SEAL: Storage-efficient Causality Analysis on Enterprise Logs with Query-friendly Compression

Peng Fei, Zhou Li, and Zhiying Wang, *University of California, Irvine;* Xiao Yu, *NEC Laboratories America, Inc.;* Ding Li, *Peking University;* Kangkook Jee, *University of Texas at Dallas*

https://www.usenix.org/conference/usenixsecurity21/presentation/fei

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

SEAL: Storage-efficient Causality Analysis on Enterprise Logs with Query-friendly Compression

Peng Fei
University of California, Irvine

Zhou Li
University of California, Irvine

Zhiying Wang
University of California, Irvine

Xiao Yu
NEC Laboratories America, Inc.

Ding Li
Peking University

Kangkook Jee
University of Texas at Dallas

Abstract

Causality analysis automates attack forensic and facilitates behavioral detection by associating causally related but temporally distant system events. Despite its proven usefulness, the analysis suffers from the innate big data challenge to store and process a colossal amount of system events that are constantly collected from hundreds of thousands of end-hosts in a realistic network. In addition, the effectiveness of the analysis to discover security breaches relies on the assumption that comprehensive historical events over a long span are stored. Hence, it is imminent to address the scalability issue in order to make causality analysis practical and applicable to the enterprise-level environment.

In this work, we present SEAL, a novel data compression approach for causality analysis. Based on information-theoretic observations on system event data, our approach achieves lossless compression and supports near real-time retrieval of historic events. In the compression step, the causality graph induced by the system logs is investigated, and abundant edge reduction potentials are explored. In the query step, for maximal speed, decompression is opportunistically executed. Experiments on two real-world datasets show that SEAL offers 2.63x and 12.94x data size reduction, respectively. Besides, 89% of the queries are faster on the compressed dataset than the uncompressed one, and SEAL returns exactly the same query results as the uncompressed data.

1 Introduction

System logs constitute a critical foundation for enterprise security. The latest computer systems have become more and more complex and interconnected, and attacker techniques have advanced to take advantage and nullify the conventional security solutions which are based on static artifacts. As a result, the security defense has turned more to pervasive system event collection in building effective security measures. Research has extensively explored security solutions using system logs. *Causality analysis* in the log setting (or *attack provenance*), as defined in [83], is one such direction that reconstructs information flow by associating interdependent system events and operations. For any suspicious events, the analysis automatically traces back to the initial penetration (root-cause diagnosis), or measures the amount of the impact by enumerating the system resources affected by the attacker (attack ramification). Encouragingly, the security solutions based on pervasive system monitoring and causality analysis no longer remain as a research prototype. Many proposed ideas have actualized as commercial solutions [8, 14, 22].

However, due to their data-dependent nature, the effectiveness of the above security solutions is heavily constrained by the system's data storage and processing capability. On one hand, keeping large volumes of comprehensive historical system events is essential, as the security breach targeting an enterprise tends to stay at the network over a long span: an industry report by TrustWave [78] shows, on average, an intrusion *prolongs over 188 days* before the detection. On the other hand, the size of a typical enterprise network and the amount of system logs each host generates could put high pressure on the security solutions. For instance, our industrial partner reported that on average *50 GB amount of logs are produced from a group of 100 hosts daily*, and they can only sustain *at most three months of data* despite the inexpensive storage cost. There is a compelling need for a solution that can scale storage and processing capacity to meet the enterprise-level requirement.

Lossless compression versus lossy reduction. Compression techniques [79] come in handy for improving the storage efficiency of causality analysis. Existing approaches [37, 45, 77, 83] tend to carry out *lossy reduction*, which *removes* logs matching pre-defined patterns, leading to unavoidable information loss. Although they showed that the validity of causality analysis is preserved on samples of investigation tasks, *there is no guarantee that every task will derive the right outcome*. In Section 2.3, we show examples about when they would introduce false positives/negatives. In addition, the accuracy of other applications such as behavioral detection [30, 53] and machine-learning based anomaly detec-

tion [10, 19, 47, 62, 63, 84] would be tampered, when they use the same log data. Alternatively, *lossless compression* [79] allows *any* information to be restored and thus causality analysis is preserved. Though the standard tools like Gzip [18] are expected to achieve a high compression rate, they are not applicable to our problem, because high computation overhead of *decompression* will be incurred when running causality analysis.

In this work, we challenge the common belief that lossless compression is inefficient for causality analysis, by developing SEAL (**S**torage-**E**fficient **A**nalysis on enterprise **L**ogs) under information-theoretic principles. Compared to the previous approaches, logs under a wider range of patterns can be compressed in a lossless fashion without the need for carefully examining conditions such as traceability equivalence or dependence preservation, while the validity and efficiency of any investigation task of causality analysis are preserved.

Contributions. The main contributions of this paper are as follows.

• We develop a framework of *query-friendly compression* (QFC) specialized for causality analysis. In this framework, the dependency graph is induced from the logs, and lossless compression is applied to the structure (vertices and edges) and then to the edge properties, or attributes (e.g., timestamp). QFC ensures every query is answered accurately, while the query efficiency is guaranteed as the majority of operations required by queries are done *directly on the compressed data*.

• We design compression and querying algorithms according to the definition of QFC. For graph structures, we define merge patterns to be subgraphs whose edges are combined into one new edge. For edge properties, delta coding [59] and Golomb codes [28] are applied to exploit temporal locality, meaning that consecutively collected logs have similar timestamps. To return answers to a causality query, the proposed method obviates decompression unless the relationship between the timestamps of a compressed edge and the time range of the query cannot be determined.

• A compression ratio estimation algorithm is provided to facilitate the decision of using the compressed or uncompressed format for a given dataset. We show that the compression ratio can be determined by the average degree of the dependency graph. Our algorithm estimates the average degree by performing random walk on the dependency graph with added self-loops, and randomly restarting another walk during the process. If the estimated compression ratio of a given dataset is smaller than a specified threshold, compression can be skipped.

• The above algorithms are implemented in SEAL, which consists of the compression system that is applied to online system logs and the querying system that serves causality analytics. Due to the large amount of merge patterns in the dependency graphs, SEAL can compress online log data into a significantly smaller volume. In addition, the query-friendly design reduces the required decompression operations. We evaluate SEAL on system logs from 95 hosts provided by our industrial partner. The experiment results demonstrate an average of 9.81x event reduction, 2.63x storage size reduction. Besides, 89% of the queries are faster on the compressed dataset than the uncompressed one. We also evaluate SEAL on DARPA TC dataset [16] and achieved 12.94x size reduction. Causality analysis to investigate attacked entities is shown to return accurate results with our compression method.

2 Background

We first describe the concepts of system logs and causality analysis. Then, we review the existing works based on lossy reduction and compare SEAL with them.

2.1 System Logs

To transparently monitor the end-host activities in a confined network, end-point detection and response (EDR) has become a mainstream security solution [35]. A typical EDR system deploys data collection sensors to collect the major system activities such as *file*, *process* and *network* related events, as well as events with high security relevancy (e.g., login attempts, privilege escalation). Sensors then stream the collected system events to a centralized *data back-end*. Data collection at end-host hinges on different operating systems' (OS) kernel-level supports for system call level monitoring [11, 55, 69].

In this study, we obtained a dataset from the real-world corporate environment. Data sources are the system logs generated by kernel audit [69] of Linux hosts and Event Monitoring for Windows (ETW) [55] of Windows hosts respectively. The system events belong to three different categories: (i) process accesses (reads or write) files (P2F), (ii) process connects to or accepts network sockets (P2N), and (iii) process creates other processes, or exits it executions. These system events captured from each end-host are transferred to the back-end and represented in a graph data structure [42] where nodes represent system resources (i.e., process, file, and network socket) and edges represent interactions among nodes. Our system labels edges with attributes specific to system operations. For instance, amounts of data transferred for file and network operations, command-line arguments for process creations. The dataset comprises of various workloads that range from simple administrative tasks to heavy-weight development and data analysis tasks and also includes end-user desktops and laptops as well as infra-structural servers.

Among the three categories of system events (file, network, and process) in the dataset, file operations account for the majority, taking over 90% portions, therefore become the primary target for SEAL compression. In particular, the *file operation* like create, open, read, write or delete is logged in each file event, alongside its *owner process*, *host ID*, *file path*, and *timestamp*. All file events have been properly anonymized

(no user identifiable information exists in any field of the table) to address privacy concerns.

Despite its improved visibility, data collection for in-host system activity results in a prohibitive amount of processing and storage pressures, compared to other network-level monitoring appliances [63]. For instance, our data collection deployment on average reported approximately 50 GB amount of logs for a group of 100 hosts daily. Given that a typical enterprise easily exceeds hundreds of thousands of hosts for its network, it is imminent to address the scalability issues in order to make causality analysis practical and applicable to a realistic network.

2.2 Causality Analysis in the Log Setting

After the end-point logs are gathered and reported to the data processing back-end, different applications are run atop to produce insights to security operators, such as machine-learning based threat detection [10], database queries [23–25] and causality analysis (or data provenance) [83]. Although our approach mainly focuses on the causality analysis, which requires high fidelity on its input data, it also benefits other analyses as our approach reduces data storage and computational costs.

To its core, causality analysis automates the data analysis and forensic tasks by correlating data dependency among system events. Using the restored causality, security operators accelerate *root cause* analysis of security incident and attack ramification. The causality analysis is considered to be a *de facto* standard tool for investigating long-running, multi-stage attacks, such as Advanced Persistent Threat (APT) campaigns [58]. For any suspicious events reported by users or third-part detection tools, the operator can issue a query to investigate causally related activities. The causality analysis then consults to its data back-end to restore the dependencies within the specified scope. The accuracy of causality analysis relies on the completeness of data collection, and the analysis response time and usability depend on the data access time. In Section 3.1, we demonstrate a causality analysis where our compression approach addresses the scalability issues without deteriorating accuracy and usability.

2.3 Comparison with Lossy Reduction

To reduce the storage overhead in supporting causality analysis, prior works advocated *lossy reduction* [37, 45, 77, 83], which removes logs of certain patterns before they are stored by the back-end server. Here we show the reduction rules of the prior works and compare their scope to `SEAL`.

