GrAALF:Supporting Graphical Analysis of Audit Logs for Forensics

Omid Setayeshfar
University of Georgia
Athens, Georgia
omid.s@uga.edu

Christian Adkins
University of Georgia
Athens, Georgia
caadkins@uga.edu

Matthew Jones
University of Georgia
Athens, Georgia
matthew.jones@uga.edu

Kyu Hyung Lee
University of Georgia
Athens, Georgia
kyuhlee@cs.uga.edu

Prashant Doshi
University of Georgia
Athens, Georgia
pdoshi@cs.uga.edu

ABSTRACT

System-call level audit logs often play a critical role in computer forensics. They capture low-level interactions between programs and users in much detail, making them a rich source of insight on malicious user activity. However, using these logs to discover and understand malicious activities from a typical computer that produces more than 3GB of logs daily is both compute and time intensive.

We introduce a graphical system for efficient loading, storing, processing, querying, and displaying system events to support computer forensics called GrAALF. In comparison to other related systems such as AIQL [13] and SAQL [12], GrAALF offers the flexibility of multiple backend storage solutions, easy-to-use and intuitive querying of logs, and the ability to trace back longer sequences of system events in (near) real-time to help identify and isolate attacks. Equally important, both AIQL and SAQL are not available for public use, whereas GrAALF is open-source.

GrAALF offers the choice of compactly storing the logs in main memory, in a relational database system, in a hybrid main memory-database system, and a graph-based database. We compare the responsiveness of each of these options, using multiple very large system-call log files. Next, in multiple real-world attack scenarios, we demonstrate the efficacy and usefulness of GrAALF in identifying the attack and discovering its provenance. Consequently, GrAALF offers a robust solution for analysis of audit logs to support computer forensics.

KEYWORDS

Security, System Forensics, Provenance

1 INTRODUCTION

Provenance data contained in system logs offers a rich source of information for computer forensics. This is because the log data comprehensively represents how processes controlled by an attacker interact with resources such as the disk and the network. System events loggers such as Linux Audit [19], Sysdig [38], DTrace [1], and Event Trace for Windows (ETW) [3] are often used to generate these logs. Although computer forensics logs are frequently analyzed off-line after an attack has been performed, real-time system monitoring can help the user in various ways. If forensic logs can be analyzed in real-time, that rich source of information allows investigation of ongoing abnormal behaviors and thus can protect the system effectively. Also, timely attack investigation is essential to protect the system from future similar attacks.

However, there exist three main challenges in developing a practical system that can support (near) real-time analysis. First, system logs grow rapidly. We observe that a single host generates more than 2.5 million system events in an hour. Other studies [14, 21, 27] also reported this problem. This challenge has motivated previous research into both lossless and lossy compression of logs [18, 27, 32, 39, 41] as well as optimizing querying and pattern matching in these logs [12, 13].

Next, there exist various logging systems for different operating systems and system architectures, but their definitions of the event entries, as well as output formats, are often different. For instance, Linux and Unix systems frequently use Linux Audit [19] to log system events. On the other hand, Windows systems mostly use Event Tracing for Windows (ETW) [3] to record system events, including system calls and Windows API calls. Linux and Windows have completely different system calls and system APIs. Furthermore, recent studies [5, 26, 34, 37] propose novel logging techniques to improve the effectiveness and efficiency of computer forensics. For instance, BEEP [26] proposes a technique to divide the long-running process into a finer-grain system object called *execution unit* to improve forensic accuracy. It is challenging to seamlessly support system logs generated by various logging platforms because they have different definitions of system objects, subjects, and relations.

Third, backward and forward tracking techniques [14, 21] are the essential techniques for computer forensics to understand causal relations between system objects (e.g., process) and subjects (e.g., file, network socket). However, they often require tracking back to previous events to identify causal chains. In practice, database solutions are often used to store sequences of system events, and the user can compose queries to identify causal relations between system components. However, we observe that today's database solutions cannot provide enough performance to process real-time, streaming system events. In this study, we evaluate the insertion and querying performance of relational and graph databases. We observe that the relational databases show very good data insertion performance; however, the response time of backtrack querying is not acceptable. It can take more than 20 minutes to extract causal relations of multiple files from a two-day audit log. On the other hand, graph databases show good response times for the same backtrack query as they store such relations explicitly in a graph format. However, the data insertion performance is not acceptable for processing real-time system events.

In this paper we present GrAALF, a graphical system for efficient loading, storing, processing, querying, and displaying system events to support computer forensics. GrAALF effectively facilitates forensic analysis in a large enterprise.

GrAALF flexibly offers the options of compressed in-memory storage, a traditional relational database system, and a graph database for storing the parsed audit logs. Though relational databases are highly optimized for the storage of tabular data, they may not perform well in cases with a large number of chained joins. This slows down those operations that require an event stored in the database to be backtracked to its origin. On the other hand, graph databases are designed with relationships in mind, making them much more efficient for backtracking relationships between events in a security log. However, graph databases tend to have low insertion performances, due to which they are unable to keep up with high-speed streams of logged data. Consequently, graph databases and in-memory storage are well suited for (near) real-time forensics when mini-batches of data are inserted and fast query execution on graphs is needed. For post-mortem analyses, the relational database is more appropriate as it allows fast loading, indexing, and subsequent querying of huge amounts of log data.

