or similar compression techniques, LEONARD further uses DNNs, removes statistical redundancy, which outperforms them.

When looking into the disk usage of each component (Figure 4(b)), we find that the cost of storing DNN models is the lowest (0.15 MB on average), and the cost of storing calibration tables is the highest (97.87 MB on average) across all datasets. The size of the model file is small because a tiny model is sufficient to fit a large dataset, as also demonstrated by previous work [16, 24]. The cost for storing calibration tables is high due to the mispredictions made on huge datasets. For example, considering the vast size (e.g., 6.57 GB on average) of the trace datasets, recording only 1% mispredictions could take up a lot of space.

Time Costs. We evaluate the time costs of storing provenance graphs with LEONARD and compare the costs with those of

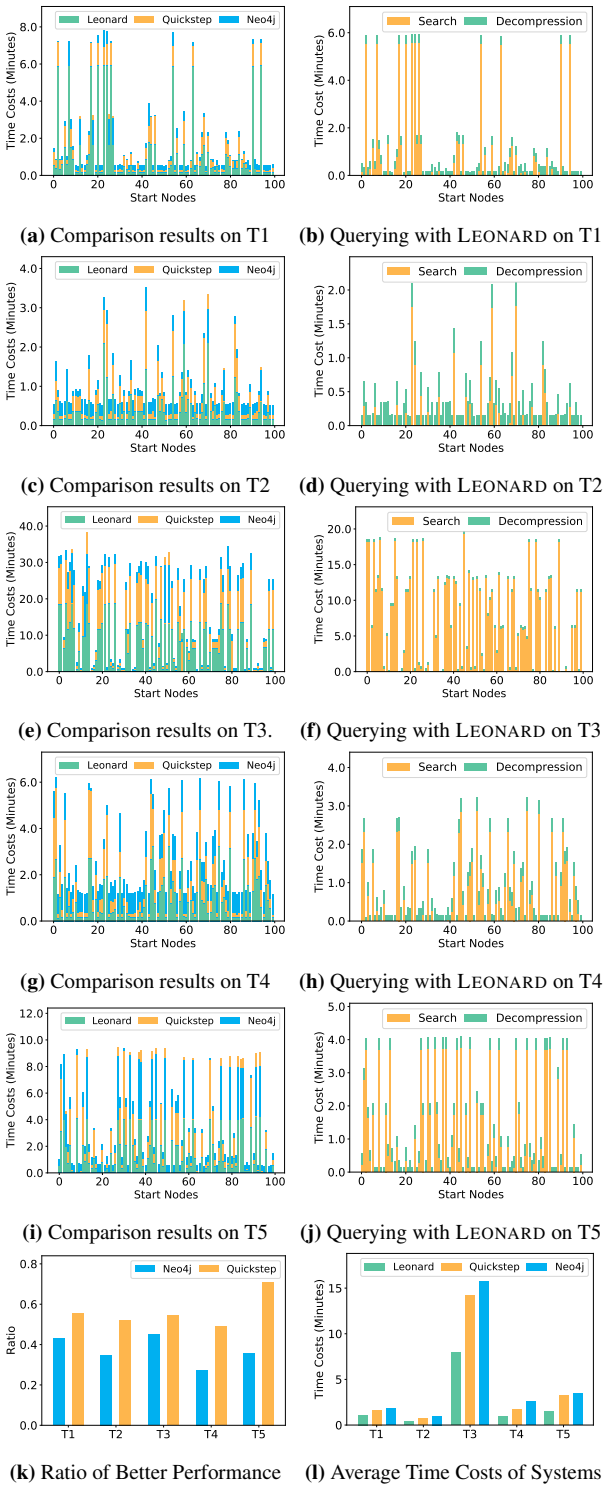
[1] https://github.com/darpa-i2o/Transparent-Computing

databases (implemented in SPADE [22]). Figure 4(c) demonstrates the total time costs of LEONARD and baselines. The time costs of each step in LEONARD are shown in Figure 4(d). In each figure, the *x*-axis shows the dataset names. The *y*-axis represents the time cost of storing graphs. Since the time costs of preprocessing are proved to be impervious in Figure 4(d), we do not additionally show the costs of databases with preprocessing. Moreover, because the time costs of the Quickstep databases are relatively low compared with other systems (the costs of Quickstep databases are from 1.35 to 2.18 minutes on different datasets), we do not show them in the figure.

From Figure 4(c), we observe that the time cost of Quickstep is the lowest among all databases (1.39 minutes on average). Quickstep with compression requires a longer time than the original Quickstep databases, but the difference in time costs is insignificant (the difference is 0.70 minutes on average). Applying compression in Quickstep is not time-consuming because the compression performs traditional character-matching, which does not need any training. The time cost of LEONARD is the highest compared to all databases. The reason is that LEONARD is a training-based method. LEONARD spends much time on training the DNN model and generating the calibration table, as shown in Figure 4(d). The high time costs are essentially caused by the low efficiency of existing model training and inference frameworks. Adopting advanced technology and using powerful machines can decrease the costs. Moreover, as discussed in §3.2, LEONARD only uses 7.89% space of exiting systems when storing provenance graphs. Considering the provenance graph-based attack investigation is a write once and long-term storage task, reducing space overhead is more significant.

3.3 Querying Provenance Graphs

To understand the costs of querying with LEONARD, we measure the time costs of different queries with LEONARD and compare the results with those of other systems. We use default settings discussed in §3.1 for LEONARD. Different configurations may affect the efficiency of LEONARD, and we include a detailed discussion in §3.4. We measure the querying costs of LEONARD on trace datasets (from T1 to T5), which is a common practice [18, 33]. We first randomly select 100 nodes from the provenance graph in each dataset as the starting points of queries. Then, we search their descendant graphs and measure the time costs of this process. Because LEONARD has a decompression process, we measure the total time costs, the time costs of searching graphs, and decompression costs for each query. To avoid the figure becoming over-complicated, we only show the results of LEONARD, Neo4j, and Quickstep. Querying results on compressed Quickstep databases are included in Appendix §A.4. Moreover, analysts do not expect the system to return a huge graph that is not readable [5, 29, 45]. Following previous works [45], we stop the searching and return the results when



(a) Comparison results on T1 **(b)** Querying with LEONARD on T1

(c) Comparison results on T2 **(d)** Querying with LEONARD on T2

(e) Comparison results on T3. **(f)** Querying with LEONARD on T3

(g) Comparison results on T4 **(h)** Querying with LEONARD on T4

(i) Comparison results on T5 **(j)** Querying with LEONARD on T5

(k) Ratio of Better Performance **(l)** Average Time Costs of Systems

Figure 5: Time Costs of Queries. (a), (c), (e), (g) and (i) show the costs on different datasets. Bars represent the costs of different systems and they are overlapped. (b), (d), (f), (h) and (j) show the time costs of querying in LEONARD. The costs of searching and decompression are not overlapped in these figures. The height of each bar is the total costs of queries. (k) and (l) are statistical results that show the average ratio (time costs of LEONARD over the costs of other systems) and average costs of each system.

the number of returned records (i.e., number of edges and vertices) is larger than a threshold (i.e., 4096) to obtain the events that are most related to the attack. The impacts of using different thresholds in querying are evaluated in §3.4.

Results and Analysis. Figure 5(a) to Figure 5(j) show the time costs of queries on trace datasets (T1 to T5). For each dataset, we use one figure to show the total time costs of each query in different systems and one figure to show the time costs of searching and decompression of LEONARD. In these figures, the *x*-axis shows dataset names and the *y*-axis measures the time costs on a minute scale. Figure 5(k) is the statistical comparison results of LEONARD with databases. The *y*-axis shows the average ratio defined as querying costs of LEONARD over the querying costs of databases. Figure 5(l) demonstrates the average time costs of querying with LEONARD and databases on each dataset.