LogGC [45] removes temporary files from the collected data that are deemed not affecting causality analysis. NodeMerge [77] merges the read-only events (`Read` events in our data) during the process initialization. The approach proposed by Xu et al. [83] removes repeated edges between two objects

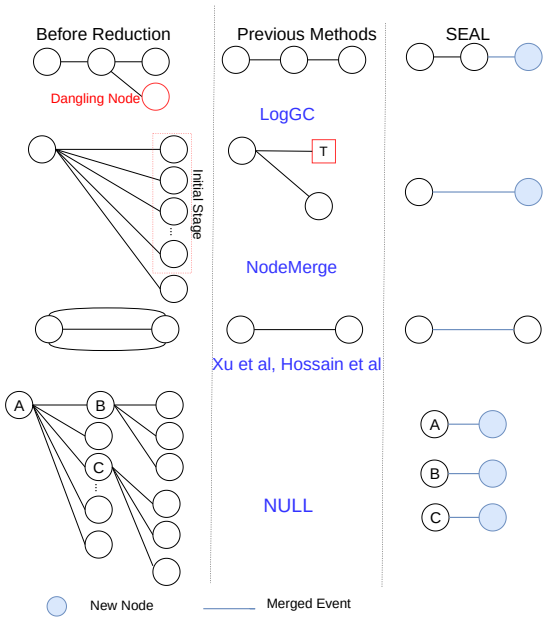

Figure 1: Comparison of our method `SEAL` to LogGC [45], NodeMerge [77], methods by Xu et al. [83] and Hossain et al. [37]. In NodeMerge (the second graph in the middle column), the node T represents a new node. In `SEAL` (the right column), the blue solid circles represent new nodes.

on the same host (e.g., multiple read events between a file and a process) when a condition termed trackability equivalence is satisfied. Hossain et al. [37] relaxes the condition of [83] such that more repeated events (e.g., repeated events *cross hosts*) can be pruned, which tends to be more conservative to maintain graph trackability.

`SEAL` is more general compared to any of the existing works. Our *lossless compression* schema is agnostic to file types and is therefore complementary to LogGC. `SEAL` also processes `Write` and `Execute` events, compared to NodeMerge, and therefore covers the whole life-cycle of a process. Compared to Xu et al. and Hossain et al., `SEAL` is more aggressive, e.g., merging not only the edges repeated between a pair of nodes. Figure 1 also illustrates the differences. In Section 5, we compare the overall reduction rate, with Hossain et al., which is the most recent work.

In terms of data fidelity, none of the prior works can guarantee false negative/positive would not occur during attack investigation. For LogGC, if the removed temporary files are related to network sockets, data exfiltration done by the attacker might be missed. For NodeMerge, the authors described a potential evasion method: the attacker can keep the malware waiting for a long time before the actual attack, so that the malware might be considered as a read-only file as determined by their threshold and break the causality dependencies (see [77] Section 10.4). PCAR of Xu et al. introduces false connectivity in two (out of ten) investigation tasks (see [83] Section 4.3). Similarly, false negatives could

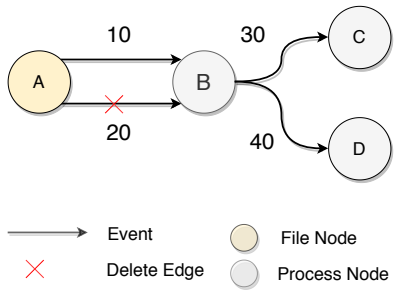

Figure 2: A dependency graph (see Section 3.1) under Full Dependency (FD) preservation reduction [37], where one edge is removed. The timestamp of each event is labeled on the edge. When querying for nodes dependent on Node A after time 15, the reduced dataset returns the empty set but the original dataset returns B, C, D. From the example we see that FD can return less number of nodes for causality analysis under time constraints.

be introduced to the system of Hossain et al. when the query has a time constraint, and we provide an example in Figure 2.

On the other hand, as SEAL ensures the completeness of logs, it can mitigate any of the above issues.

3 Log Compression

SEAL aims to compress the *dependency graph* constructed from system logs, as illustrated in Figure 3, while supporting the causality analysis without sacrificing query efficiency and analysis accuracy. If every analysis task results in decompressing a large portion of data, the goal of query efficiency will not be achieved. If compression causes significant information loss, the forensic analysis might lead to incorrect conclusion. Therefore, we design SEAL to compress the vertices and edges of a large amount of redundant information, and the compressed sets of edges are chosen such that we can restrain the frequency or overhead of decompression.

In this section, we first describe the dataset to be processed and the query to be run by an analyst. Then, we introduce the concept Query-friendly Compression (QFC) and show how it can be applied to system logs. Moreover, we introduce the compression algorithms that can be applied on the logs and compare them with the prior research. Finally, we propose an algorithm to estimate the compression ratio based on which one can determine when to compress.

3.1 Dataset and Event Query

Table 1 shows the primary dataset (FileEvent) we need to compress and the main fields. The start and end timestamp of each event are logged by starttime and endtime. An event links a source object and a destination object, distinguished by srcid (Source ID) and dstid (Destination ID). The object associated with each event can be file or process. All events occur within a host, denoted by agentid, and there is no cross-host event. There are three types of operations associated with an event, including Execute, Read and Write, recorded by accessright. To notice, the properties of objects, like the filenames and paths, are stored in other tables. But because the other tables' volume is small, we do not process them specifically.

Field	Exemplar Value
starttime	1562734588971
endtime	1562734588985
srcid	15
dstid	27
agentid	-582777938
accessright	Execute

Table 1: On example entry of FileEvent.

New Node	Represented Nodes
a	A, B
b	B, C
c	G, H

Table 2: Node map for the example in Figure 3.

Causality analysis on FileEvent. We assume that a dependency graph $G = (V, E)$ can be derived from FileEvent, in which the vertices (V) are the objects and the *directed* edges (E) are the events. Causality analysis uncovers the causality dependency of edges, and we define its computation paradigm below, in a way similar to the definition from Wu et al. [83].

Definition 1 (Causality dependency). Given two adjacent directed edges $e_1 = (u, v)$ and $e_2 = (v, w)$, there is a causality dependency between them, denoted by $e_1 \rightarrow e_2$, if and only if $f_e(e_1) < f_e(e_2)$, where f_e extracts starttime of an event. Dependency is also defined for non-adjacent edges by transitivity: if $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$, then $e_1 \rightarrow e_3$.

To conduct *causality analysis*, the analyst issues a query specifying the constraints to find the POI (Point-of-Interest) vertex v. The set of edges directly linked to v (termed E_v) and the ones with causality dependency to E_v will be returned. To notice, both forward-tracking (i.e., finding e_{fwd} such that $e \rightarrow e_{fwd}$) and back-tracking (i.e., finding e_{bck} such that $e_{bck} \rightarrow e$) can be supported, and in this work we focus on back-tracking [42], which is a more popular choice. Usually, newly discovered vertices/edges are returned to the analyst in an iterative way. The process will terminate when no more vertices/edges are discovered or the maximum depth specified by the analyst has been reached. In each iteration, the analyst can refine the query constraints to reduce the analysis scope.

Figure 3 (left) shows an example of a dependency graph generated from FileEvent. There are three file nodes (A, B and C) and five process nodes (D, E, F, G, H). The edge is formatted as [starttime, endtime, srcid, dstid,

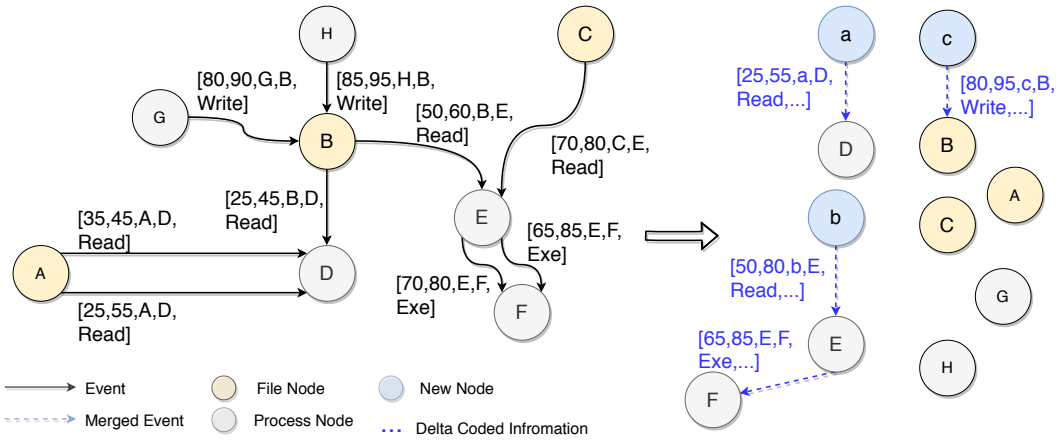

Figure 3: An example of dependency graph (left) and its compressed version after applying SEAL (right). Edges are merged if and only if they share the same destination node. To facilitate causality queries, the smallest starttime and the largest endtime are defined as the first two fields in the new edge. The ellipsis mark represents the time for all the compressed edges. For example, the edge between a and D is [25,55,a,D,Read,(25,55),(35,45);(25,45)]. We use comma to separate repeated edges and use semicolon to separate different edges. Therefore, combining with the node map in Table 2, we can see the compression is lossless. The edge properties will be further compressed as described in Section 3.4.

accessright]. Given a back-tracking query about POI vertex F and starttime ranged in [45, 100], three causal events will be reported: [70, 80, E, F, Execute], [65, 85, E, F, Execute], and [50, 60, B, E, Read]. The other edges do not satisfy the definition of causality analysis.

3.2 Query-friendly Compression

While compression is a well-developed area, with numerous methods available, many of them will introduce prominent overhead to causality analysis, as they require decompression every time a vertex/edge is examined. In this work, we adopt a concept from the data-mining community, termed *Query-friendly Compression (QFC)* [20, 52, 54], and develop compression techniques around it. In essence, the techniques under QFC should compress graphs "in a way that they still can be queried efficiently without decompression" [54]. For example, 4 types of queries can be supported with QFC algorithms of [20], including neighborhood queries, reachability queries, path queries, and graph pattern queries. Causality analysis can be considered as an iterative version of neighborhood queries.

Yet, the QFC schema of prior works cannot be directly applied to our setting. Firstly, some mechanisms require significant change on the data structures [54]. For our deployment, regular SQL queries have to be supported as well so the data format after compression has to adhere to the database schema. Secondly, the edges in all prior works have no associated properties [20, 52, 54], therefore only merging vertices is sufficient to fulfill their goal. While we can follow the same approach and keep the edge properties concatenated without compression, such a design is not optimal. Moreover, the queries on dependency graphs depend on not only the connectivity of the nodes but also the edge properties like starttime, leading to the challenge of retrieving the answers.