GrAALF allows stored audit logs to be queried using a simple query language whose syntax and semantics are close to those of SQL. Importantly, and unlike SQL, this query language supports path queries and backtracking to an arbitrary depth from an identified resource. We note that this latter functionality is particularly crucial for successful forensics of popular computer attacks such as data exfiltration and kernel injections. User queries are parsed and interpreted by GrAALF's query and visualization layer; then the satisfying data is displayed as a color-coded graph or tree that can be rearranged, focused, and magnified for study.

GrAALF belongs to a growing family of systems that support forensic analysis, which include Elastic [7], AIQL [13], SAQL [12], and Loglens [10]. However, these previous systems support simple keyword- and regular-expression based log data filtering only, and do not offer the useful backward trace query functionality. Furthermore, systems such as AIQL and SAQL are proprietary and not available for public use, in contrast with GrAALF, which is open-source and free [1].

We evaluate GrAALF on a very large system call audit log of a real-world attack. We explore the performances of the various storage options as logs of increasing sizes are processed. Next, we report on the time taken to execute various types of queries that are relevant for forensics, including those involving substantial backtracking. Our experiments reveal that GrAALF scales linearly with the number of records processed and also demonstrate queries for which one storage option is better than others. Finally, we briefly discuss a few case studies that illustrate the pragmatic use of GrAALF in computer forensics.

The rest of the paper is organized as follows. In section 2, we present a high-level overview of GrAALF. We describe details of three core layers of GrAALF in the next three sections; we discuss the log ingestion layer in Section 3, the log storage layer in Section 4, and the query and visualization layer in Section 5. In section 6, we use large, real-world system logs to demonstrate the practical utility of GrAALF. We summarize related work in Section 7 and Section 8 concludes the paper.

[1] https://github.com/omid-s/cyber_deception

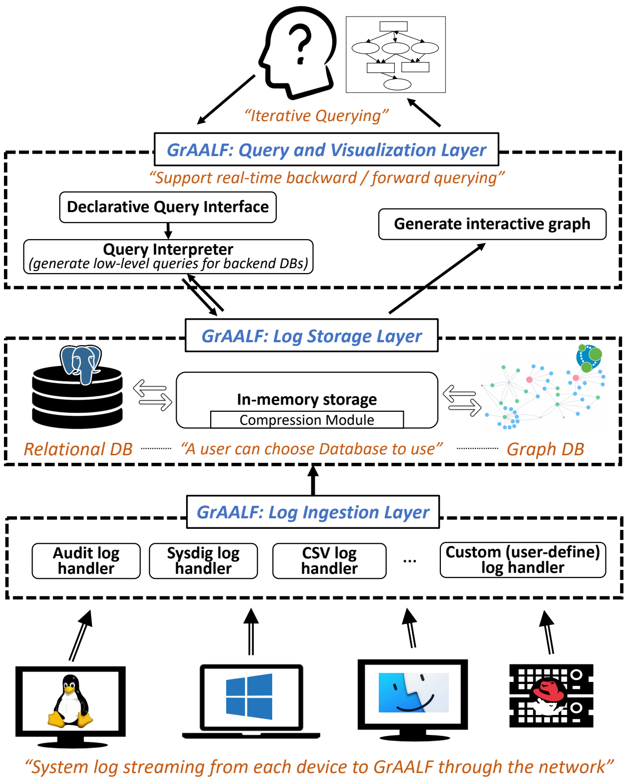
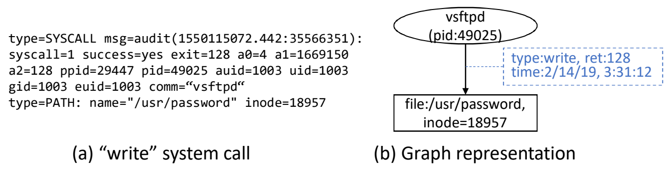

"Iterative Querying"
GrAALF: Query and Visualization Layer
"Support real-time backward / forward querying"
Declarative Query Interface
Query Interpreter
(generate low-level queries for backend DBs)
Generate interactive graph
GrAALF: Log Storage Layer
In-memory storage
Compression Module
Relational DB
"A user can choose Database to use"
Graph DB
GrAALF: Log Ingestion Layer
Audit log handler
Sysdig log handler
CSV log handler
...
Custom (user-define) log handler
"System log streaming from each device to GrAALF through the network"

Figure 1: A high level overview of GrAALF.

2 OVERVIEW OF GRAALF

We illustrate the high-level overview of GrAALF in Fig. 1. GrAALF consists of three layers: log ingestion, log storage, and query and visualization layers. The log ingestion layer receives streaming system logs from hosts in the enterprise and processes them. The output of this layer will be formatted data that represent causal relations between system subjects (e.g., process, thread) and objects (e.g., file, network socket).

The log storage layer stores the output from the log ingestion layer into a permanent database. We support both relational databases and graph databases and allow a user to choose which storage method is appropriate. We design in-memory buffer storage to enable the processing of enormous streaming data from multiple sources. This layer also handles user queries. It consistently passes sophisticated user queries to the corresponding storage method, including in-memory buffers and permanent databases. The output from various databases are seamlessly stitched in this layer and delivered to the upper layer.



```
type=SYSCALL msg=audit(1550115072.442:35566351):
syscall=1 success=yes exit=128 a0=4 a1=1669150
a2=128 ppid=29447 pid=49025 auid=1003 uid=1003
gid=1003 euid=1003 comm="vsftpd"
type=PATH: name="/usr/password" inode=18957
```

Figure 2: System call event and graph representation.