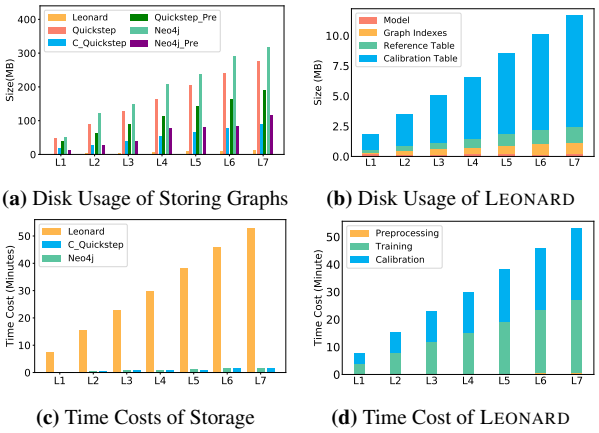
The results in Figure 5(l) show that LEONARD is more time-effecient. The time costs of LEONARD are only 43.09%, 34.70%, 45.00%, 27.45%, and 35.62% of Neo4j on different datasets, respectively. The costs of LEONARD are only 55.80%, 52.19%, 54.33%, 48.99% and 71.11% of Quickstep. When comparing existing databases in Figure 5(l) and Figure 5(k), the time costs of Neo4j databases are slightly higher than Quickstep databases. Because it uses two tables in the relational database to store simple graph structure and graph details. When querying, we only search the simple graph structure. However, during the query process in graph database, we need to search the entire graph database and check each record, yielding heavier overhead than relational databases. Therefore, the time cost of Neo4j is higher than Quickstep.

The main factors affecting searching are the raw graph size and the graph complexity. When the size of the graph or the complexity of the graph increases, searching takes more time. As shown in Figure 5(l) and Table 3, the querying costs on T3 is the highest because the graph in T3 is the largest among all datasets. Also, we inspect the dataset and find that the graph in T3 dataset mainly consists of three large components. Two of them are from long-running `Firefox` processes, and the other is from the `sshd` process. Searching on such complicated graph components consumes much time on searching the related events and sorting their importance values. The factor that affects the time costs of decompression is the returned graph size. When the returned graph is larger, the number of decompressed records increases, and the time costs of decompression increase. Detailed analysis and results are included in §3.4.5.

3.4 Ablation Study

3.4.1 Provenance Graph Size.

To understand how the performance of LEONARD and other systems changes when the provenance graph size increases, we evaluate LEONARD and databases on Linux datasets, where the size of logs and provenance graphs increases almost linearly between each dataset. Specifically, we measure the disk usage and time costs of storing provenance graphs. The results are shown in Figure 6.



(a) Disk Usage of Storing Graphs **(b)** Disk Usage of LEONARD

(c) Time Costs of Storage **(d)** Time Cost of LEONARD

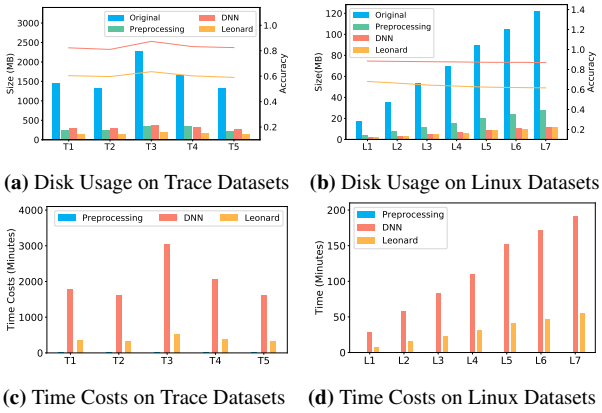
Figure 6: Disk Usage and Time Costs on Linux Datasets

From Figure 6(a), we observe that the disk usage of all systems increases when the provenance graph size increases. Among all systems, Neo4j grows fastest and LEONARD grows slowest. The results further demonstrate the advantage of LEONARD. Moreover, Figure 6(b) shows the increased costs of LEONARD are mainly caused by storing the calibration table. On average, LEONARD uses 5.24 MB to store the calibration table. For other components (i.e., model file, indexes and reference table), the costs are 0.18 MB, 0.57 MB and 0.76 MB. This is because calibration tables need to record mispredictions continuously, but other components may not need to do much modification.

When looking into the time costs of storing graphs (Figure 6(c), i.e., the costs of Quickstep databases are from 2.89 to 14.00 seconds), we find that the time cost also increases linearly in each system. When checking costs of each step in LEONARD (Figure 6(d)), the main costs are from the model training, which is different from the results on trace datasets. This is because we have different training settings for Linux and trace datasets. Since Linux datasets are smaller than trace datasets, we train the model for more epochs on Linux datasets to make the model overfit the data (15 epochs for Linux datasets and five epochs for trace datasets). The different configuration makes the costs of training on Linux datasets more significant than other components.

3.4.2 Impacts of Individual Component in LEONARD.

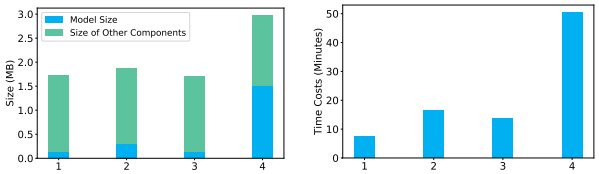
LEONARD consists of two components: the preprocessing for removing redundancies and DNN compression for further optimization. To understand the impacts of each component, we evaluate the effectiveness and the efficiency of preprocessing, DNN compression, and their combination (LEONARD) in isolation. Specifically, we include the original size, the size



(a) Disk Usage on Trace Datasets **(b)** Disk Usage on Linux Datasets

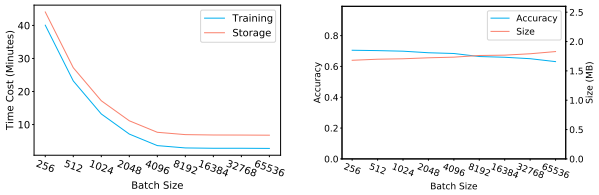
(c) Time Costs on Trace Datasets **(d)** Time Costs on Linux Datasets

Figure 7: Storage and Time Costs of Individual Component



(a) Disk Size Usage for Storage **(b)** Time Costs for Storage

Figure 8: Evaluation Results with Different Models



(a) Costs of Training and Storage. **(b)** Model Accuracy and Disk Usage.

Figure 9: Evaluation Results with Different Batch Sizes.

of using preprocessing, DNN compression, and combination LEONARD to store the same provenance graphs from trace datasets. We also collect the same results for Linux datasets to show the impacts of different provenance graph sizes to individual components. The results are shown in Figure 7. Figure 7(a) and Figure 7(c) show the disk usage and time costs for storing graphs in trace datasets. Figure 7(b) and Figure 7(d) present the results on Linux datasets.

From the disk size usage results on trace datasets (Figure 7(a)), we find that both preprocessing and DNN compression can achieve good compression results. Combining them and using LEONARD lead to optimal compression results. Specifically, the disk usage of using preprocessing, DNN and LEONARD is 284.08 MB, 313.75 MB and 164.95 MB, respectively. For the time costs on trace datasets, preprocessing is impervious. Because preprocessing consists of purely string matching operations, it does not require training and is pretty fast. DNN compression and LEONARD are more time-consuming than preprocessing because of the additional training process. When comparing DNN compression with LEONARD, applying preprocessing in LEONARD significantly reduces the time costs.

When we inspect the impacts of provenance graph sizes on each component in Figure 7(b) and Figure 7(d), the time costs of all components increase almost linearly with the increase of dataset sizes. Notice that Figure 7(b) shows that DNN compression and LEONARD require similar space when storing graphs. The reason is that even though the preprocessing in LEONARD reduces the total size of processed data compared to DNN compression, it corrupts some natural semantic information, leading to low accuracy. However, preprocessing speeds up training as shown in Figure 7(d).