Therefore, we modify QFC according to causality analysis, which enforces *"decompression-free"* compression on graph structure and *"query-able"* compression on edge properties. Below we define the adjusted QFC based on the definition from [20].

Definition 2 (*Query-friendly Compression*). Assume a dependency graph $G = (V, E)$ is to be compressed. Let the class of causality analysis queries be $\mathcal{Q}$, and let $Q(G)$ be the answer to the query $Q \in \mathcal{Q}$. A QFC mechanism is a triple $< R, F, P >$, where R is a compression method, $F : \mathcal{Q} \to \mathcal{Q}$ re-writes $\mathcal{Q}$ to accommodate the compressed data, and P is a post-processing function. Compression can be expressed as $R(G) = R_p(R_s(G))$, where R_s compresses the structures (vertices and edges), and R_p compresses the edge properties or fields. Denote by $G_r = R(G) = (V_r, E_r)$ the graph after compression, such that $|E_r| \leq |E|$. QFC requires that for any query $Q \in \mathcal{Q}$,

- $Q(G) = P(Q'(G_r))$, where $Q' = F(Q)$ is the query on the compressed graph, and $P(Q'(G_r))$ is the result after post-processing the query answer on G_r.
- With only R_s applied, any algorithm for evaluating Q can be directly used to compute $Q'(G_r)$ *without decompression*.
- When both R_s and R_p are applied, decompression is needed only when the relationship between the timestamps of a compressed edge $e \in E_r$ and the time range of the query cannot be determined.

Next, we describe our choices of R_s and R_p in Section 3.3 and Section 3.4. The query transformation F and the post-processing P are investigated in Section 3.5. Figure 3 (right) overviews the graph compressed with SEAL.

3.3 Compression on Graph Structure

We design the function R_s such that multiple edges (from one pair of nodes or multiple pairs) can be reduced into a single edge. In particular, our algorithm finds sets of edges satisfying a certain *merge pattern* and combines all edges in the set. By examining the fields of `FileEvent`, one expects a higher compression ratio if edges with common fields are merged. Moreover, edges within proximity can be merged without sacrificing causality tracking performance. As illustrated in Figure 3, we choose the merge pattern to be *the set of all incoming edges of any node $v \in V$, which will share properties such as* `dstid` *or* `agentid`. Correspondingly, a new node is added in the new graph G_r, representing the combination of all the parent nodes of v, if the number of parent nodes is more than 1.

We give an example in Figure 3. The new node `a` is generated to correspond to two individual nodes {`A`,`B`}, and the new edge [`25, 55, a, D, Read, (25, 55), (35, 45); (25, 45)`] is generated to correspond to three individual edges {[`25, 55, A, D, Read`], [`35, 45, A, D, Read`], [`25, 45, B, D, Read`]}. Similarly, we merge the two incoming edges of node `B`, merge the two incoming edges of node `E`, and create new nodes `c, b`, respectively. We also merge the two repeated edges between nodes `E, F`, but no new node needs to be created for them. Individual edges are removed in the compressed graph G_r if they are merged into a new edge. However, as can be seen in Figure 3, individual nodes should not be removed. For example, even if the individual node `B` is included in the new node `a`, it cannot be removed because of its own incoming edges. The new nodes are recorded in a *node map*, shown in Table 2.

Our algorithm for R_s is shown in Algorithm 1. It takes all the events as input, and creates two hash maps: (i) *NodeMap*, child node with all its parent nodes, and (ii) *EdgeMap*, a pair of nodes with all its corresponding edges. Then for each child node $v \in V$, all its parent nodes and the corresponding incoming edges are identified and merged. Meanwhile, the node map as in Table 2 is also updated. The time complexity of this algorithm is linear in the size of the graph, namely, $O(|V|+|E|)$. When responding to queries, decompression is selectively applied to *restore* the provenance, with the help of *NodeMap* and *EdgeMap*.

3.4 Compression on Edge Properties

For all the properties or fields for a merged edge, they should be combined and compressed due to the redundant information, which is the focus of the compression function R_p. We propose delta coding for merged timestamp sequence, and Golomb code for the initial value in the sequence.

Delta coding. Delta coding represents a sequence of values with the differences (or delta). It has been used in updating webpages, copying files online backup, code version control and etc. [59]. We apply delta coding on timestamp fields (`starttime`, `endtime`) , as they usually share a long prefix. For instance, as shown in Figure 4, the `starttime` field is a long integer, and merged individual edges have values like 1562734588980, 1562734588971, 1562734588984, 1562734588990. Those values usually share the same prefix as the events to be compressed are often collected in a small time window, hence delta coding can result in a compact representation.

As shown in Figure 4, assume a node x has d incoming edges and p parent nodes, $1 \le p \le d$. Let the `starttime` of the j-th edge be t^j_{start}, $1 \le j \le d$. We first construct a sequence

$$\mathbf{t}_{start} = [t^0_{start};\quad t^1_{start};\quad t^2_{start}, t^3_{start};\quad \ldots;\quad t^d_{start}:]$$

where $t^0_{start} = \min_{1\le j\le d}(t^j_{start})$. Here comma is used to separate different edges from the same parent node, and semicolon separates different parent nodes. The colon at the end is used to separate the timestamp fields. For `endtime`, we choose the initial entry t^0 to be the *maximum* among the edges. Then we concatenate both fields into one sequence.

Algorithm 1 Graph structure compression.

Input: a set of edges E.
Output: a set of new edges E', a node map *NodeMap*.

```
NodeMap ← ∅          ▷ hash map (key = a node, value = parent nodes)
EdgeMap ← ∅ ▷ hash map (key = a pair of nodes, value = edges)
for e = (u,v) ∈ E do
    NodeMap.put(v,u)
    EdgeMaps.put((u,v),e)
end for
E' ← ∅
for v ∈ NodeMap.keys do
    e' = ∅                                   ▷ a new edge
    U ← NodeMap.get(v)
    for u ∈ U do
        e' ← e' ∪ {EdgeMap.get((u,v))}
    end for
    E' ← E ∪ {e'}
end for
```

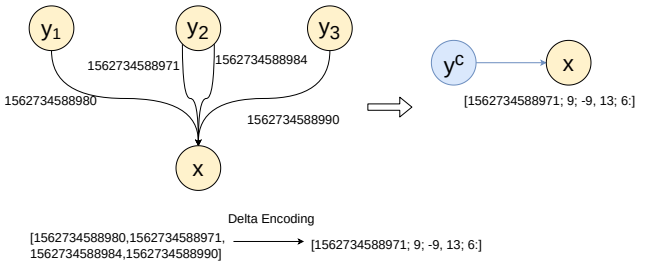

Figure 4: Delta coding for `starttime`. The first number in the combined time vector is the minimum time among the edges.

Then, we compute the delta for every consecutive pair of timestamps: for $1 \leq j \leq d$, $\Delta^j_{start} = t^j_{start} - t^{j-1}_{start}$. The resulting coded timestamp of the merged edge is:

$$[t^0_{start};\quad \Delta^1_{start};\quad \Delta^2_{start}, \Delta^3_{start};\quad \Delta^4_{start};\quad ...,\quad \Delta^d_{start}:]$$

and delta coding is also applied to the other timestamp fields. The time complexity of delta coding is $O(d)$ where d is the number of edges.

To conform to the uncompressed `FileEvent` format, the t^0_{start} and t^0_{end} are stored in the `starttime` and `endtime` field of the new edge e^c respectively, and the generated delta-coded `starttime` and `endtime` are stored in a new `delta` field.

Golomb coding. Delta coding can compress all the elements of the combined time sequence except t^0 which is still a long integer. Moreover, if an individual edge is not merged, its timestamps are also long integers. We choose to employ Golomb coding [28] to compress long integers to relatively small integers. Alternatively, a more aggressive approach is to use delta coding to compress t^0 of different merged events, but the database index will be updated [13] and the query cost will be high. One favorable property of Golomb coding is that the relative order of the numbers is not changed, which fits well with the requirements of `QFC`. That is, if $t > t'$, then we have the Golomb coded variable $\text{Gol}(t) > \text{Gol}(t')$.

Golomb code uses a parameter M to divide an input datum N into two parts (quotient q and reminder r) by

$$q = \lfloor \frac{N-1}{M} \rfloor,\ r = N - qM - 1. \tag{1}$$

Under the standard Golomb coding schema, the quotient q is then coded under unary coding, and the reminder r is coded under truncated binary encoding to guarantee that the value after coding (called codeword) is a prefix code. In our case, however, the truncated binary encoding is not necessary because the codewords are separated by different entries automatically. As such we use a simpler mechanism, binary coding, for r. The coded data is then calculated by concatenating p and r. For instance, given a long integer 1562734588980 (64 bits) and a $M = 1562700000000$, the binary form of p and r after coding will be 10 (2 bits) and 10000011111100100100110100 (26 bits). In this example, 32 bits are sufficient to store the Golomb codeword.

3.5 Query and Decompression

As defined by `QFC`, decompression is only necessary when the relation between the time range specified in the query and in the edge cannot be determined. If there are no intersections of these two ranges, decompression can be skipped. In our back-tracking queries, the above property holds for two reasons. First, due to the *order preservation* property of Golomb coding, it is unnecessary to decode all Golomb codes in the database to answer a query with a timestamp constraint. The specified timestamp can be simply encoded by Golomb code, and used as the new constraint issued to the database. Second, the minimum `starttime` t^0_{start} is recorded in a merged edge. Hence, if we back-track for events whose `starttime` is smaller than some given t_{query}, then all individual edges of an combined edge with $t^0_{start} > t_{query}$ will be rejected. Therefore, the database does not need to decompress and can safely reject this combined edge.

Here we use the example shown in Figure 3 to demonstrate how the query and decompression work. Assume a query tries to initiate back-tracking on `E` to find the prior causal events whose `starttime` is less than $t_{query} = 65$. First, t_{query} will be Golomb coded into $Gol(65)$. And the database needs to find events such that $Gol(t^0_{start}) < Gol(65)$ and the destination node is `E`. For the merged event [`50, 80, b, E, Read`], its $t^0_{start} = 50$ value is stored as $Gol(50)$. By order preservation of Golomb code, $Gol(50) < Gol(65)$. Thus this merged event will be identified. Second, we decompress this merged event for further inspection. We extract `starttime` $Gol(t^0_{start}) = Gol(50)$ and Golomb decoding is applied to obtain $t^0_{start} = 50$. Then we recover the timestamp sequence $\mathbf{t}_{start}$ by calculating $t^j_{start} = t^{j-1}_{start} + \Delta^j_{start}, j \geq 1$. In this example, $t^1_{start} = 50, t^2_{start} = 70$. Comparing the individual timestamps now is feasible. It will be found that only the first individual edge is a valid answer. The final step is to find the individual nodes corresponding to the valid edges from the node map in Table 2. After that, the result [`50, 60, B, E, Read`] is returned.