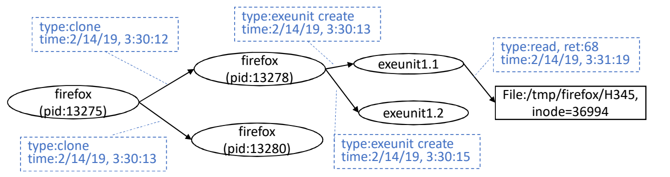


Figure 3: Representing a new type of subject, execution unit.

The query and visualization layer interacts with a user to receive queries and provide output as an interactive graph. The user can iteratively pose queries based on the prior output graphs. The output graph visualizes causal relations in the system elements as well as interactions between different machines. More importantly, GrAALF supports (near) real-time backward and forward queries through which the user can easily understand the origin of each system component and how one affects other components. We carefully design the storage layer and query interpreter for querying in-memory storage with the permanent database seamlessly. As soon as the ingestion layer processes the logs and delivers the output to in-memory storage, they are ready for querying.

The query and visualization layer also accepts queries to monitor the system for both stability and security purposes; this module will continuously monitor graphs produced by these queries and will notify the user when changes happen in any of them. This enables automated monitoring as well as automated detection of potential security incidents as they happen.

3 LOG INGESTION LAYER

System logs collected from hosts in the enterprise are streamed to GrAALF's log ingestion layer. This layer contains diverse log handlers to support various types of system logs from different platforms. For instance, most distributions of Linux and Unix systems use Linux Audit tool [19] to monitor system events such as system calls that reveal causal relations between system objects (e.g., process or thread) and subjects (e.g., file, network socket). On the other hand, Windows platforms mostly use Event Tracing for Windows (ETW) [3] for system monitoring by intercepting system calls along with Windows API calls. DTrace [1] is another popular tool that supports multiple platforms, including Linux, BSD, Solaris, and Windows. Commercial tools, such as Sysdig [38] and Dynatrace [2], are also available. Also, various approaches [5, 26, 34, 37] have proposed to improve the effectiveness and efficiency of computer forensics.

Our main design goal in this layer is to provide extensibility and pluggability in the system. We aim to support any types of log events that can be represented as object nodes, subject nodes, and their relation edges. For example, Fig. 2 (a) shows a system call event in Linux where a process *"vsftpd"* with pid 49025 reads a file *"\etc\password"* with inode 18957. This event is converted to a graph representation, as shown in Fig. 2 (b). Nodes contain information about the process and the file while the edges show how they are causally linked. In this manner, any system events that can be represented as graph nodes and causal relations between them can be ingested by GrAALF.

Fig. 3 demonstrates how GrAALF can support a new form of system objects and subjects. A series of recent studies [25, 26, 31] demonstrates that default system logging approaches suffer from the *dependency explosion* problem mainly due to long-running, event-handling processes. Process execution partitioning techniques [25, 26, 30–32] have been proposed to address the dependency explosion by introducing fine-grained system objects, called **execution units**, for both Windows and Linux platforms. In this example, we adopt an execution unit as another type of system object. Additionally, we introduce an edge from a thread which includes the target execution unit to represent the relation between the two. This edge contains a timestamp of the unit's creation time. Fig. 3 shows that the Firefox process with pid 13275 spawns two threads, 13278 and 13280, and 13278 creates two execution units and unit1.1 reads a file. In this way, a user can easily add system objects and subjects by adding a definition of a new node, its fields, and edges from/to existing subjects and/or objects.

In this project, we implement the following three log handlers to process logs created by different systems:

- **Audit Log handler** processes logs generated by Linux Audit [19]. It mainly contains system call events that contain process information (e.g., pid, parent process, user id, binary path, etc), system object information (e.g., file name, inode, network socket address, child process information, etc), and the description of event (e.g., system call number, arguments, return value, timestamp, etc). The format of Linux Audit log is a key-value pair record that directly derives from system call arguments, so it often only provides a file descriptor instead of file name, path, inode, or network socket information. To address this we maintain file open tables for each process to map file information, including an absolute path and inode, with a corresponding file descriptor. These tables are populated when we observe file open or other system calls that can manipulate the open file table (e.g., dup call to duplicate the open file descriptor). Then our handler converts an event into a graph format where subjects and objects become graph nodes, and an edge describes their relations and sends it to the storage layer. Additionally, we support modified Audit modules by [26, 31] to include execution unit objects. We follow the same definition of execution units and unit dependency as the original author described in [26], and we maintain unit tables for each thread to identify the parent thread that created the execution unit and also maintain the memory dependencies with other execution units.
- **Sysdig handler** handles logs generated by Sysdig system monitor [38]. Sysdig supports Linux, MacOS, and Windows. Similar to Linux Audit, Sysdig records system calls events. We support both output formats from Sysdig: json and plain text. Unlike Linux Audit, Sysdig provides target file and network information along

with file descriptor number; that reduces our burden, so we can directly convert Sysdig logs into our graph format.

- **CSV handler** is a generic log processing module we provide where the user can define fields in a CSV (comma-separated value) header. We believe supporting CSV format is particularly useful because CSV is a well-known and popular format for storing and sharing data. It is easy to define the nodes, their fields, and relations between them in the CSV header, and it helps to map data in the CSV to the internal object structure.