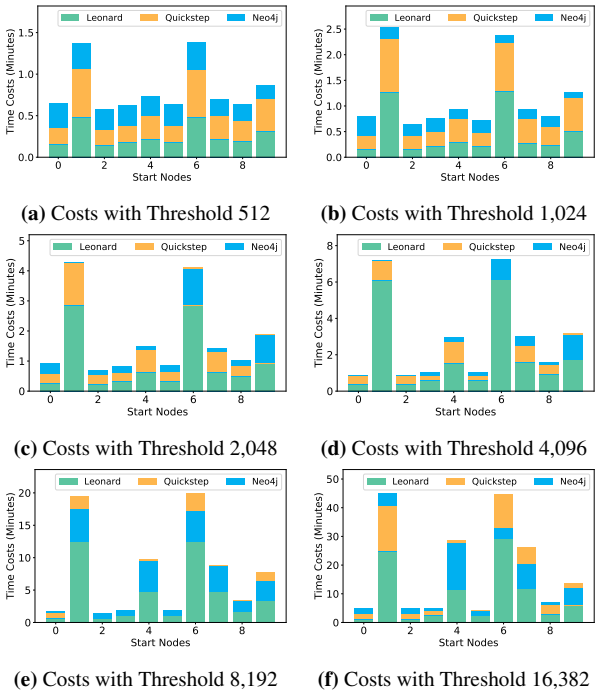
3.4.3 Different Models.

To measure the impacts of models, we consider two factors: model architectures and model sizes. We design and evaluate four different models, including two LSTM models (we use M1 to refer to the small one and M2 to refer to the large one) and two GRU models (M3 for the small one and M4 for the large one), to help understand how different models affect the performance of LEONARD. M1 is our default model. The sizes of each model are 0.14 MB, 0.29 MB, 0.14 MB, and 1.52 MB. We measure the total size, model size, and time costs of using different models to store the provenance graph in L1. The results are summarized in Figure 8. The *y*-axis on the left shows the total time costs used to store the graph and the *y*-axis on the right reflects the disk usage.

The results show that using a small model to store the provenance graph is faster than using a large model of its kind. For example, the time costs of M1 are lower than M2. This is because fewer parameters need to be trained in smaller models, leading to faster training. Moreover, we observe that using larger models can reduce the size of other components (i.e., calibration tables) but increase the final size of storing graphs. Because large models can learn more contextual data information, achieving higher training accuracy and reducing the number of mispredictions. However, the size of the model itself can be very large compared to the total size of the graph, leading to high costs of storing the model and graph. Therefore, choosing a model is a trade-off between time costs, model size, and final disk usage. For our datasets, a smaller model can achieve good results. We also observe that, although the model size and final disk usage of small models (M1 and M3) are similar, the time costs of M1 are lower than the costs of M3.

3.4.4 Batch Size in Training.

To study the impacts of batch sizes on LEONARD, we use different batch sizes from 256 to 65,536 to train DNN models on the provenance graph in the L1 dataset. Then, we collect



(a) Costs with Threshold 512 **(b)** Costs with Threshold 1,024

(c) Costs with Threshold 2,048 **(d)** Costs with Threshold 4,096

(e) Costs with Threshold 8,192 **(f)** Costs with Threshold 16,382

Figure 10: Efficiency of Querying with Different Thresholds

the time costs of model training, the time costs of storing provenance graphs, the training accuracy, and the disk usage of storing provenance graphs. We show the results in Figure 9. Figure 9(a) shows the time costs of training models with different batch sizes and the time costs of storing provenance graphs. Figure 9(b) presents the model accuracy and the total disk usage for storing the provenance graph.

Overall, the time costs of training and storing the graph decrease when batch size increases, and the costs finally reach the saturation points (when batch size is larger than 4,096). The results reveal that increasing batch size could significantly speed up the training process, but the improvement is not linear. The training speed does not significantly increase after a certain threshold. This finding is consistent with existing work [23]. In addition, the training accuracy of models and the disk usage for storage are insensitive to changes in the batch size. Therefore, considering the time costs almost do not decrease when the batch size is larger than 4,096, and the final disk size for storage is stable, we set batch size 4,096 as the default batch size for model training.

3.4.5 Threshold in Querying.

As mentioned in §3.3, we constrain the final graph size. We evaluate the effects of different thresholds (i.e., graph size) configured in the querying process by measuring the querying costs on the graph with different threshold settings. Specifically, to ensure the fairness of comparison, we choose ten nodes from the T1 dataset and evaluate the costs of searching their descendants with the threshold from 512 to 16,382. We ensure that the number of descendants of each picked start node is larger than the largest threshold of 16,382. The results are shown in Figure 10. The time costs of different systems are overlapped in these figures. From the result, we observe that LEONARD always outperforms other systems under different threshold settings, showing the efficiency of LEONARD. Moreover, when the threshold increases, the time costs of all systems increase. This is reasonable because we need to search more events and sort their importance values, which is more complicated.

3.4.6 CPU Performance.

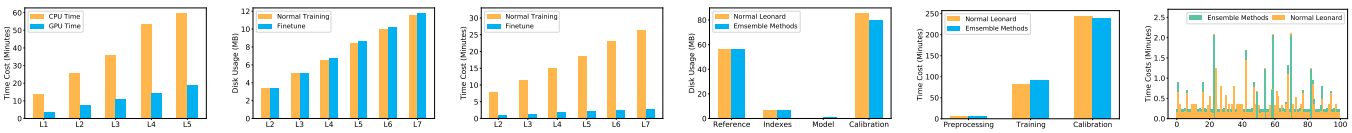
We measured the performance of LEONARD model training on CPU devices. The results are shown in Figure 11(a). The x-axis shows the dataset names, and the y-axis demonstrates the time costs. We observe that the costs of training on CPUs takes 3.37x time than training on GPU. Considering the cold data target scenario, running LEONARD on CPU is acceptable.

3.4.7 Model Reusing and Ensemble Methods.

As introduced in §2.2, LEONARD can reuse models by fine-tuning to improve the efficiency on similar data distribution. LEONARD can also apply ensemble methods to handle new data efficiently. To show that, we evaluate the performance of fine-tuning and ensemble methods. Specifically, we train a model on a small dataset (L1) and fine-tune new models on other datasets (L2 to L7). We show the final disk usage (Figure 11(b)) and training time costs (Figure 11(c)). For the ensemble method, we divide the T2 dataset into five parts to emulate five caches arriving at different times. We first train LEONARD on the first block of data and iteratively update LEONARD by processing the subsequent four parts of data. The results are shown in Figure 11(d), Figure 11(e) and Figure 11(f), respectively.

We observe that fine-tuning significantly reduces the time costs without affecting the final disk usage. The disk usage is only slightly higher than that of the original training. But the training and total time costs are reduced to 11.62% and 54.85% of original training, respectively. Therefore, fine-tuning can reduce the overhead of LEONARD further.

We also find that applying ensemble methods to handle new data can achieve similar effectiveness and efficiency as the original training. Specifically, the disk usage of ensemble methods is similar to the costs of original LEONARD. As we use more models to store the data, the model size is larger. Meanwhile, using more models to remember the same amount of data decreases the misprediction rates, making the size of the calibration table smaller. For the efficiency, the time cost of storing the graph with ensemble methods is slightly higher. As we need to manage data and different models, the scheduling poses additional overhead. However, the overhead



(a) Training (CPU) **(b)** Disk Usage (Finetune) **(c)** Training (Finetune) **(d)** Disk Usage (Ensemble) **(e)** Storage (Ensemble) **(f)** Querying (Ensemble)

Figure 11: Results of Model Reusing and Ensemble Methods

is small. Moreover, here we measure the total time costs for all models. When processing multiple models in parallel and measure the costs of each model, they will take less time. For example, it takes only around 20% time when we handle the five models with five different processes separately.