It can be seen that if $t_{query} = 30$, all incoming edges of `E` can be rejected without Golomb or delta-code decompression (t_{query} still needs to be Golomb encoded before issuing the query).

3.6 Compression Ratio Estimation

Applying compression to the log data may be desirable only if the compression ratio is higher than a threshold. As a result, it is important to obtain the compression ratio or its estimate before compression. While a full scan of the causality graph gives a precise compression ratio, the overhead is significant. As a result, we develop an algorithm to estimate the compression ratio. To that end, we show that this estimation is reduced to obtaining d_{avg}, the average degree of the undirected version of the causality graph (Appendix A). A degree estimator is developed with a sample size only depending on the required accuracy rather than on the number of nodes or the number of edges. As described in Section 4, we implement `SEAL` for online compression, this algorithm is applied to chunks of data sequentially.

Compression ratio estimation. Let $G_{undirected}$ denote the undirected graph which is identical to the dependency graph except that edge directions are removed. Let d_{avg} be its average node degree. From Appendix A, we find that the compression ratio is an explicit function of d_{avg}. The compression ratio

estimation reduces to estimating the average degree. To minimize the data access and query time during estimation, we present an average degree estimation algorithm that samples nodes in an undirected graph H based on random walk (see Algorithm 2). The algorithm can be applied to $H = G_{undirected}$ and outputs d_{avg}. In the following, we use the notation d_H for the average degree of H, and $\hat{d}$ the estimated average degree. For any vertex v of H, denote by d_v its degree.

One way to estimate the average degree is to uniformly sample nodes in H and get their degrees, and obtain the average of the sampled degrees [21]. The estimator from the sample set S is:

$$\hat{d} = \frac{\sum_{v \in S} d_v}{|S|} = \frac{\sum_{v \in S} d_v}{\sum_{v \in S} 1}. \quad (2)$$

This method can be improved when we also obtain a random neighbor of each sampled nodes [27]. The required number of samples (sample complexity) is $O(\sqrt{n})$ to obtain a constant-factor estimation, where n is the number of nodes. Another way is to sample nodes according to the node degree, and use collisions in the samples to obtain the estimate [41], where the required sample complexity is $\Omega(\sqrt{n})$. Our algorithm is inspired by the 'Smooth' algorithm of [17], where a node v is sampled proportional to its degree plus a constant, $d_v + c$, where the constant $c = \alpha d_H$ is a coarse estimate of the average degree with a multiplicative gap α. The coarse estimation c can be obtained from history or a very small subgraph in our problem. The resultant sample complexity is no more than $\max(\alpha, \frac{1}{\alpha})\frac{6}{\varepsilon^2}\log\frac{4}{\delta}$, and the average degree estimate $\hat{d}$ satisfies

$$Pr\Big((1-4\varepsilon)d_H \leq \hat{d} \leq (1+4\varepsilon)d_H Big) \geq 1-\delta, \quad (3)$$

for all $0 < \varepsilon \leq 0.5, 0 < \delta < 1, \alpha > 0$.

In large graphs, it is hard to sample nodes in the entire graph according to some distribution as we do not know the number of nodes and the node degrees. To overcome such difficulty, the Smooth algorithm can be modified such that the sampled nodes are obtained by random walk [17]. However, it makes some assumptions that do not readily fit the dependency graph problem: (i) The graph needs to be irreducible and aperiodic. However, the dependency graph naturally contains disconnected components. (ii) The sample complexity needs to be high enough to pass the mixing time and approach the steady-state distribution, which varies depending on the structure of the graph.

To overcome these issues, two techniques are used in Algorithm 2. First, random walk with escaping [7] jumps to a random new node with probability p_{jump} and stays on the random walk path with probability $1 - p_{jump}$ (see Line 4). Therefore, we can reach different components of the graph. Second, thinning [41] takes one sample every θ samples as in Line 7. We obtain θ groups of thinned samples. If the samples are indexed by $0, 1, 2, \ldots$, then in our algorithm the j-th group, denoted by S_j, contains samples indexed by $j, j+\theta, j+2\theta, \ldots$, for $0 \leq j \leq \theta - 1$. Each group produces its own estimate (Line 13), and the final estimate is the average of these groups (Line 14). Since the sample distribution is not uniform, we cannot directly use the estimator of Equation (2). The sampled degrees need to be re-weighted using the Hansen-Hurwitz technique [32] to correct the bias towards the high degree nodes, corresponding to the term $d_v + c$ in the numerator and the denominator of Line 13. Note that due to the difficulty to sample a node from the entire graph, the sample distribution is not specified in Lines 2 and 9.

Algorithm 2 Average degree estimation.

Input: undirected graph H, sample size r, coarse average degree estimator c, thinning parameter θ, jumping probability p_{jump}
Output: average degree estimator $\hat{d}$

1: $S_j \leftarrow \emptyset, j = 0, 1, \ldots, \theta - 1$
2: Randomly sample a node v_{pre} of H
3: **for** $i = 0$ to $r - 1$ **do**
4: $\quad rnd \sim$ Bernoulli(p_{jump})
5: $\quad$ **if** $rnd = 0$ **then**
6: $\quad\quad$ Uniformly sample a neighbor v of v_{pre} assuming v_{pre} also has c added self loops
7: $\quad\quad S_{i \mod \theta} \leftarrow S_{i \mod \theta} \cup \{v\}$
8: $\quad$ **else**
9: $\quad\quad$ Randomly sample a node v of H
10: $\quad$ **end if**
11: $\quad v_{pre} \leftarrow v$
12: **end for**
13: $\hat{d}_j = \frac{\sum_{v \in S_j} d_v/(d_v+c)}{\sum_{v \in S_j} 1/(d_v+c)}, j = 0, 1, \ldots, \theta - 1$
14: $\hat{d} = \frac{1}{\theta}\sum_{j=0}^{\theta-1} \hat{d}_j$

4 Architecture

Design rationale. Figure 5 shows the architecture of SEAL and how it is integrated into the log ingestion and analysis pipeline. SEAL resembles the design [77] at the very high level. In [77], the compression system mainly includes three elements: computing components, caches, and the database. In this work, we redesign those elements according to our algorithm for both the compression system and the query system. The *compression system* receives online data streams of system events, encodes the data, and saves them into the database. The *query system* takes a query, applies the query transformation and recovers the result with post-processing, and returns the result. The information flow follows closely the definition of QFC in Section 3.2 and includes the structure and property compression R_s, R_p, the query transformation F, and the post-processing P, which are explained in details in Sections 3.3 – 3.5.

Due to the current monitoring system structure of our indus-

trial collaborator, SEAL is solely deployed at the server-side by the data aggregator. Note that, alternatively, one can choose to compress the data at the host end before sending them to the data aggregator. Since there are no cross-host events in FileEvent, the compression ratio will be identical for both choices.

Online compression. While offline compression can achieve an optimized compression ratio with full visibility to the data, it will add a long waiting time before a query can be processed. Given that causality analysis could be requested any time of the day, offline compression is not a viable option. As such, we choose to apply online compression.

The online compression system is built by the following main components: (i) The optional compression ratio estimator. If the estimated ratio as described in Section 3.6 is more than the given threshold, data is passed through the following components. Otherwise, data is directly stored in the database. (ii) Caching. It organizes and puts the most recent data steam into a cache. When the cache is filled, the data will be compressed. The cache size is configurable, called *chunk size*. (iii) Graph structure compression. It merges and encodes all the edges that satisfy the edge merge pattern as in Section 3.3. It also generates the node mapping between the individual nodes and the new nodes, shown in Table 2. (iv) Edge property compression. It encodes each event timestamp entry using delta coding and Golomb codes as in Section 3.4.

Next, we remark on some design choices. The configurable chunk size provides a tradeoff between the memory cost and the compression ratio. The larger the chunk size, the more edges can be combined. We found in our experiments as in Section 5 that 134 MB per host is a large enough chunk size offering sufficiently high compression capability.

Query. The query system comprises three main components. (i) Query transformation. Given a query Q, SEAL transforms it into another query Q' that the compressed database can process. In particular, it needs to transform the queried timestamp and the srcid constraints, if there are any. The timestamp constraint is encoded into a Golomb codeword, which is used as the new constraint as in Section 3.5. If a srcid is given, then this individual node is mapped to all the corresponding new nodes using the node map. (ii) Querying. The transformed query Q' is issued to the database and the answer is obtained. (iii) Post-processing. The combined edges are decompressed from delta codes, the timestamp constraint is checked, the merged node is mapped to individual nodes, and the valid individual edges are returned as described in Section 3.5.

Note that, in the query transformation component, if dstid is a query constraint, then no node mapping is required since only source nodes are merged during compression. In our work, we focus on back-tracking, where srcid is not a query constraint, hence the query transformation is simplified. Moreover, the node mapping progress is fast due to the small number of objects compared to the events.

For each given destination ID, at most one combined edge will be returned as an answer in each chunk (containing 10^5 to 10^6 events depending on the chunk size). This observation combined with the fact that the dependency graph is much smaller after compression effectively controls the query overhead in our experiments.

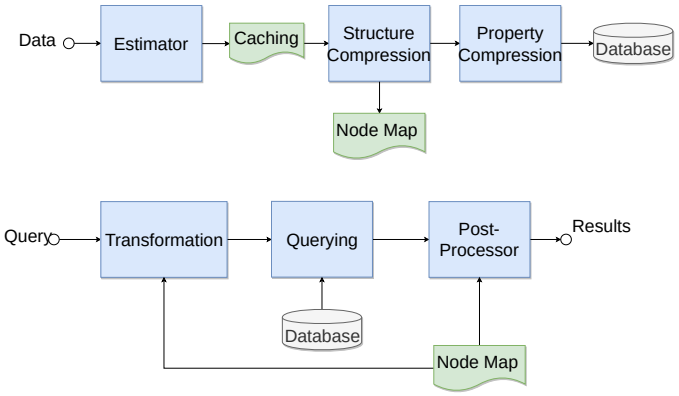

Figure 5: The SEAL architecture of online compression and querying.

To quickly access the node map as in Table 2, it is cached using a hash map. Given that the number of nodes is much smaller than the number of edges, the memory size of this hash map is a small fraction of the database size.