If a user desires to use a custom logging system, 1) they can use CSV as an output format that GrAALF can directly handle, or 2) they can implement a new handler that can parse events and pass the graph representation to GrAALF. Specifically, for each event, at least one object is required. Multiple objects and/or multiple subjects can exist in a single event. Each object and subject (i.e., node in a graph representation) has a unique signature (e.g., process id, file inode, etc) and unlimited fields for detailed information. These can be employed for user querying. An event can be represented as a directed edge between the object and the subject. There is no required field for edges, but we highly recommend providing a timestamp for each event to accurately identify indirect causal relations. For example, assume that *process A* reads *file1* and then writes *file2*. After that, *process A* reads *file3*. We can infer that *file2* might be causally related to *file1* through *process A*. However, read events of *file3* happen after the write event, and thus there are no causal relations between *file2* and *file3*. If we do not have timestamps in the edge, we conservatively conclude that both *file1* and *file3* (potentially) affect *file2*, and this causes a false dependence.

4 LOG STORAGE LAYER

GrAALF's log storage layer consists of two components: temporary in-memory storage and permanent database. We consider two popular back-end storage systems: table-based *relational* database and *graph-based* database systems. For the relational database, we mainly use *PostgreSQL* in this paper but GrAALF also fully supports *mysql* databases and it can easily be extended to support *Microsoft SQL Server* and *Oracle* with minimal query adjustments. While most traditional database implementations support *de facto* SQL language, newly emerging graph databases[6, 11] do not have a standard language. We use *Neo4j* as a backend graph database and GrAALF works well with Neo4j's Cypher[35] query language. We believe GrAALF can be extended to support other graph databases compatible with Cypher, but we do not guarantee that. To support a new graph database, GrAALF requires a certain amount of modification to adopt the new structures and query language.

We design this layer to support both relational databases (e.g., PostgreSQL) and graph databases (e.g., Neo4j) as backend storage systems. In addition to offering flexibility to the user to choose the favorite database system as backend storage, we show performance evaluations of each database that can help a user to choose the right one for their environment.

In general, queries about causal relationships between system subjects and objects perform better in a graph database because it does not require frequent table joins in processing the query. Graph databases directly store the relationships as edges between nodes and thus perform well in computer forensics. We measure querying performance with the following two scenarios:

- A forensic query from the user that returns an empty result. This query causes the database to scan the whole dataset but does not track any causal relationships.
- A forensic query to backtrack the lineage of all files in */home/* directory. This query causes the database to iteratively track causal relations for each file in */home/*. The output graphs contain all system nodes that directly or indirectly affect each file.

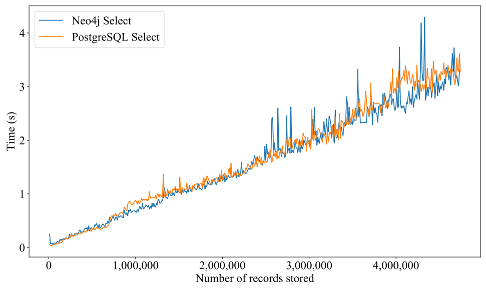
In this experiment, we use a real-world system log that we collect from a single host for 48 hours. The log is recorded by Linux Audit [19] that runs on top of Ubuntu-16.04. We execute each query with various sizes of datasets and measure the response time. Fig. 4a shows the performance when we execute the first query, which yields empty output. The performance of both databases is roughly linear to the size of the dataset. In this evaluation, we do not observe any noticeable differences between PostgreSQL and Neo4j. Fig. 4b shows the performance of the backtrack query. It shows that the graph database (i.e., Neo4j) performs better for the query that requires tracking relationships.

On the other hand, considering data insertion performance, graph databases do not scale well with a large stream of data. In this experiment, we use real-world system logs that we collect from a single host for 48 hours. The same as the previous experiment, we use the real-world system log recorded by Linux Audit [19]. We insert system events in the log to each database according to the corresponding timestamp. Fig. 5a shows the insertion delay caused by each database. PostgreSQL can ingest all events without noticeable delay. However, Neo4j shows a severe delay. For instance, many events have to wait in the queue for more than 20 minutes. Fig. 5b shows the length of the insertion queue required to process all events for each database. After 32 hours of the experiments, Neo4j requires more than 2 million additional insertions to handle the logs from a single host, and it rapidly grows. It clearly shows that Neo4j does not provide enough insertion performance to handle real-world system events.

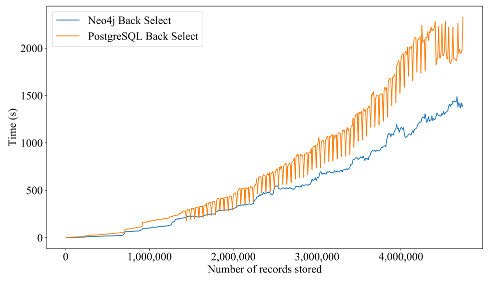
Note that in our evaluations, we do not try to configure or optimize each database; instead, we use default configuration to show how they perform with the forensics queries as well as insertion of system event data. For instance, both databases provide tools that can drastically speed up the insertions by potentially sacrificing the reliability of data. Neo4j offers a tool called *batch insert* that can significantly improve insertion speed, but it bypasses multiple layers of validations and thus could lead to inconsistencies in the final database. Additionally, there exist studies [12, 13, 17] that suggest methods for indexing and optimizing the data in different databases. Recently proposed high-performance databases [33] can be used as well, but that is out of our scope in this project.

4.1 In-memory Storage

To address the insertion delay observed in Neo4j and enable real-time querying for streaming inputs, we design and implement an in-memory storage system that can 1) maintain pending insertions in the system, 2) provide multiple graph compression techniques, 3) support seamlessly integrated querying between in-memory and database storage.