3.5 Software Version and Parallel Processing.

LEONARD is previously implemented in CUDA 11.0.0. To show the effects of different software version and other processing, we upgraded to CUDA 11.0.4 and observed better performance. Also, when handling the calibration of large datasets (i.e., trace datasets), LEONARD ran processes in parallel. We observe that parallel processing can lead to heavy I/O overhead, and removing parallel handling can reduce the time costs of calibration. To show this, we compare the time cost differences. The results are shown in Figure 12. The changes in training and calibration time costs are summarized in Figure 12(a) to Figure 12(j). The querying costs are included in Figure 12(k) and Figure 12(l). Specifically, Figure 12(l), Figure 12(b), Figure 12(e), and Figure 12(f) show the changes of LEONARD on different datasets. Figure 12(d), Figure 12(d), Figure 12(d), and Figure 12(d) show the performance when only using DNN to store the graph. Figure 12(i) and Figure 12(j) show the changes of using different batch sizes in model training. Figure 12(k) and Figure 12(l) show the average querying costs on five trace datasets and the costs under different graph size constraints.

Training time decreases in all cases because the improved latest CUDA has better support for GPU. We also notice that the calibration costs keep unchanged on small Linux datasets (the costs slightly increase in Figure 12(f)) and are decreased on large trace datasets or pure DNN processing task. The reason is that calibration of large datasets (trace dataset and all datasets used by DNN) is time-consuming and imposes heavy overhead on the system. Parallel processing does not work for that cases because it causes resource competition. Therefore, we removed such a setup of storing multiple large datasets in parallel (consistent with settings on small Linux datasets and also databases), reducing the overhead caused by competing resources and achieving higher efficiency. For the time cost, we observe that the cost of queries remains almost unchanged since the querying was not running on CUDA, and there is no competition issue.

We conclude that software versions and configurations can affect the efficiency of LEONARD. Optimizing the code and improving the system could make LEONARD faster.

Table 2: Results of Forensics Analysis from 5 Start Nodes.

Queries	LEONARD		Databases		Graph Match
	# Nodes	# Edges	# Nodes	# Edges	
1	1867	2231	1867	2231	✓
2	1878	2219	1878	2219	✓
3	1339	2759	1339	2759	✓
4	1892	2206	1892	2206	✓
5	938	3159	938	3159	✓

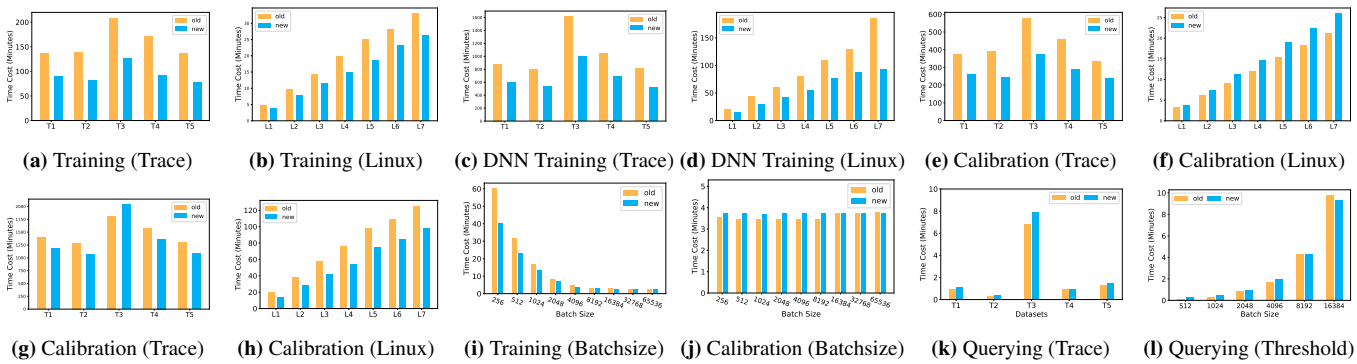
3.6 Integrity of Provenance Graphs

LEONARD is designed for effective storage and efficient investigation of provenance graphs. LEONARD guarantees the integrity of stored data by design with the calibration mechanism. To explore whether LEONARD supports forensics analysis based on the provenance graph, we check whether LEONARD can return the same query results as databases. Specifically, we show the number of nodes and edges for the first five queries on the T1 dataset conducted in Figure 5. Notice that we stop the searching when the number of edges and nodes is larger than the threshold of 4,096. LEONARD can still search for new events (the event can contain 1 or 2 unvisited nodes). Therefore, the final number of the result could be up to 4,098. Table 2 summarizes the results, including the number of nodes and edges of the obtained descendant graphs using databases and LEONARD (from columns 2 to 5 in Table 2). As the table indicates, LEONARD can return the same graphs as databases and support the forensics analysis. The results show that LEONARD can support forensic analysis. Due to the page limits, the complete results of 500 queries on T1 to T5 datasets are in our project page[2].

4 Discussion and Future Work

LEONARD is a storage system, and it is complimentary with provenance data/graph compression and reduction [18, 42, 43, 66]. In the best case (the model accuracy is 100%), the model memorizes all the data without mispredictions. The storage costs are the costs of storing a small model and some indexes. In the worst case (the accuracy of model prediction is 0), LEONARD stores the raw data, i.e., databases with no

[2]https://anonymous.4open.science/r/query_result-5468/README.md



(a) Training (Trace) **(b)** Training (Linux) **(c)** DNN Training (Trace) **(d)** DNN Training (Linux) **(e)** Calibration (Trace) **(f)** Calibration (Linux)

(g) Calibration (Trace) **(h)** Calibration (Linux) **(i)** Training (Batchsize) **(j)** Calibration (Batchsize) **(k)** Querying (Trace) **(l)** Querying (Threshold)

Figure 12: Comparison of Time Costs. We show the changes on model training, model calibration, and querying. Figure 12(a) to Figure 12(d) show changes on training time costs. Figure 12(e) to Figure 12(h) show time costs on calibration. Figure 12(i) and Figure 12(j) show the costs under different batch size settings. Figure 12(k) and Figure 12(l) demonstrate the costs on querying. The x-axis shows the dataset names or different settings. y-aixs shows the time costs.

compression. LEONARD spends much time on storing the data caused by slow model inference, which can be alleviated by DNN acceleration techniques (e.g., AI chips like NVIDIA H100 tensor core GPUs are reportedly 4.5x faster than previous-generation GPUs [8]. TensorRT can achieve 6X faster inference and JAX on the GPU [3] is about 30x faster than numpy on GPU [2]).

LEONARD lacks many database features, such as transactions. It does not guarantee the ACID (atomicity, consistency, isolation, durability) of a sequence of operations. LEONARD targets a typical *write once read multiple times* task. In our scenario, the provenance graph itself should not be changed due to integrity requirements. Without multiple writers, there is no need to support transactions. When writing to the database, we always verify the correctness and completeness of the operation. Returned results are correct as long as the execution integrity is maintained. LEONARD can also be extended to other similar scenarios.

Extending LEONARD to general data is non-trivial and raises new challenges, including reducing improving the efficiency, supporting for complex data operations, and redesign for domain-specific data reduction. The time costs of storage is high and the update operation in LEONARD is inefficient (the model will be re-trained whenever the stored data changes). Provenance graph data is append-only thus does not need to be updated. However, the inefficient update operation is a problem in other scenarios, such as storing user relation graphs in recommendation systems. When applying to different data, the reduction rules in the preprocessing also need to be redesigned.

5 Related Work

Intrusion Detection. Many intrusion detection methods use predefined normal behavior patterns to detect attacks. Some work defines the normal behavior of processes with a sequence of system calls [20, 70, 71]. Along this line of work, Tandon et al. [65] and Sekar et al. [62] propose to use additional syscall information and sequence information, such as arguments in syscalls and loop structure, to improve the detection. Provenance graphs provide a more intuitive way to present the detailed historical information of system execution by converting system calls to a connected graph, reducing the false positives in detection. NoDoze [29] analysis historical logs and gives rarely occurring events high anomaly values. Then, it updates the anomaly values of each event based on its casually related events in the provenance graph to get a more accurate estimation. Different from NoDoze, Unicorn [26] does not give events anomaly values. It learns and defines several event sequences in the benign provenance graph as normal behavior models to detect abnormal behaviors.