5 Evaluation

5.1 Experiment Setup

Our evaluation about compression is primarily on a dataset of system logs collected from 95 hosts by our industrial partner, which we call DS_{ind}. This dataset contains 53,172,439 events and takes 20GB in uncompressed form. For querying evaluations, we select a subset of DS_{ind} covering 8 hosts, with 46,308 events and a total size of 8 GB. As DS_{ind} does not have ground-truth labels of attacks, we use another data source under the DARPA Transparent Computing program [16]. The logs are collected on machines with OS instrumented, and a red team carried out simulated APT attacks. Multiple datasets are contained, and each one corresponds to a simulated attack. We use CARDETs dataset, which simulates an attack on Ngnix server, with a total of 1,183M events (27% write, 25.8% read and 47.2% execute), and we term this dataset DS_{dtc}. Since our system focuses on event merging, we only compress the edges and a subset of the attributes, with 233GB data size.

We implemented SEAL using JAVA version 11.0.3. We use JDBC (the Java Database Connectivity) to connect to PostgreSQL Database version Ubuntu 11.3-1.pgdg18.04+1. For DS_{ind}, we run our system on Ubuntu 14.04.2, with 64 GB memory and Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHZ. To run the queries, one machine with AMD Ryzen 7 2700X

Eight-Core Processor and 16GB memory is used. For DS_{dtc}, we run the system on Ubuntu 16.04, with 32 GB memory and Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz.

Section 2.3 compares the designs between SEAL and other systems, and demonstrates when other systems introduce errors to attack investigation. In this section, we quantify the difference, and select the method of Full Dependency (FD) preservation [37] as the comparison target, which strikes a good balance between reduction rate and preservation of analysis results. Under FD, A node u is *reachable* to v if either there is an edge $e = (u, v)$ or there is a causality dependency $e_u \rightarrow e_v$, where e_u is an outgoing edge of u, and e_v is an incoming edge of v. We implement a relaxed FD constraint, where repeated edges (between any pair nodes) are merged such that the reachability for any pair of nodes in the graph is maintained. The corresponding compression ratio is better than FD since it is a relaxation. We compare the relaxed FD with our method SEAL.

Our evaluation focuses on the following aspects. In Section 5.2, we study the data compression ratio and the number of reduced events for different hosts and different accessright operations (read, write, and execute). We demonstrate the impact of the assigned chunk size (for caching events) on the reduction factor. We compare our method to relaxed FD on compression rate. In Section 5.3, we compare the processing time of running back-tracking queries on the compressed and uncompressed databases. For the compressed case, the time for the database to return the potential merged events and the time for SEAL to post-process them are investigated. We show the accuracy advantages of lossless compression under queries with time constraints. Finally in Appendix B, we evaluate the accuracy of the average degree estimator and compare it with direct uniform sampling.

5.2 Compression Evaluation

Compression ratio. We measure the compression ratio as the original data system over the compressed data system using the above two chunk sizes. For DS_{ind}, when the chunk size (number of cached events) is 10^6, the compressed data is reduced to 7.6 GB from 20 GB, resulting in a compression ratio of **2.63x**. For DS_{tpc}, the chunk size equals one file size and contains around 5×10^6 events. The compressed size is 18 GB reduced from 233GB, resulting in a compression ratio of **12.94x**.

Reduction factor for different operations and hosts. To further understand the compression results, we investigate the reduction factor, defined as the number of original events divided by the number of compressed events. We focus on DS_{ind}, and some examples of the hosts and the average reduction factors from 95 hosts are illustrated in Table 3. In the table, we list results for the chunk size of 10^6 as well as 10^5.

It can be observed that the types of events (read, write, and execute) differ by the hosts. We observed that in DS_{ind}, read is the most popular operation among most of the hosts, where 72 hosts have more than 50% read events. Write is much less prevalent in general, where 67 hosts have between 10% to 30% write events. Finally, execution varies by the host, and 69 hosts have between 10% to 60% executions.

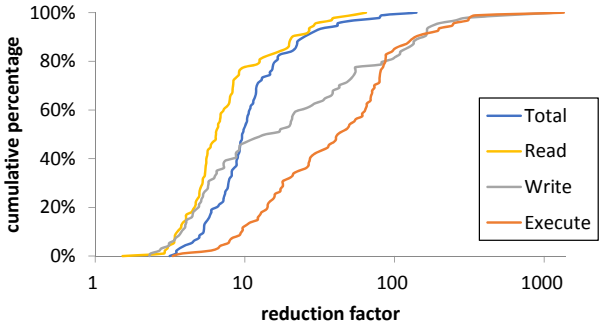

Figure 6: The cumulative distribution of the reduction factors for the 95 hosts in DS_{ind}. The reduction factor is calculated for all three types of operations, read, write, and execute, and the overall events in each host.

On average, the reduction factor of execute events is higher than reads, and writes have the lowest reduction factor, as can be seen from the last row of Table 3. However, for each host, this ordering changes depending on the structure of the dependency graph, e.g., if there exist many repeated events between two nodes. Host 5 is an example that has reduction factors similar to the average case. Hosts 23, 52, 3 see higher reduction factors of read, write, and execute events, respectively. Host 94 is an example of high reduction factors for all events. In Figure 6 we plot the cumulative distribution of the reduction factors among the 95 hosts.

The number of events of a host affects the reduction factor to some extent. In particular, if the number of events is less than the chunk size, as occurred for a few hosts when the chunk size is 10^6, the cache is not fully utilized, and fewer merge patterns may be found. However, some hosts with a small number of events still outperform the overall case as the last row in Table 3, due to their high average degree.

Previous works like NodeMerge focus on one type of operation, such as read [77], and show a high data reduction ratio on their dataset. Our result suggests such an approach is not always effective, when compressing data from different types of machines (e.g., Host 94). As such, SEAL is more versatile to different enterprise settings.

Chunk size. When the chunk size is increased from 10^5 to 10^6, the overall reduction factor is increased by 1.7 as in the last row of Table 3. Correspondingly, the consumed memory size is increased from 134 MB to 866 MB. The cumulative distribution of the reduction improvement, which is the reduction factor of chunk size 10^6 divided by that of chunk size 10^5, is plotted in Figure 7. The improvement is due to the fact that when more events are considered in one chunk,

Host ID	Event Count / Reduction	Read % / Reduction	Write /Reduction	Execute / Reduction
5	278913 / 9.25x / 5.85x	61% / 6.6x / 4.1x	11% / 9.2x / 8.4x	28% / 65.3x / 33.0x
23	880162 / 25.45x / 19.14x	91% / 37.7x / 26.9x	8% / 5.3x / 4.5x	1% / 35.8x / 13.2x
52	523671 / 41.45x / 17.36x	70% / 39.1x / 14.8x	22% / 54.9x / 47.5x	8% / 36.6x / 14.7x
3	312392 / 15.37x / 13.31x	36% / 12.6x / 10.8x	29% / 8.8x / 8.4x	34%/ 125.8x / 52.3x
94	517978 / 78.82x / 26.9x	20% / 19.8x / 6.1x	8% / 200.6x / 96.3x	72% / 346.0x / 209.1x
All	53172439 / 9.81x / 5.71x	65% / 10.3x / 5.5x	19% / 5.3x / 3.7x	15% / 76.3x / 38.7x

Table 3: Example hosts and the reduction factors. The reduction factors are measured for two chunk sizes: 10^6 and 10^5. The last row shows the overall result for the 95 hosts.

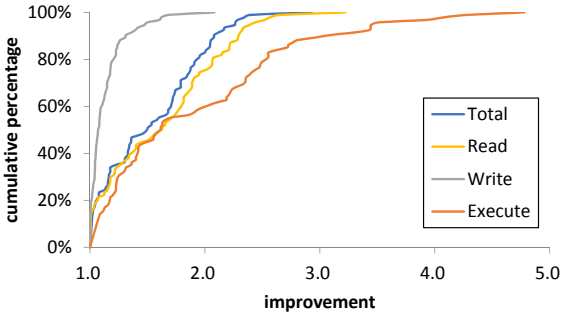

Figure 7: The cumulative distribution of the improvement over the 95 hosts when the chunk size is increased from 10^5 to 10^6. The improvement for Read, write, execute, and overall events in each host is calculated.

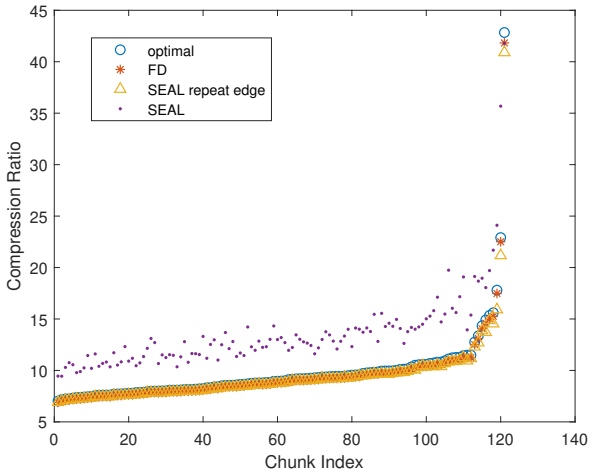

Figure 8: Comparison between our methods and FD.

more edges exist in the dependency graph, but the number of nodes does not increase as fast. A larger average degree and hence a larger reduction factor is achieved. It can be seen that the execute events change the most with a larger chunk size, while the write events change the least with the chunk size. This also is consistent with the fact that executions have more repeated edges between processes while write events operate on different files over time.

Comparison to FD. We use DS_{dtc} to compare `SEAL` and FD, as the DARPA data is also used by Hossain et al. [37]. Figure 8 shows the compression ratio of four methods: 1) "optimal" – keeping only one random edge between any pair of nodes, which violates causality dependency but gives an upper bound on the highest possible compression ratio when repeated edges are reduced, 2) "FD" – removing repeated edges under relaxed full dependency preservation, 3) "SEAL repeat edge" – our method that only compresses all repeated edges, and 4) "SEAL" – our method that compresses all incoming edges of any node.

Figure 8 shows that if we only compress the repeated edges by SEAL, we can get almost the same compression ratio as FD. Both methods are close to the minimum possible compressed size under repeated edge compression. Besides, if we compress all the possible edges using `SEAL`, we get a compression ratio of 12.94x compared to 8.96x for FD preservation.