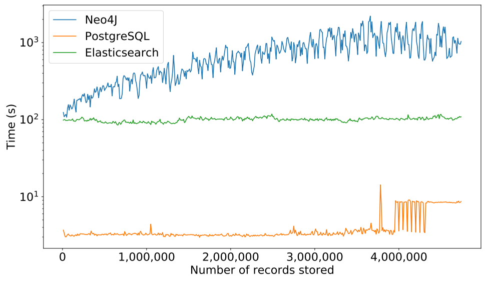


(a) Query returns an empty result

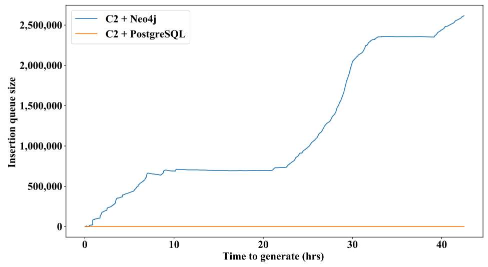


(b) Backtrack the lineage of files in ***/home/***

Figure 4: Querying performance comparison between Neo4j and Postgres



(a) Insertion delay (Y-axis is in log-scale)



(b) Length of insertion queue

Figure 5: Insertion performance comparison between Neo4j and Postgres

First, GrAALF's in-memory storage acts as an event buffer where events can be queued and wait for insertion into the backend database. Note that the user can query all events stored in the in-memory storage. After an event is inserted into the backend, we keep it in the in-memory storage only if the system has enough memory space for better querying performance. We discuss details of in-memory querying in the next section.

Next, we introduce three graph compression techniques to reduce storage cost and improve querying and insertion performance. All compressions are done while the events are stored in the in-memory storage to improve insertion performance as well as to save storage in the backend database. We have four options, as follows:

- **No Compression (C0):** This method returns all edges along with all of their metadata. We call it no-compression or C0. For instance, assume the system call event that Execution Unit (E_1.1) reads File *'/tmp/firefox/H345'* (F_1) 3 times. Three read events will be stored as separate edges.
- **Lossless Compression (C1):** This approach merges all edges if their object node, subject node, and the relation type are the same. We merge edges into a single edge but keep all fields, including time stamps, in order to preserve all of the information that each edge contains. With this approach, the output graph has fewer edges but the quality of information is the same as in C0. Applying C1 to the previous example, we merge all 3 edges into one but keep 3 timestamps.
- **Keeping Forensic Accuracy (C2):** This compression level also merges edges if their object, subject, and relation type are the same. However, unlike C1, we only keep the details of the first and last edges. With C2 used on the previous example, we merge 3 edges into one and keep only the first and last timestamps. This approach will lose some information (e.g., number of events between E_1.1 and F_1, total bytes that E_1.1 reads from F_1, etc) but we maintain the accuracy of backward and forward tracking output.
- **Lossy Compression (C3):** In this case, we keep the first edge and discard following edges if they have the same object, subject, and relation type. This approach will lose most of the ordering in events but still keeps the high-level causal relationship between nodes. More importantly, there is a chance that C3 causes bogus causality (i.e., false positives). For example, assume that E_1.1 reads from F_1 and later on, E_2.2 writes to F_1. With C0, C1, and C2, we can clearly tell that E_1.1 read F_1 before E_2.2 writes F_1, and thus, E_1.1 is not affected by E_2.2's write event. On the other hand, if we adopt C3, we do not have enough information about which event happens before the other, and we should conservatively assume that E_1.1 might be affected by E_2.2 via F_1. This is a false causality introduced by the lossy compression.

In this project, we use all four compression levels to evaluate the insertion performance and at runtime we allow the user to choose the compression level to use. We recommend C1 or C2 for most cases because C1 has the same quality of information as C0 with better performance, and C3 can cause false dependencies[29]. In

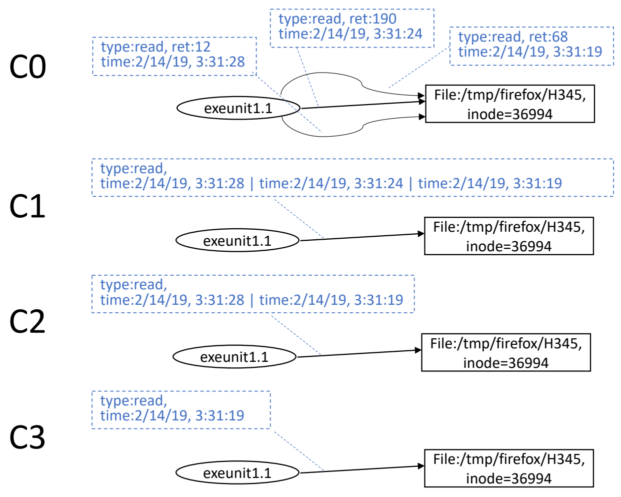


Figure 6: Shows different compression modes - C0 keeps all edges with all their meta information, C1 will merge similar edges but keeps all timestamps, C2 keeps only the first and last timestamp, C3 keeps only the first timestamp.

addition to that, we provide two query modes: *normal* and *verbose*. In verbose mode, each event will be represented using a single unique edge so that C1's verbose output is the same as C0. The normal mode groups similar edges into one to make the resulting graph more readable. Furthermore, we plan to adopt recent log reduction techniques [18, 27, 32, 41] into GrAALF as a part of graph creation or as an independent system to further reduce the size of the forensics graph. Fig. 6 shows the four compression models.