Attack Investigation. Besides intrusion detection, many works apply provenance graphs to investigate attacks [6, 54, 74]. Backtracing the provenance graph from an attack symptom can help analysts understand the root cause of the attack [40] and forward-searching to find the consequences [45]. Besides NoDoze and other works [53] try to understand the attack from system logs, UIScope [73] analysis attacks in GUI applications with causality analysis on both UI elements/events. CLARION [11], PRovINTENT [68], PicoSDN [69] and Prov-Trust [39] provide solutions to generate provenance graphs for microservice deployments, intent-based networking, software-defined networking attacks and SGX-based systems, respectively. We notice that most of the existing work based on provenance graphs requires the system to collect the provenance graph and system logs for a long time, resulting in very high storage overhead.

Provenance Graph Compression. Existing work removes redundant edges and nodes in the graph to reduce storage overhead. LogGC [43] finds that temporary files are usually isolated in the provenance graph. Therefore, they can be garbage collected. Lee et al. [42] propose to use execution partition to simplify the causalities in long-running processes. Protracer [49] further reduces both the runtime overhead and stor-

age overhead. Although these lossy compression methods can compress logs and graphs, they are designed for specific tasks and lose important information. Recently, researchers focused on lossless compression for provenance graphs. SEAL [18] merges repeated fields of events and reduces the costs of storing timestamps. LEONARD does not only reduce the content redundancies such as repeated field but also reduces the correlation redundancies with DNN models. Moreover, existing graph compression methods try to compress the graph during the collection. LEONARD focuses on a different stage, i.e., the storage process; they are compatible. In the provenance graph collection, graph compression methods identify and remove useless elements of the graph for specific tasks. Then, they can use LEONARD or databases to store compressed graphs.

Database Compression. We also notice that there are some compression methods designed for databases [13, 60]. Databases that store the data in a column-oriented way can potentially be compressed [36, 52]. In this work, we explored the compression with Quickstep [52]. Applying compression on databases can reduce the time costs of reading operations because smaller amounts of data need to be moved from disk to memory. However, applying compression requires additional time for decompression. We investigate more effective compression and storage methods for provenance graphs by fully understanding the redundancies in the graph.

6 Conclusion

In this paper, we present a novel research system LEONARD, which uses Deep Neural Networks (DNNs) as the provenance graph storage system. It features high storage efficiency and query efficiency. Compared with existing solutions based on relational databases and graph databases, it learns a compact data representation and supports parallel data querying, incurring less storage and runtime overhead. Our evaluation of 12 datasets shows promising results, reducing the space overhead by 25.90 times and boosting up to 99.6% queries.

Acknowledgments

We thank the anonymous reviewers for their constructive comments. This material is based upon work supported by the National Science Foundation under Grant No. 2152908, No. 2212629, and No. 2238847.

References

[1] Neo4j graph database platform. https://neo4j.com/, 2021.
[2] Jax reference documentation. https://jax.readthedocs.io/en/latest/, 2022.
[3] Nvidia tensorrt. https://developer.nvidia.com/tensorrt, 2022.
[4] Martín Abadi and Paul Barham et al. Tensorflow: A system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2016.
[5] Abdulellah Alsaheel, Yuhong Nan, Shiqing Ma, Le Yu, Gregory Walkup, Z Berkay Celik, Xiangyu Zhang, and Dongyan Xu. Atlas: A sequence-based learning approach for attack investigation. In *USENIX Security Symposium*, 2021.
[6] Adam Bates, Dave Tian, Kevin R. B. Butler, and Thomas Moyer. Trustworthy whole-system provenance for the linux kernel. In *USENIX Security*, 2015.
[7] Bin Bi, Chenliang Li, Chen Wu, Ming Yan, Wei Wang, Songfang Huang, Fei Huang, and Luo Si. Palm: Pre-training an autoencoding&autoregressive language model for context-conditioned generation. *arXiv:2004.07159*, 2020.
[8] NVIDIA Blog. Nvidia hopper sweeps ai inference benchmarks in mlperf debut. https://blogs.nvidia.com/blog/2022/09/08/hopper-mlperf-inference/, 2022.
[9] Kris Cao and Marek Rei. A joint model for word embedding and word morphology. In *Proceedings of the 1st Workshop on Representation Learning for NLP, Rep4NLP@ACL*, 2016.
[10] Suganya Chandrababu and Dhundy R. Bastola. Comparative analysis of graph and relational databases using herbmicrobedb. In *IEEE International Conference on Healthcare Informatics Workshops, ICHI Workshops*, 2018.
[11] Xutong Chen, Hassaan Irshad, Yan Chen, Ashish Gehani, and Vinod Yegneswaran. Clarion: Sound and clear provenance tracking for microservice deployments. In *USENIX Security*, 2021.
[12] François Chollet et al. Process monitor. https://keras.io, 2015.
[13] Gordon V. Cormack. Data compression on a database system. *Commun. ACM*, 28(12):1336–1342, 1985.
[14] Balázs Csanád Csáji et al. Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, 24(48):7, 2001.
[15] Cve.mitre.org. Cve - cve-2021-22893. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-22893, 2022.
[16] Hailun Ding, Shenao Yan, Juan Zhai, and Shiqing Ma. ELISE: A storage efficient logging system powered by redundancy reduction and representation learning. In *USENIX Security Symposium*, 2021.
[17] En.Wikipedia.Org. 2020 united states federal government data breach. https://en.wikipedia.org/wiki/2020_United_States_federal_government_data_breach, 2021.
[18] Peng Fei, Zhou Li, Zhiying Wang, Xiao Yu, Ding Li, and Kangkook Jee. Seal: Storage-efficient causality analysis on enterprise logs with query-friendly compression. In *USENIX Security Symposium*, 2021.
[19] Fireeye. [report] m-trends. https://content.fireeye.com/m-trends/rpt-m-trends-2021, 2021.
[20] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for unix processes. In *IEEE Symposium on Security and Privacy, SP*, 1996.
[21] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *SIGMOD*, 2018.
[22] Ashish Gehani and Dawood Tariq. SPADE: support for provenance auditing in distributed environments. In *ACM/IFIP/USENIX International Middleware Conference*, volume 7662 of *Lecture Notes in Computer Science*, pages 101–120. Springer, 2012.
[23] Noah Golmant, Nikita Vemuri, Zhewei Yao, Vladimir Feinberg, Amir Gholami, Kai Rothauge, Michael W. Mahoney, and Joseph Gonzalez. On the computational inefficiency of large batch sizes for stochastic gradient descent. 2018.
[24] Mohit Goyal, Kedar Tatwawadi, Shubham Chandak, and Idoia Ochoa. Deepzip: Lossless data compression using recurrent neural networks. In *Data Compression Conference, DCC*, 2019.
[25] Gzip.Org. The gzip home page. https://www.gzip.org/., 2021.
[26] Xueyuan Han, Thomas F. J.-M. Pasquier, Adam Bates, James Mickens, and Margo I. Seltzer. Unicorn: Runtime provenance-based detector for advanced persistent threats. In *NDSS*, 2020.
[27] Xueyuan Han, Thomas F. J.-M. Pasquier, and Margo I. Seltzer. Provenance-based intrusion detection: Opportunities and challenges. In *USENIX Workshop on the Theory and Practice of Provenance*, 2018.
[28] Wajih Ul Hassan, Adam Bates, and Daniel Marino. Tactical provenance analysis for endpoint detection and response systems. In *IEEE Symposium on Security and Privacy, SP*, 2020.
[29] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. In *NDSS*, 2019.
[30] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997.
[31] Kurt Hornik, Maxwell B. Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.
[32] Md Nahid Hossain, Sadegh M. Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R. Sekar, Scott D. Stoller, and V. N. Venkatakrishnan. SLEUTH: real-time attack scenario reconstruction from COTS audit data. In *USENIX Security Symposium*, 2017.
[33] Md Nahid Hossain, Sanaz Sheikhi, and R. Sekar. Combating dependence explosion in forensic analysis using alternative tag propagation semantics. In *IEEE Symposium on Security and Privacy, SP*, 2020.
[34] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
[35] Ibm.Com. Cost of a data breach report 2021. https://www.ibm.com/security/data-breach, 2021.