5.3 Query Evaluation

We measured the querying and decoding time cost of `SEAL` as well as the querying time of the uncompressed data. We use a dataset with 830,235 events under DS_{ind} and run backtracking through breadth-first search (BFS) to perform the causality analysis for every node. We use BFS here as it can be seen as a generalization of causality analysis: if no additional constraints are assumed, causality analysis is BFS under causality dependency. In particular, starting from any POI node x, we query for all incoming edges $e_1, e_2, \ldots, e_d$ and the corresponding parent nodes $y_1, y_2, \ldots, y_d$, where d is the incoming degree of x. Then for each node y_i, $1 \leq i \leq d$, we query for its incoming events whose `starttime` is earlier than that of e_i. The process continues until no more incoming edge is found.

Figure 9 shows the performance of this evaluation. The querying and the decoding time on the compressed data normalized by the querying time on the uncompressed data are plotted. We obtain 133 start nodes each of which returns more than 2,000 querying results. We observe that **89%** starts nodes (118 out of 133) use less time than the uncompressed data, and 30 start nodes use less than half the time of the uncompressed data. Moreover, on average decompression only takes 18.66% of the overall time, because only potentially valid answers are decompressed. It is also observed that the

NID	Number of Reachable Nodes/Edges			
	Uncmp	SEAL	Cnstrnd Uncmp	Cnstrnd SEAL
1	1093/4302	1093/4302	293/779	293/779
2	9496/37944	9496/37944	1457/5999	1457/5999
3	178/616	178/616	116/358	116/358
4	45/3739	45/3739	11/2113	11/2113

Table 4: The results of back-tracking starting from 4 nodes. "NID", "Uncmp" and "Cnstrnd" are short for "Node ID", "Uncompressed" and "Constrained".

querying time of SEAL is only 63.87% of the querying time for uncompressed data. For DS_{dtc}, SEAL runs on about 5.27M nodes, 15.47% nodes use less time than the uncompressed data, and on average takes 1.36x time of the uncompressed data.

Note that queries usually have a restrictive latency requirement while compression of collected logs can be performed at the background of a minoring server. Our method tradeoff the computation during compression for better storage efficiency and query speed.

Evaluation of attack provenance. Here we use the simulated attacks of DS_{dtc} to evaluate whether SEAL preserves the accuracy for data provenance. We use four processes on two hosts (two for each) which are labeled as attack targets (ta1-cadets-2 and ta1-cadets-1) as the starting nodes. Then we run the BFS queries, and count 1) the number of nodes reachable from a starting node (reachable is defined in Section 5.1) and 2) the number of edges from a starting node to all its reachable nodes. Table 4 (Columns 2 and 3) shows the number of reachable nodes and edges in the BFS graph. It turns out SEAL returns the exact same number of reachable nodes and edges as the uncompressed data, indicating it preserves provenance accuracy. Next, we demonstrate the versatility of our lossless method for queries with time constraints, for example, when the analyst knows that the attack occurred in an approximate time period $[t_1, t_2]$. Since our lossless compression can restore all the time information, we can add arbitrary constraints to our analysis without any concerns, which is verified by the last two columns (Column 4 and 5) of Table 4. Lossy reduction methods, such as FD, even though preserve certain dependency, still lose time information once edges are removed, and thus might introduce false connectivity under time constraints (see Figure 2 for an example).

6 Discussion

Limitations and future works. DS_{ind} is collected for a small number of days and a subset of all hosts from our industrial partner. Therefore, a larger dataset may provide a more comprehensive understanding of the performance for SEAL. The

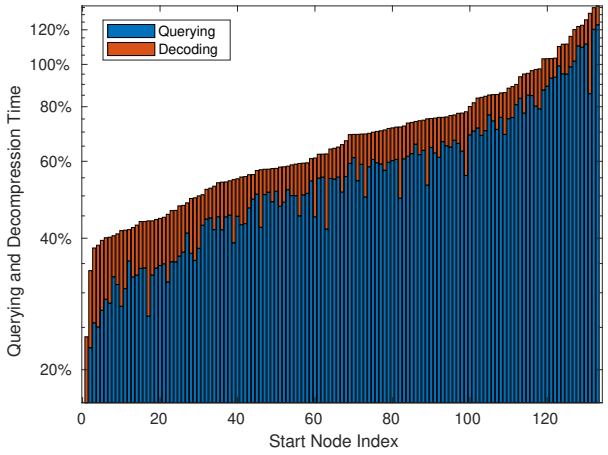

Figure 9: Querying and decompression time of back tracking with 133 start nodes that return the largest result sizes, normalized by the querying time of the uncompressed data. The nodes index are sorted by the query time.

compression ratio can be further improved through two possible methods. First, the proposed algorithms reduce the number of events, but the properties of all merged events are losslessly compressed together. Even though such compression produces a hundred percent accuracy for log analytics and the merge patterns can be easily found, dependency-preserving timestamp lossy compression may improve the storage size. Second, domain-specific knowledge can be explored such as removing temporary files [45]. Another limitation is the memory overhead to store the node map as in Table 2, which is the only extra data other than the events. Our experiment results show that the node map takes 114 MB on disk, but consumes 1.4 GB when loaded into memory. The memory cost can be reduced by replacing generic hash maps of Java with user-defined ones.

Potential attacks. When the adversary compromises end-hosts and back-end servers, she can pro-actively inject/change/delete events to impact the outcome of SEAL. Log integrity needs to be ensured against such attacks, and the existing approaches based on cryptography or trusted execution environment [9, 40, 60, 64, 71] can be integrated to this end.

One potential attack against SEAL is denial-of-service attack. Though delta coding and Golumb coding are applied to compress edges, all timestamps have to be "remembered" by the new edge. The adversary can trigger a large number of events to consume the storage. This issue is less of a concern for approaches based on data reduction, as those edges will be considered as repeated and get pruned. Moreover, knowing the algorithm of compression ratio estimation, the adversary can add/delete edges and nodes to mislead the estimation process to consider each block incompressible. On the other hand, such denial-of-service attack will make the performance of casualty analysis fall back to the situation when no compression is applied at most. The analysis accuracy will not be

impacted. Besides, by adding/deleting an abnormal number of events, the attacker might expose herself to anomaly detection methods.

Out-of-order logs. Due to reasons like network congestion, logs occasionally arrive out of order at the back-end analysis server [84]. Since the dependency graph possesses temporal locality, such "out-of-order" logs result in potential impact on the compression ratio. This issue can be addressed by the method described as follows. Assuming the probability of out-of-order logs is p, the server can reserve pN temporary storage to hold all out-of-order logs in a day, where N is the daily uncompressed log size. During off-peak hours, the server can process each out-of-order log. For log from Node u to Node v, we 1) retrieve in the compressed data the merged edges to v and decompress the timestamps, and 2) merge the edge (u,v) with the retrieved edges and compress the timestamps. Since the probability p is typically small and off-peak hours are utilized, out-of-order logs can be handled with smoothly.

Generalizing SEAL. Though SEAL is designed for causality analysis in the log setting, it can be extended to other graphs/applications as well. Generally, SEAL assumes the edges of a graph have attributes of timestamp, and the application uses time range as a constraint to find time-dependent nodes/edges. Therefore, the data with timestamp and entity relations, like network logs, social network activities, and recommendations, could benefit from SEAL. Besides forensic analysis, other applications relying on data provenance, like fault localization, could be a good fit. We leave the exploration of the aforementioned data/applications as future work. In terms of the execution environment of SEAL, we assume SQL database stores the logs on a centralized server, like prior works [37, 45, 77, 83]. It is possible that the company deploying SEAL in a distributed environment (e.g., Apache Spark) with non-SQL-based storage. How to adjust SEAL to this new environment worth further research as well.

7 Related Works

Attack Investigation. Our work focuses on reducing the storage overhead of system logs while maintaining the same accuracy for attack investigation, in particular causality analysis. Nowadays, causality analysis is mainly achieved through back-tracking, which was proposed by King et al. [42]. This technique has been extended to scenarios like file system forensics [73] and intrusion recovery [26]. In addition to desktop computers, the technique has been applied to high-performance computers [15] and web servers [3, 4].

As keeping system logs from machines in an enterprise consumes paramount storage space, recent research efforts are focused on reducing such overhead. One main approach is to *remove* logs matching certain conditions, including edges with same source and target [83], temporary files [45], frequent invocation of programs [77] and identical events with multiple versions [37]. In addition, the labels provided by operating systems [76] and access control polices [5] have been leveraged for log removal. The fundamental difference of SEAL is that it *compresses* logs under various codes, such that *all* information useful for attack investigation is preserved.

As the system logs are generated by the logger of end-hosts, a number of works studied how to improve the efficiency of the logger. In particular, in-kernel cache [49], kernel-space hooks [65, 66], dual execution [43], execution partitioning [50], tracing-tainting alternation [51], on-demand information flow tracking [39], and library-aware tracing [81] have been proposed to enhance the logger for efficient provenance analysis. In addition to efficiency, the security of the logger itself has been investigated [6, 9]. Since SEAL is applied on the side of the data aggregator, it can complement the approaches on the side of end-hosts.

After the logs are collected, how to connect and represent them is very important for effective attack investigation. Most of the effort has been spent on tailoring the logs for graph-based analytics [31, 33, 36, 57, 58, 70]. Besides, some works studied how to reconstruct the "crime scene" from the logs [46, 56, 67, 80]. Since the result of the log analytics needs to be processed by human analysts, research has been done to develop new domain-specific query languages [23–25, 44, 72] and task prioritization [34, 48] to reduce their workload.

Breach Detection in Enterprise Environment. Prior to attack investigation, the detection systems like Firewall, Web Proxy, and IDS deployed by the enterprise needs to identify the breach promptly and generate alerts. A line of research has been done to mine attacks from enterprise logs with machine-learning techniques. Classical machine-learning models like logistic regression and random forest have been applied to detect malicious domains from network logs [62, 63, 84] and malicious files from end-host logs [10]. Recently, deep-learning models like Long short-term memory (LSTM) [19] and embedding [47] are leveraged for the similar purposes. While the primary application of SEAL is on causality analysis, it could be adjusted for breach detection and we leave this exploration as future works.

Data Compression on Databases. Compression on databases can be traced back to Bassiouni [2] and Cormack [13] in 1985, where compression techniques are applied to the properties or fields of the records. Graefe and Shapiro [29] proposed to compress the properties and query on compressed data as much as possible. Compression across different properties was investigated in [68] in order to explore the dependency among related fields. Column-oriented database [1, 38, 75] stores each column of the database separately, and compression potential along the column can be explored. Compression of bitmap indexes has been developed in a variety of works, e.g., [12, 61, 74, 82] to improve query processing. We investigate a new application, causality analysis, with compression.