Fig. 8 shows insertion performance with different levels of compression. X-axis shows the event arrival time and Y-axis presents the time when the event is consumed by GrAALF. The dotted line shows the event arrival speed. If the insertion performance graph is above the dotted line, it means the processing is slower than the event arrival speed. If the performance graph lies below the dotted line, the processing speed is faster than event arrival and thus, acceptable for real-time event processing. As shown in Fig. 8, Neo4j alone is not acceptable for real-time event processing. Our in-memory storage with all compression levels can handle the data faster than event arrival speed and the performance of C1, C2, and C3 do not show much difference. In-memory storage with compressions can improve PostgreSQL's insertion performance a little, but it is already faster than event arrival speed. We do not show PostgreSQL results for better readability.

Fig. 7 shows the amount of memory consumed by the in-memory storage process when we use C2 compression. It includes the space for graph buffer as well as the space for compression operations. The process occupies a large amount of memory for the compression and frequently garbage collects to release the memory that is not required anymore, and that is the reason why the graph fluctuates heavily.

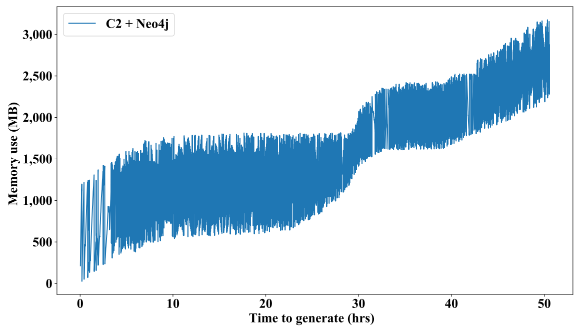


Figure 7: Amount of memory used to keep a forensics graph in memory for a single host

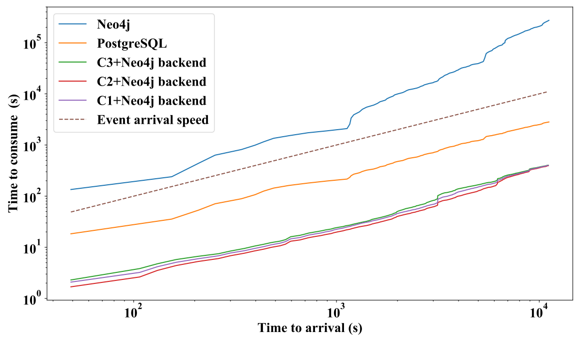


Figure 8: Compares how different configurations can handle data in GrAALF for a single machine stream (X-axis and Y-axis are in log-scale).

4.2 Graph Database Backend

GrAALF supports the storage of its graph in a graph database. To store records in a backend data store, and to minimize the memory used to keep them, we turn each record read from input into a small graph; the format of the log determines the size of this small graph. For example, if the graph is formatted as *object affects subject* in each record; then, the subject and object will be considered as nodes and "effect" will be represented as an edge. The created small graph structure is then passed to the graph database using a query that will first ensure the nodes do not already exist in the graph and then will insert them; any preexisting node will be kept to preserve space and queryability. Edge insertion, on the other hand, depends on the compression configurations. If C0 is selected, every edge is inserted in the database. However, with higher compression levels an edge is inserted only once, and future occurrences of the same edge are considered updates that will make changes to the time stamps on the first edge. GrAALF maintains a buffer of all the pending insertions. Buffer also checks all nodes and edges and prevents adding a node that already exists to the queue; this reduces the number of queries sent to the database.

In order to improve insertion performances, multiple backend databases can be utilized, each with data for a particular time window. This will improve insertion performance.

4.3 Relational Database Backend

GrAALF's relational data module supports PostgreSQL and MySQL database servers as a tabular back end. Relational databases can be used in two forms, 1) the flat tabular format, 2) the two-table graph format. In order to store data in a flat tabular format, we use one table with as many fields as in the internal record object. This opens room for direct storage and retrieval of those records; on the other hand, this comes at storage cost as some of the information will be copied multiple times. Also, as we will show in the evaluation sections, querying information from a relational database for longer backtracking tasks is more time consuming compared to running the same query in a graph database. To store logs in the flat format, each record passed to the storage module from the reader modules will turn into an insert query for one record with all of its fields that have values.

The two-table graph model is implemented using two tables in a relational database: a vertex table and an edge table. The creation of the nodes and edges happens the same as in the graph database backend, but nodes are inserted into the vertex table and edges are inserted in the edge table. Queries sent to the backend for retrieving data in this format have both a higher number and more complexity than the graph database counterparts.

5 QUERY AND VISUALIZATION LAYER

5.1 Query language

GrAALF's query language supports projection and filtering. The returned graph always starts from the nodes specified by the filter criteria and then applies the edge criteria if applicable.

The general structure of our query language is as follows :

```
[verbose] [[back]/[forward]] select * ,[ Projection of Edge Types
] from *,[ projection of Node Types] [where [[field] [operator]
[value]] ] [;]
```

- **[Projection of edge types]:** this option is either * for all, or is a selection of event types for edges. For instance, if the dataset represents system call events, edge types can be one or more system calls (e.g., read, write, open, execve) that the user wants to track.
- **[Projection of node types]:** this option is either * for all, or is a selection of nodes (i.e., system objects and subjects). For instance, *file, process, socket, pipe*, can all be types of nodes for the system call datasets.
- **[[field] [operator] [value]]:** these parameters can be any of the types described in the projection of node section. This part is optional, as you can skip it or use 'any' to consider all types in the query processing. [field] can be any field of a node. For example, 'pid' for a process id, 'name' for a process name, and 'uid' for a user id who launches the process can be used to selectively track process nodes in system call events. [operator] can be 'is', '>', '<' or 'has' for exact matches or the 'contains' operator; arguments to the "has" keyword can also be regular expressions as long as they conform to POSIX standard. Different criteria including ",and,"; ",or," logical operators are also supported.