[36] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[37] Amir Ilkhechi, Andrew Crotty, Alex Galakatos, Yicong Mao, Grace Fan, Xiran Shi, and Ugur Çetintemel. Deepsqueeze: Deep semantic compression for tabular data. In *SIGMOD*, 2020.

[38] Muhammad Adil Inam, Yinfang Chen, Akul Goyal, Jason Liu, Jaron Mink, Noor Michael, Sneha Gaur, Adam Bates, and Wajih Ul Hassan. Sok: History is a vast early warning system: Auditing the provenance of system intrusions. In *IEEE Symposium on Security and Privacy*, 2022.

[39] Nesrine Kaaniche, Sana Belguith, Maryline Laurent, Ashish Gehani, and Giovanni Russello. In *International Joint Conference on e-Business and Telecommunications, ICETE*, 2020.

[40] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *SOSP*, 2003.

[41] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *SIGMOD*, 2018.

[42] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *NDSS*, 2013.

[43] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. Loggc: garbage collecting audit log. In *ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2013.

[44] Linux.Die.Net. Auditd. https://linux.die.net/man/8/auditd, 2021.

[45] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *NDSS*, 2018.

[46] Shiqing Ma, Kyu Hyung Lee, Chung Hwan Kim, Junghwan Rhee, Xiangyu Zhang, and Dongyan Xu. Accurate, low cost and instrumentation-free security audit logging for windows. In *Annual Computer Security Applications Conference*, 2015.

[47] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela F. Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. Kernel-supported cost-effective audit logging for causality tracking. In *USENIX ATC*, 2018.

[48] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. MPI: multiple perspective attack investigation with semantic aware execution partitioning. In *USENIX Security Symposium*, 2017.

[49] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS*, 2016.

[50] Mandiant.com. Highly evasive attacker leverages solarwinds supply chain to compromise multiple global victims with sunburst backdoor | mandiant. https://www.mandiant.com/resources/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor, 2022.

[51] Sadegh Momeni Milajerdi, Rigel Gjomemo, Birhanu Eshete, R. Sekar, and V. N. Venkatakrishnan. HOLMES: real-time APT detection through correlation of suspicious information flows. In *IEEE Symposium on Security and Privacy*, 2019.

[52] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. Quickstep: A data platform based on the scaling-up approach. *VLDB Endow.*, 11(6):663–676, 2018.

[53] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, Shiqing Ma, Fei Wang, Zhiwei Zhang, Luo Si, Xiangyu Zhang, and Dongyan Xu. HERCULE: attack story reconstruction via community discovery on correlated log graph. In *Annual Conference on Computer Security Applications, ACSAC*, 2016.

[54] Devin J. Pohly, Stephen E. McLaughlin, Patrick D. McDaniel, and Kevin R. B. Butler. Hi-fi: collecting high-fidelity whole-system provenance. In *Annual Computer Security Applications Conference, ACSAC*, 2012.

[55] PurpleSec. 2021 cyber security statistics trends & data. https://purplesec.us/resources/cyber-security-statistics/#Cyberattacks, 2021.

[56] PurpleSec. 2021 accellion data breach: What happened & who was impacted? https://purplesec.us/accellion-data-breach-explained/, 2022.

[57] PurpleSec. Kaseya ransomware attack explained: What you need to know. https://purplesec.us/kaseya-ransomware-attack-explained/, 2022.

[58] PurpleSec. Saudi aramco $50 million data breach explained. https://purplesec.us/saudi-aramco-data-breach-explained/, 2022.

[59] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[60] Gautam Ray, Jayant R. Haritsa, and S. Seshadri. Database compression: A performance enhancement tool. In *COMAD*, 1995.

[61] Marko A. Rodriguez. The gremlin graph traversal machine and language (invited talk). In *Symposium on Database Programming Languages*, 2015.

[62] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, 2001.

[63] Servethehome. Firefox is eating your ssd - here is how to fix it. https://www.servethehome.com/firefox-is-eating-your-ssd-here-is-how-to-fix-it/, 2016.

[64] Windows Sysinternals. Process monitor. https://docs.microsoft.com/en-us/sysinternals/downloads/procmon, 2021.

[65] Gaurav Tandon and Philip K. Chan. On the learning of system call attributes for host-based anomaly detection. *Int. J. Artif. Intell. Tools*, 2006.

[66] Yutao Tang, Ding Li, Zhichun Li, Mu Zhang, Kangkook Jee, Xusheng Xiao, Zhenyu Wu, Junghwan Rhee, Fengyuan Xu, and Qun Li. Nodemerge: Template based efficient data reduction for big-data causality analysis. In *SIGSAC Conference on Computer and Communications Security, CCS*, 2018.

[67] Silke Trißl and Ulf Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD*, 2007.

[68] Benjamin E. Ujcich, Adam Bates, and William H. Sanders. Provenance for intent-based networking. In *IEEE Conference on Network Softwarization*, 2020.

[69] Benjamin E. Ujcich, Samuel Jero, Richard Skowyra, Adam Bates, William H. Sanders, and Hamed Okhravi. Causal analysis for software-defined networking attacks. In *USENIX Security*, 2021.

[70] Andreas Wespi, Marc Dacier, and Hervé Debar. Intrusion detection using variable-length audit trail patterns. In *Recent Advances in Intrusion Detection, Third International Workshop, RAID*, 2000.

[71] Andreas Wespi, Hervé Debar, Marc Dacier, and Mehdi Nassehi. Fixed- vs. variable-length patterns for detecting suspicious process behavior. *J. Comput. Secur.*, 2000.

[72] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. High fidelity data reduction for big data security dependency analyses. In *ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[73] Runqing Yang, Shiqing Ma, Haitao Xu, Xiangyu Zhang, and Yan Chen. Uiscope: Accurate, instrumentation-free, and visible attack investigation for GUI applications. In *NDSS*, 2020.

[74] Le Yu, Shiqing Ma, Zhuo Zhang, Guanhong Tao, Xiangyu Zhang, Dongyan Xu, Vincent E. Urias, Han Wei Lin, Gabriela F. Ciocarlie, Vinod Yegneswaran, and Ashish Gehani. Alchemist: Fusing application and audit logs for precise attack provenance without instrumentation. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

A Appendix

A.1 Primitives of LEONARD

We provide a list of basic query primitives of LEONARD in List 1. Some of them are simple, e.g., `ngetd()` and `edgetd()` get detailed node and edge information, respectively. Others implement common functionalities, e.g., `bgetn()` and `bgete()` perform backward tracking from a given node or an edge.