8 Conclusion

Causality analysis reconstructs information flow across different files, processes, and hosts to enable effective attack investigation and forensic analysis. However, it also requires a large amount of storage, which impedes its wide adoption by enterprises. Our work shows the concern about storage overhead can be eased by query-friendly compression. Comparing to prior works based on data reduction, our system SEAL offers similar or better storage (e.g., 9.81x event reduction and 2.63x database size reduction on DS_{ind}) and query efficiency (average query speed is 64% of the uncompressed form) with guarantee of no false positive and negative in casualty queries. We make the first attempt to integrating the techniques in the coding area (like Delta coding and Golumb coding) with a security application. We hope in the future more security applications can be benefited with techniques from the coding community and we will continue such investigation.

Acknowledgments

We thank our shepherd Birhanu Eshete and the anonymous reviewers for their helpful comments that improved the paper.

References

[1] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. Column-oriented database systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665, 2009.

[2] Mostafa A. Bassiouni. Data compression in scientific and statistical databases. *IEEE Transactions on Software Engineering*, (10):1047–1058, 1985.

[3] Adam Bates, Kevin Butler, Alin Dobra, Brad Reaves, Patrick Cable, Thomas Moyer, and Nabil Schear. Retrofitting applications with provenance-based security monitoring. *arXiv preprint arXiv:1609.00266*, 2016.

[4] Adam Bates, Wajih Ul Hassan, Kevin Butler, Alin Dobra, Bradley Reaves, Patrick Cable, Thomas Moyer, and Nabil Schear. Transparent web service auditing via network provenance functions. In *Proceedings of the 26th International Conference on World Wide Web*, pages 887–895, 2017.

[5] Adam Bates, Dave Tian, Grant Hernandez, Thomas Moyer, Kevin RB Butler, and Trent Jaeger. Taming the costs of trustworthy provenance through policy reduction. *ACM Transactions on Internet Technology (TOIT)*, 17(4):1–21, 2017.

[6] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. Trustworthy whole-system provenance for the linux kernel. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 319–334, 2015.

[7] Monica Bianchini, Marco Gori, and Franco Scarselli. Inside pagerank. *ACM Transactions on Internet Technology (TOIT)*, 5(1):92–128, 2005.

[8] VMware Carbon Black. Threat hunting and incident response for hybrid deployments. https://www.carbonblack.com/products/edr/, 2020.

[9] Kevin D Bowers, Catherine Hart, Ari Juels, and Nikos Triandopoulos. Pillarbox: Combating next-generation malware with fast forward-secure logging. In *International Workshop on Recent Advances in Intrusion Detection*, pages 46–67. Springer, 2014.

[10] Ahmet Salih Buyukkayhan, Alina Oprea, Zhou Li, and William Robertson. Lens on the endpoint: Hunting for malicious software through endpoint data analysis. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 73–97. Springer, 2017.

[11] Bryan Cantrill, Michael W Shapiro, and Adam H Leventhal. Dynamic Instrumentation of Production Systems. In *{USENIX} Annual Technical Conference (ATC)*, Boston, MA, 2004.

[12] Zhen Chen, Yuhao Wen, Junwei Cao, Wenxun Zheng, Jiahui Chang, Yinjun Wu, Ge Ma, Mourad Hakmaoui, and Guodong Peng. A survey of bitmap index compression algorithms for big data. *Tsinghua Science and Technology*, 20(1):100–115, 2015.

[13] Gordon V Cormack. Data compression on a database system. *Communications of the ACM*, 28(12):1336–1342, 1985.

[14] Cybereason. EDR | cybereason defense platform. https://www.cybereason.com/platform/endpoint-detection-response-edr. Accessed: 2020-2-15.

[15] Dong Dai, Yong Chen, Philip Carns, John Jenkins, and Robert Ross. Lightweight provenance service for high-performance computing. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 117–129. IEEE, 2017.

[16] DARPA/I2O. DARPA Transparent Computing. https://github.com/darpa-i2o/Transparent-Computing, 2020.

[17] Anirban Dasgupta, Ravi Kumar, and Tamas Sarlos. On estimating the average degree. In *Proceedings of the 23rd international conference on World wide web*, pages 795–806, 2014.

[18] Peter Deutsch et al. Gzip file format specification version 4.3. Technical report, RFC 1952, May, 1996.

[19] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298, 2017.

[20] Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. Query preserving graph compression. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 157–168, 2012.

[21] Uriel Feige. On sums of independent random variables with unbounded variance and estimating the average degree in a graph. *SIAM Journal on Computing*, 35(4):964–984, 2006.

[22] FireEye. Endpoint security software and solutions. https://www.fireeye.com/solutions/hx-endpoint-security-products.html, 2020.

[23] Peng Gao, Xusheng Xiao, Ding Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, Chung Hwan Kim, Sanjeev R Kulkarni, and Prateek Mittal. {SAQL}: A stream-based query system for real-time abnormal system behavior detection. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 639–656, 2018.

[24] Peng Gao, Xusheng Xiao, Zhichun Li, Kangkook Jee, Fengyuan Xu, Sanjeev R Kulkarni, and Prateek Mittal. A query system for efficiently investigating complex attack behaviors for enterprise security. *Proceedings of the VLDB Endowment*, 12(12):1802–1805, 2019.

[25] Peng Gao, Xusheng Xiao, Zhichun Li, Fengyuan Xu, Sanjeev R Kulkarni, and Prateek Mittal. {AIQL}: Enabling efficient attack investigation from system monitoring data. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 113–126, 2018.

[26] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal De Lara. The taser intrusion recovery system. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 163–176, 2005.

[27] Oded Goldreich and Dana Ron. Approximating average parameters of graphs. *Random Structures & Algorithms*, 32(4):473–493, 2008.

[28] Solomon W Golomb, Basil Gordon, and Lloyd R Welch. Comma-free codes. *Canadian Journal of Mathematics*, 10:202–209, 1958.

[29] Goetz Graefe and Leonard D Shapiro. *Data compression and database performance*. University of Colorado, Boulder, Department of Computer Science, 1990.

[30] Xueyuan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. UNICORN: Runtime Provenance-Based Detector for Advanced Persistent Threats. In *NDSS*, San Diego, CA.

[31] Xueyuan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. Unicorn: Runtime provenance-based detector for advanced persistent threats. *arXiv preprint arXiv:2001.01525*, 2020.

[32] Morris H Hansen and William N Hurwitz. On the theory of sampling from finite populations. *The Annals of Mathematical Statistics*, 14(4):333–362, 1943.

[33] Wajih Ul Hassan, Lemay Aguse, Nuraini Aguse, Adam Bates, and Thomas Moyer. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *Network and Distributed Systems Security Symposium*, 2018.

[34] 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.

[35] Kelly Jackson Higgins. The rebirth of endpoint security. https://www.darkreading.com/%20endpoint/the-rebirth-of-endpoint-security/d/d-id/1322775, 2015.

[36] Md Nahid Hossain, Sadegh M Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R Sekar, Scott Stoller, and VN Venkatakrishnan. {SLEUTH}: Real-time attack scenario reconstruction from {COTS} audit data. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 487–504, 2017.

[37] Md Nahid Hossain, Junao Wang, Ofir Weisse, R Sekar, Daniel Genkin, Boyuan He, Scott D Stoller, Gan Fang, Frank Piessens, Evan Downing, et al. Dependence-preserving data compaction for scalable forensic analysis. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1723–1740, 2018.

[38] S Idreos, F Groffen, N Nes, S Manegold, S Mullender, and M Kersten. Monetdb: Two decades of research in column-oriented database. *IEEE Data Engineering Bulletin*, 2012.

[39] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 377–390, 2017.

[40] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. Sgx-log: Securing system logs with sgx. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 19–30, 2017.

[41] Liran Katzir, Edo Liberty, and Oren Somekh. Estimating sizes of social networks via biased sampling. In *Proceedings of the 20th international conference on World wide web*, pages 597–606, 2011.

[42] Samuel T King and Peter M Chen. Backtracking intrusions. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 223–236, 2003.

[43] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. Ldx: Causality inference by lightweight dual execution. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 503–515, 2016.

[44] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela F Ciocarlie, et al. Mci: Modeling-based causality inference in audit logging for attack investigation. In *NDSS*, 2018.

[45] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. Loggc: garbage collecting audit log. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1005–1016. ACM, 2013.

[46] Bo Li, Phani Vadrevu, Kyu Hyung Lee, Roberto Perdisci, Jienan Liu, Babak Rahbarinia, Kang Li, and Manos Antonakakis. Jsgraph: Enabling reconstruction of web attacks via efficient tracking of live in-browser javascript executions. In *NDSS*, 2018.

[47] Fucheng Liu, Yu Wen, Dongxue Zhang, Xihe Jiang, Xinyu Xing, and Dan Meng. Log2vec: A heterogeneous graph embedding based approach for detecting cyber threats within enterprise. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1777–1794, 2019.

[48] 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.

[49] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. Kernel-supported cost-effective audit logging for causality tracking. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 241–254, 2018.

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

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

[52] Antonio Maccioni and Daniel J Abadi. Scalable pattern matching over compressed graphs via dedensification. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1755–1764, 2016.

[53] Emaad Manzoor, Sadegh M Milajerdi, and Leman Akoglu. Fast Memory-efficient Anomaly Detection in Streaming Heterogeneous Graphs. In *SIGKDD*, pages 1035–1044, New York, New York, USA, 2016. ACM Press.

[54] Hossein Maserrat and Jian Pei. Neighbor query friendly compression of social networks. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 533–542, 2010.

[55] Microsoft. Event tracing for windows (etw). https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/event-tracing-for-windows--etw-, 2017.

[56] Sadegh M Milajerdi, Birhanu Eshete, Rigel Gjomemo, and Venkat N Venkatakrishnan. Propatrol: Attack investigation via extracted high-level tasks. In *International Conference on Information Systems Security*, pages 107–126. Springer, 2018.

[57] Sadegh M Milajerdi, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1795–1812, 2019.

[58] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, R Sekar, and VN Venkatakrishnan. Holmes: real-time apt detection through correlation of suspicious information flows. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1137–1152. IEEE, 2019.

[59] Jeffrey Mogul, Balachander Krishnamurthy, Fred Douglis, Anja Feldmann, Yaron Goland, Arthur van Hoff, and D Hellerstein. Delta encoding in http. *IETF, Gennaio*, 65, 2002.