For instance, if a user wants to identify all sources that affect a file "/etc/active_users.txt", they can compose the following backtrack query:

```
back select * from * where filename is '/etc/active_users.txt'
```

Similar to other query languages, users can also issue queries to get or set environment variables and configurations. These environment variables include: the maximum number of edges forward or backward tracking will explore; whether to force the system to merge a new output into the previous graph; and display details of query execution including the execution status of the storage layer and back-end database.

Users may choose to see the resulting graph in normal or verbose mode. Normal mode will group edges of a similar type between the same nodes, and this will produce a more straightforward graph for better readability. On the other hand, the verbose mode can be activated by adding "verbose" keyword to the beginning of the query to create separate edges for each type of relation.

Output graphs can be exported to various formats. We currently support textual description, JSON format, and DOT, and GrAALF can be extended to new formats. This is useful to use the output from GrAALF in other tools for further analysis. The "describe" keyword can be used in a query to include textual description of the output graph. The output will be the list of events that can be sorted by time or any fields. Sort can be selected by issuing an "orderby=[field]" keyword in the describe query. A user can use "format=" keyword to determine the output format including 'text', 'json' or 'dot'.

5.2 Query Interpreter

GrAALF translates a user query into one or more queries and fetches the corresponding data from in-memory storage and backend databases in the log storage layer. When a user issues a query via GrAALF's console, our query execution engine parses the query and decides whether the query is a configuration query which will make changes to program settings or if it is a query to fetch data from the forensics graph. Configuration queries will be sent to the configuration manager who is in charge of maintaining the parameters and configurations of GrAALF. Data-fetch queries will be routed to their corresponding module based on the selected back-end. This module runs as a separate thread to allow the user to interact with GrAALF's console while the previous query is being processed. As different backend databases use different query languages, we support both SQL-based and Neo4j's Cypher query languages. Regardless of the specific language used, we implement highly optimized in-memory query processing to allow the user to investigate real-time events.

5.2.1 In-memory Storage. This is the primary area in which we process the query. All incoming queries are analyzed here and outputs are quickly generated if the query can be fully answered without accessing backend databases. If it is required to access a backend database, we create queries according to the backend database types (e.g., graph or relational DB). All outputs from backend databases will be transferred into in-memory storage and seamlessly correlated into a single graph. This allows GrAALF to reduce graph construction efforts in other modules, while also providing a single interface to other modules to be able to consume the data.

The in-memory graph module maintains three hash maps which are used to create indexes on nodes and edges. One map contains information in a nested map model which starts with node type and then is mapped by their identifier. Another map uses a combination of identifiers from vertices to identify edges between every two nodes. The last hash map maintains a title-based map of the vertices; this will help with resolving regular expression and title matching queries faster. Select queries will start based on their criteria; e.g. if the user is looking for processes, the search navigates to the type hash map, finds processes, and then looks for title matches. If no title is expressed in the query, all nodes in that type will be considered. To lookup nodes based on their titles however, the search starts from the title hash map by finding keys that match the criteria and then processing their values. Edge selection happens after nodes are selected. To find edges we pick all combinations of two nodes and go through each, finding their corresponding edges from the edge map based on the combined identifiers of the nodes. If this search yields any results, those edges will be evaluated against edge criteria in the input query; e.g. if the user is asking for write system calls, returned edges should have write set as their system call. This helps efficiently find the desired nodes while maintaining a small overhead of nodes.

Back select queries start with a select query which finds the leaf nodes in the final graph; these nodes will be the nodes the user wants to backtrack from. Then a Breadth First Search traversal is performed on the forensics graph, backtracking from those nodes. After the first layer, all nodes will have only one parent; while many processes might access one file or socket, a process cannot be created by more than one process. This assumption makes back tracking significantly faster as there is no need to do redundant lookups, and many graph paths converge to processes like 'initd' or remote access clients like 'sshd'. Forward select queries are designed to see the effects one node has on other nodes. Like back selects, forward selects start with a selection of the initial nodes in question using a select query. Once the initial nodes (root or roots) are found, BFS will be performed on the nodes, following them level by level. Breadth First Search is chosen as a uniform search strategy in graph traversals for two main reasons: 1) each record in a log input will contain information about multiple levels of the graph. Thus DFS has a higher chance of performing redundant queries. 2) especially in backtracking, the parents will be a line graph starting from the first of the second layer after the initial nodes. While this could benefit either DFS or BFS, we stay with BFS throughout GrAALF to keep the model consistent. Once the nodes are explored, all nodes will be listed and their connecting edges will be added to the final graph from the edges map following the supplied edge filter criteria. To maintain the final results of queries as a graph data structure as well as its presentation, we leveraged JUNG [36] which provides a lightweight wrapper and user interface presentation for graphs and networks.