Example: Backward tracking with constrained depth. We show how to use the given primitives to perform backward tracking in algorithm 1. Notice that typical provenance graphs are large, and we limit the search depth by using an input parameter M_d. In the beginning, we first initialize our search depth to be 0 (line 1). Then, we start our search by initializing the graph (lines 2 and 3). Specifically, we first get detailed information on node m, the attack symptom object, and then add it to the graph with an unvisited flag. The main body of this backward tracking algorithm is a loop (lines 4 to 15), which performs a BFS (Breadth-first search). For each unvisited node in the graph (line 5), we first get all its pre-nodes and incoming edges (line 6). For each node, we get its information, mark it as unvisited and then add it to the graph (lines 7 to 10). For each edge, we get its information and add it to the graph (line 12). After visiting all nodes and edges, we mark the current node n as seen (line 14) and move to the next depth by increasing it by 1 (line 16). When we reach the maximal depth, we return the graph G. Notice that this example is simplified to demonstrate how to interact with LEONARD via its query engine primitives. Many details are

```
# Get source vertices and edges of a given node n
# Parameter: n, a given vertex/node
# Return: list of vertices and edges pointing to n
def bgetn (n)
# Get edges of a given edge e (backward)
# Parameter: e, a given edge
# Return: list of vertices and edges pointing to e
def bgete (e)
# Get outgoing vertices of a given node n
# Parameter: n, a given vertex/node
# Return: a list of vertices start from n
def fgetn (n)
# Get edges of a given node n (forward)
# Parameter: n, a given vertex/node
# Return: a list of edges start from n
def fgete (n)
# Get detailed informaion for node n
# Parameter: n, a given vertex/node
# Return: detailed information of n from the DNN
def ngetd (n)
# Get detailed informaion for edge e
# Parameter: e, a given edge
# Return: detailed information of e from the DNN
def egetd (e)
```

Listing 1: Primitives of Query Engine

Algorithm 1: Backward Tracking from Symptoms

Input: m: symptom events (e.g., malware); M_d: the maximal search depth.
Output: G: graph, G_e and G_v are edges and vertices.

$d \leftarrow 0$;
$m.visited \leftarrow false$; $m \leftarrow$ `ngetd` (m);
$G_v \leftarrow \{m\}$;
while $(\ d \leq M_d\ )$ **do**
  for $(n$ ***in*** $[n$ ***in*** G_v ***if*** $n.visited = false])$ **do**
    $N, E \leftarrow$ `bgetn` (n);
    for $(v$ ***in*** $N)$ **do**
      $v \leftarrow$ `ngetd` (v); $v.visited \leftarrow false$;
      $G_v \leftarrow G_v \cup v$
    end
    for $(e$ ***in*** $E)$ **do**
      $e \leftarrow$ `egetd` (e); $G_e \leftarrow G_e \cup e$
    end
    $n.visited = true$
  end
  $d \leftarrow d + 1$
end
return G_e, G_v

omitted, such as maintaining the visited status using queues, handling loops in the provenance graphs, etc.

A.2 LEONARD Running Example

In this section, we show how LEONARD works by using a simplified example of a single event: `Firefox` writes data to a file `a.txt` at `155`. Later, analysts backtrack from the file `a.txt` and try to find out which process wrote to the file. Figure 13 shows the process of building the storage, and Figure 14 shows the query execution. We refer to these two graphs in the following sections if not specified.

A.2.1 Building Storage

Input: The input to LEONARD is a provenance graph (① in Figure 13). In this graph, rectangles denote processes, and ovals denote files. The edge represents the write operation, the direction follows the information flow, and its annotation shows its timestamp.

Step 1: Decoupling. LEONARD first decouples the graph and converts each edge/vertex to a textual representation, represented in JSON format (②). Every edge and vertex will contain a unique identifier, content values (i.e., `Firefox`, `a.txt`, and `(E0, E1)`), types (e.g., `Process`, `File`, and `Write`), and necessary annotations (e.g., `timestamp`). Here, we use `(source, destination)` to denote an edge using its source and destination identifiers.

Step 2: Content Reduction. There are two types of reductions. First, for monotonous values, such as timestamps (`155`), LEONARD replaces them by using the base and offset. Rule 2 in the reference table records such reductions and the base value. Thus, the timestamp in `E0` is replaced by `5`, according to this rule. Second, for redundant values, LEONARD replaces them with shorter numerical values. In this case, `File` is replaced by `0`, `Write` is replaced by `1`, and `Process` is replaced by `2`. The serial number of the rule represents the order of application. For example, R0 is applied before R1. In Step B of Figure 13, we use arrows to show how individual entries in the JSON entry are converted to a reduced version and ④ shows the rules we used.

Step 3: Content Indexing. LEONARD builds both vertex and edge indexes, as shown in ⑤ and ⑦, respectively. Our indexes are secondary indexes containing the content values of vertexes and types of edges, which are neither ordering nor key fields. We chose such indexes because they are the most frequently used fields during the investigation. To simplify our design, we just use the unique identifier as the prompt (i.e., k in index $\{c,k\}$, it is an input to the DNN model that can recover all information of a node) when building the indexes. For the vertex index, we have two entries `<V0, 0, a.txt, 0>` and `<V1, 0, 0, 2>` as shown in ⑥. Recall that its format is $\{c,k\}$, where c is the content and k is prompt. Here, we have two entries. `V0` is the main indexing for the vertex `V0`, and `a.txt` is the secondary index. Similarly, the edge index

contains one entry, where `(V1, V0)` denotes the edge source and destination and `E0` is the prompt.

Step 4: Training and Calibration. LEONARD uses a DNN model (⑧) to memorize all the graph data, and when needed, we feed a prompt (i.e., an identifier) to the model and get complete vertex/edge information. This is a text-completion task, where the model tries to complete a sentence (i.e., the whole vertex/edge entry) for given words (i.e., prompt). LEONARD vectorizes all data by char2vec [9] and then trains the model by using next-character prediction, a typical method for such NLP tasks. When the model stabilizes, the accuracy can still be lower than 100%. For our example, the ninth value in entry `V1` is wrong. We record such entries in the calibration table (⑨), which contains the position `9` and the ground truth `0`.

A.2.2 Query Execution

The querying execution (described in algorithm 1) consists of two kinds of primitives. Searching primitives such as `fgetn()` used to tracing the graph from a given vertex or edge, and recovery primitives like `ngetd()` used to recover the graph details. In this section, we show an example of each kind. Specifically, we illustrate how to use `fgetn()` to track all descendant nodes from a node named as `Firefox` and use `ngetd()` to recover its details by the given identifier `V1`.

•`fgetn(`*`Value: Firefox`*`)`. `fget` in `fgetn` indicates the searching direction is `Forward` and `n` implies the beginning of searching is a node. *`Value: Firefox`* shows the secondary index is `Value`, and the concrete value is `Firefox` (translated to `0` according to the reference table). When searching, LEONARD first searches the vertex indexes, finds the vertex whose value is `Firefox`, and obtains the identifier (`V1` in the example). Then, LEONARD searches edges starting from `V1` and continue the searching process until no new entries can be added. Finally, LEONARD returns the identifiers of all obtained vertices and edges (③).

•`ngetd(`*`V1`*`)`. The input `V1` is an identifier of a vertex, which is usually obtained in searching primitives. Firstly, LEONARD converts `V1` to a vector `<9, 1>` by char2vec as the input of the DNN model. The model predicts the following characters of `V1`, and the calibration table revises the mispredictions during this process. As shown by the example, the model outputs `<[9, 1], 0, 1, 2>`, the ninth character then is revised as `0` by the calibration table, and the final output is revised to `<[9, 1], 0, 0, 2>`. LEONARD converts the numerical outputs as text representation by using vec2char (a reverse of char2vec) and gets `<V1, 0, 0, 2>`. LEONARD further searches the keys represented by the first `0` and get the keys `<Value, Type>`. LEONARD then translates frequent word codes to concrete words and obtains `<V1, Value: Firefox, Type: Process>` by using the reference table. For example, `0` is identified as the value of field `Value` and maps to the concrete string `Firefox`. Finally, LEONARD outputs the recovered vertex (⑥).