[60] Hung Nguyen, Radoslav Ivanov, Linh TX Phan, Oleg Sokolsky, James Weimer, and Insup Lee. Logsafe: secure and scalable data logger for iot devices. In *2018 IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 141–152. IEEE, 2018.

[61] Patrick E O'Neil. Model 204 architecture and performance. In *International Workshop on High Performance Transaction Systems*, pages 39–59. Springer, 1987.

[62] Alina Oprea, Zhou Li, Robin Norris, and Kevin Bowers. Made: Security analytics for enterprise threat detection. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 124–136, 2018.

[63] Alina Oprea, Zhou Li, Ting-Fang Yen, Sang H Chin, and Sumayah Alrwais. Detection of early-stage enterprise infection by mining large-scale log data. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 45–56. IEEE, 2015.

[64] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher W Fletcher, Andrew Miller, and Dave Tian. Custos: Practical tamper-evident auditing of operating systems using trusted execution. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*, 2020.

[65] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 405–418, 2017.

[66] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, and Margo Seltzer. Runtime analysis of whole-system provenance. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1601–1616, 2018.

[67] 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 *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, pages 583–595, 2016.

[68] Vijayshankar Raman and Garret Swart. How to wring a table dry: Entropy compression of relations and querying of compressed relations. In *Proceedings of the 32nd international conference on Very large data bases*, pages 858–869, 2006.

[69] Redhat. Chapter 7. system auditing. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security_guide/chap-system_auditing, 2019.

[70] Omid Setayeshfar, Christian Adkins, Matthew Jones, Kyu Hyung Lee, and Prashant Doshi. Graalf: Supporting graphical analysis of audit logs for forensics. *arXiv preprint arXiv:1909.00902*, 2019.

[71] Carlton Shepherd, Raja Naeem Akram, and Konstantinos Markantonakis. Emlog: tamper-resistant system logging for constrained devices with tees. In *IFIP International Conference on Information Security Theory and Practice*, pages 75–92. Springer, 2017.

[72] Xiaokui Shu, Frederico Araujo, Douglas L Schales, Marc Ph Stoecklin, Jiyong Jang, Heqing Huang, and Josyula R Rao. Threat intelligence computing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1883–1898, 2018.

[73] Sriranjani Sitaraman and Subbarayan Venkatesan. Forensic analysis of file system intrusions using improved backtracking. In *Third IEEE international workshop on information assurance (IWIA'05)*, pages 154–163. IEEE, 2005.

[74] Michał Stabno and Robert Wrembel. Rlh: Bitmap compression technique based on run-length and huffman encoding. *Information Systems*, 34(4-5):400–414, 2009.

[75] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: a column-oriented dbms. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pages 491–518. 2018.

[76] Yujuan Tan, Hong Jiang, Dan Feng, Lei Tian, and Zhichao Yan. Cabdedupe: A causality-based deduplication performance booster for cloud backup services. In *2011 IEEE international parallel & distributed processing symposium*, pages 1266–1277. IEEE, 2011.

[77] 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 *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1324–1337. ACM, 2018.

[78] Trustwave. Trustwave global security report, 2015.

[79] J Uthayakumar, T Vengattaraman, and P Dhavachelvan. A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications. *Journal of King Saud University-Computer and Information Sciences*, 2018.

[80] Phani Vadrevu, Jienan Liu, Bo Li, Babak Rahbarinia, Kyu Hyung Lee, and Roberto Perdisci. Enabling reconstruction of attacks on users via efficient browsing snapshots. In *NDSS*, 2017.

[81] Fei Wang, Yonghwi Kwon, Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Lprov: Practical library-aware provenance tracing. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 605–617, 2018.

[82] Kesheng Wu, Ekow J Otoo, and Arie Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems (TODS)*, 31(1):1–38, 2006.

[83] 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 *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 504–516. ACM, 2016.

[84] Ting-Fang Yen, Alina Oprea, Kaan Onarlioglu, Todd Leetham, William Robertson, Ari Juels, and Engin Kirda. Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 199–208, 2013.

A Compression Ratio as a function of Average Degrees

In this section, we derive explicit expressions for the compression ratio. We show that the compressed graph size is always smaller than the original size, though new vertices might be introduced.

In a dependency graph $G = (V, E)$, the number of vertices (nodes) is denoted by $n = |V|$, and the number of edges is denoted by $m = |E|$. Recall that the edges are directed and multiple edges (repeated edges) may exist from one node to another. For node $v \in V$, let its number of parent nodes be p_v, and its number of incoming edges be m_v. We have $m = \sum_{v \in V} m_v$. Moreover, denote by $p = \sum_{v \in V} p_v$ the total number of parent nodes for all nodes in V. Therefore, p represents the number of edges of G after removing repeated ones. Let $G_{undirected}$ denote the undirected graph which is identical to G except that edge directions are removed. Let G_{simple} denote the simple graph obtained by removing the edge directions and the repeated edges from G. The average degree of the graph $G_{undirected}$ is denoted by d_{avg}. Then,

$$d_{avg} = \frac{2m}{n} = \frac{2\sum_{v \in V} m_v}{n}. \tag{4}$$

The average degree of G_{simple} is denoted by p_{avg}, which is

$$p_{avg} = \frac{2p}{n} = \frac{2\sum_{v \in V} p_v}{n}. \tag{5}$$

Denote by S_{event}, S_{node} the event and node sizes before compression, and by S'_{event}, S'_{node} the sizes after compression. They can be calculated by

$$S_{event} = \sum_{v \in V} m_v C_{event}, \tag{6}$$

$$S'_{event} = \sum_{v \in V : m_v > 1} \left(C_{event} + 2 m_v C_{\Delta} \right) + \sum_{v \in V : m_v = 1} C_{event}, \tag{7}$$

$$S_{node} = n C_{node}, \tag{8}$$

$$S'_{node} = n C_{node} + size\_map. \tag{9}$$

Here $C_{event} = 105$ (measured in bytes) is the size of all attributes of an event in the uncompressed format. In our database, C_{event} includes the sizes of `starttime` , `endtime`, `agentid`, etc. C_{Δ} is the delta-encoded data and separator size for each time entry, and the factor 2 reflects that two time attributes are recorded for every event. For most of the cases we have observed, $C_{\Delta} \leq 4$ bytes. C_{node} is the size of one node entry in the uncompressed format, including the size of `nodeid`, `nodename`, etc. Finally, *size_map* is the node map shown in table 2, and can be expressed as

$$size\_map = \sum_{v \in V} C_{ID}(p_v + 1). \tag{10}$$

Here $C_{ID} = 4$ is the constant size required for each `nodeid`. The above size parameters depend on the particular database

attributes, and to allow for an arbitrary database design, we use the general expressions instead of the particular sizes. In our experiments, S_{node} and S'_{node} take a negligible fraction of the total storage. As a result, we ignore the node sizes in the following calculations. However, an exact calculation can be carried out if the node size is comparable to the event size.

The difference between the original size and the compressed size is:

$$S_{event} - S'_{event} \tag{11}$$

$$= \sum_{v \in V : m_v > 1} \Big(m_v (C_{event} - 2C_\Delta) - C_{event} \Big). \tag{12}$$

It is obvious that the compressed size is always smaller than the original size if $C_{event} > 2C_\Delta$, which is true in our deployment. The compression ratio can be expressed as

$$ratio = \frac{S_{event}}{S'_{event}} \tag{13}$$

$$\geq \frac{\sum_{v \in V} m_v C_{event}}{\sum_{v \in V} \Big(C_{event} + 2 m_v C_\Delta \Big)} \tag{14}$$

$$= \frac{m C_{event}}{n C_{event} + 2 m C_\Delta} \tag{15}$$

$$= \frac{d_{avg} C_{event}}{2 C_{event} + 2 d_{avg} C_\Delta} \tag{16}$$

Equation (14) holds because we remove the condition $m_v > 1$ in the denominator, and thus we obtain a lower bound on the ratio. In Equation (16) we multiply the numerator and the denominator by $\frac{2}{n}$ and used Equation (4). If the node size is also included, the ratio will also depend on p_{avg} defined in Equation (5).

Remark. 1) Let us call the dependency graph "incompressible" if its compression ratio is lower than a given threshold. It is often unacceptable to compress graphs that are incompressible. Therefore, our estimated compression ratio is a lower bound of the exact ratio (e.g., the inequality of Equation (14)). 2) The discussion in Section 3.6 assumes that the node map size is negligible. If it is not, we also need to estimate p_{avg}. Note that p_{avg} corresponds to the average degree of the simple graph G_{simple}, one can simply apply Algorithm 2 to $H = G_{simple}$.

B Average Degree Estimator Evaluation

We measure the performance of the compression ratio (or average degree) estimator on our dataset following Algorithm 2. We run the algorithm on 8 chunks, each containing 10^6 events. For each chunk, 20 independent trials are conducted. The parameters are chosen to be $\theta = 10, p_{jump} = 0.1$, such that the estimation error is minimized for the chunks in the experiment. We measure the mean squared error (MSE) between the estimated average degree $\hat{d}$ and the true average degree d_{avg}, averaged over all trials and all chunks in the experiment. The results are shown in Figure 10. The MSE value has an obvious drop as the sample size percentage grows up to 5% and quickly converges when the samples cover half of the whole trunk. We then set 5% as the sample size.

Figure 10 also shows that our method has better accuracy compared to the naive estimator, which estimates d_{avg} by directly calculating the average degrees of uniformly sampled nodes.

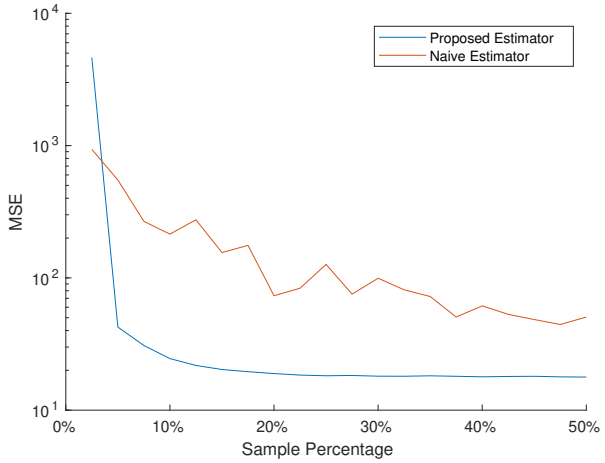

Figure 10: The y axis is the mean square error distance between the estimated average degree from the sampled data and the true average degree, averaged over all trials and all chunks in the experiment. The x axis is the percentage of the sample.