Table 3: Overview of Datasets

Name	Collector	Log Size	Graph Size	# Edges	# Vertices
L1	Auditd	200 MB	17.95 MB	70,746	46,951
L2	Auditd	400 MB	35.97 MB	141,773	94,131
L3	Auditd	600 MB	53.44 MB	209,001	139,017
L4	Auditd	800 MB	69.69 MB	271,947	182,651
L5	Auditd	1,000 MB	89.40 MB	348,090	235,592
L6	Auditd	1,200 MB	104.86 MB	407,871	276,809
L7	Auditd	1,400 MB	122.22 MB	475,251	322,970
T1	DTrace	6.49 GB	1.42 GB	3,838,235	2,104,167
T2	DTrace	6.48 GB	1.30 GB	3,502,032	2,101,552
T3	DTrace	6.83 GB	2.23 GB	5,188,353	2,615,885
T4	DTrace	6.49 GB	1.62 GB	4,123,248	2,392,776
T5	DTrace	6.56 GB	1.29 GB	3,626,617	1,822,855

A.3 Datasets and Models

The details of our datasets are listed in Table 3. The trace datasets from the trace group (referred to as character T plus indexes, e.g., T1) are collected by DTrace on the Linux system and wildly used to evaluate forensics analysis applications. Since each log file provided by the transparent computing program is compressed into a binary file, we show the sizes of datasets after decompression in the third column of Table 3. The raw size of the graph (in text format) is shown in the fourth column. Other Linux datasets (whose names are the character L plus the index of the dataset, e.g., L1) are used to evaluate how LEONARD and other systems perform when dataset size increases. When collecting them, we run several popular applications on an Ubuntu 16.04 experiment machine, including office software suites, browsers, etc. We use SPADE (an open-sourced provenance collection system) and built-in Linux auditd to collect the log file, generate the provenance graph and store the graph in databases. Specifically, we monitor system calls, processes (by default, all processes), specific files (e.g., `/etc/passwd`) and user account information (i.e., `euid`). We collect a 1400 MB log file and split it into files of different sizes (from 200 MB to 1400 MB). A smaller file is a subset of a larger file (e.g., L1 is a subset of L2). Since we collect the log files on the same machine and the workloads are consistent. As shown in Table 3, the number of edges and vertices of graphs grows almost linearly with the log size.

The model we used are shown in Figure 15. The LSTM model has four layers, including two LSTM layers and two dense layers. Each LSTM layer contains 32 units. The input $S = \{s_1, s_2, \ldots, s_L\}$ is the text of an index, i.e., k in §2.1.3. The model iteratively predicts the next character of S and finally recovers the details of a vertex/edge.

A.4 Querying on Compressed Databases

Compared with original databases, compressed Quickstep databases apply optimizations to reduce disk usage. To understand how these optimizations affect the querying results of databases, we measure the time costs of 500 queries (same

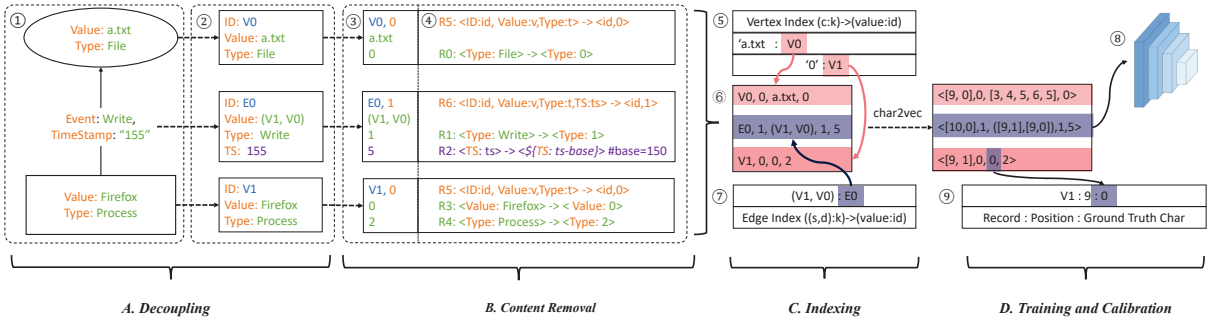


Figure 13: Storage Example.

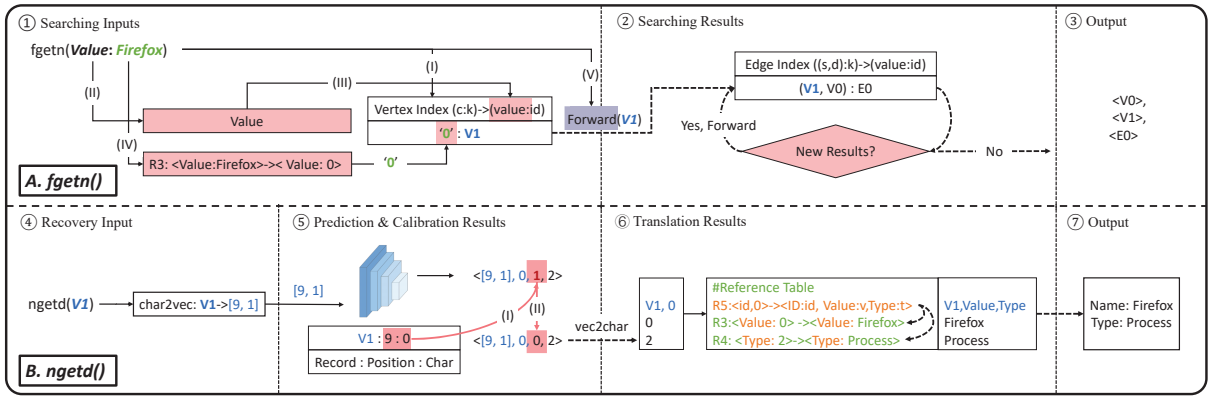


Figure 14: Querying Example.

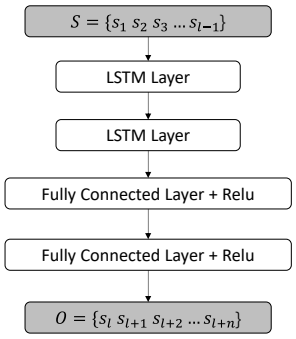
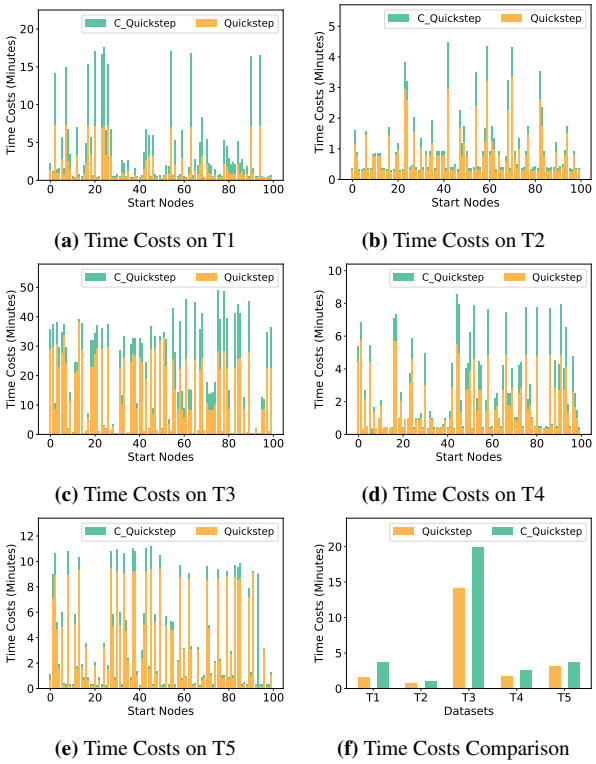


Figure 15: Model Architecture

queries in §3.3) on original Quickstep databases and compressed ones. The parameters used for the evaluation are the same as those used in Figure 5, and we show the results in Figure 16. Figure 16(a) to Figure 16(e) show the results of each queries on different provenance graphs. Figure 16(f) demonstrates the average time costs of using original databases and compressed databases on different datasets.

The results show that the compressed Quickstep databases always spend more time than original ones. This is because the built-in optimization techniques require extra time to decompress the data. Even though applying database compression reduces disk usage, it increases the time cost of queries.



(a) Time Costs on T1 **(b)** Time Costs on T2

(c) Time Costs on T3 **(d)** Time Costs on T4

(e) Time Costs on T5 **(f)** Time Costs Comparison

Figure 16: Querying with Compressed Quickstep Databases