5.2.2 *Relational Database.*

Similar to in-memory storage, once the input query is parsed and the parent module handles configuration queries, the SQL module will look for the identified nodes in the database. If process information is given as a criterion, then process, parent process, and ancestor process fields are considered in the initial query and they are fetched. When criteria are provided for a socket or file, file descriptor fields are the main source of filtering in the query. By default, GrAALF stops running user queries with no criteria; this is because real life logs will flood the memory and the output will be partial, thus it cannot be accurate or trusted. In order to stop the misconceptions that would be caused by such skewed results, we stop such queries before running them. Select queries will stop at this stage; all nodes that have been discovered will be sent to the memory module to create the final filtered graph. Once seed nodes are discovered for a back select query, BFS is performed on the database following the trace. As mentioned in the previous section, each record contains information about the file or socket as well as its process, parent process, and ancestor process; thus each fetched record will create between one and four layers of depth in the final graph. For each new level in the graph, a new query is created and executed until there are no more layers in the graph to follow. While this requires many queries, GrAALF tries to lessen this workload by merging queries for different paths in each layer into one query to minimize the round trip times and transaction creation overheads. Forward select queries are handled in the same way as back selects, except that in a forward select the maximum output degree of nodes is not one. To handle forward queries, GrAALF runs the initial query to find the seed nodes. Then it will run BFS for the children of those nodes.

5.2.3 *Graph Database.*

This module is the simplest among the query modules described. A graph database will already have the data stored in a graph model; thus there is no need for tediously repeated queries. The graph-based module acts as an intermediary and will translate the user query into Cypher queries. Cypher queries are sent to the Neo4j database, and the returned records are formed into a graph which will be appended to the existing graph in the memory. Most GrAALF queries are translated into one optimized Cypher statement. Internal mechanisms of the graph database are trusted to be optimal when backtracking on paths of arbitrary lengths, thus no particular graph traversal method is applied in their case.

6 CASE STUDIES

In this section, we discuss a few example scenarios to show the practical utility of GrAALF.

6.1 Observation of Forensic Graph

In this section, we collect system logs from a real-world Linux device and discuss how the forensic graph evolves. In this experiment, a graduate student used a Linux desktop for typical daily work for 48 hours. The user heavily used Firefox web browser for surfing of various websites. The user also frequently used text editors, a compiler, and productivity software working with documents and worksheets.

To understand how forensic graphs evolve, we choose a subset of the data to study. We compose a query to identify all sources of files in the home directory (*/home/*). For each of these files, we count the number of related nodes and edges. We further execute backward tracking on all the nodes found in */home/* to find how their connectivity changes over time. We count the number of nodes and edges for every 1,000 new event arrivals. Fig. 9 shows the trend. Increasing connectivity is shown by the number of new edges introduced to the graph. The number of nodes increases as well, and that shows that the number of system subjects and objects

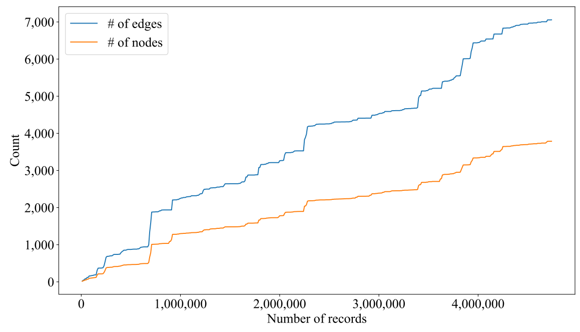


Figure 9: The number of edges and nodes in a forensics graph for files in */home/* directory on a Linux machine.

related to the target files increases over time. While the data we consider in this experiment is only a subset of the system logs, we choose the home directory as a considerable number of processes store files there and make changes to it. Also, most files the user touches are stored under the home directory, and we believe it shows the characteristics of regular user activity.

6.2 Case Studies

In this section, we use three scenarios to demonstrate the practical usability of GrAALF.

6.2.1 Background. In this study, we use two realistic cyberattacks produced by the red team from the DARPA Transparent Computing program [9]. The system log contains over 5 billion records from multiple servers and desktop environments. We demonstrate three cases in which GrAALF can help to investigate cyberattacks, as well as real-time monitoring tasks for early detection of malicious activities. As follows, we use two attack investigation scenarios and one system monitoring case:

- ***Attack Investigation 1*** : A user is investigating a data exfiltration attack. The attacker logs into the system via *ssh* and transfers an important file to a remote host using *csp* process.
- ***Attack Investigation 2*** : This attack involves a known backdoor in a *vsftp* server application. The backdoor opens on a known port when login into the FTP server is attempted using any user name including a smiley face sequence of ':)'. The attacker exploits the backdoor functionality to drop a shell script that causes the server to restart.
- ***System Monitoring*** : A system administrator is monitoring the system to check whether any process has been executed from an executable file that was downloaded through the network in the last 24 hours.

6.2.2 Data Exfiltration Attack. At the time of computer forensics work, the user has already been informed that an exfiltration attack has occurred. Thus the first step will be to find which important files have been exfiltrated. Fig. 10 shows the steps which the user needs to follow in order to trace the attack.

- *Step 1.* The user tries to find what file/files have been accessed during the given period. Running the first query will return a list of nodes in an important directory - for simplicity, we assume 'important files' - and the user then decides which files to track.
- *Step 2.* Next, the user will be interested in discovering what happened to the files by backtracking from them. Back select will provide the user with a hierarchy of processes that lead to accessing the file/files in question.
- *Step 3.* The user has noticed a suspicious process (eg. the user sees an *scp* process in the back select from the last query). In order to determine where the file has been sent to, the third query is used; this query will show the remote IP in question.

```
Q1: select * from file where name has /important-files/
Q2: back select * from * where name is /important-files/plan-file.cad ;
Q3: forward select * from soc where name is scp and pid is 4667 ;
```

Figure 10: A set of queries used for data exfiltration attack.

The final graph from the above 3 steps is shown in Fig. 11. Node colors represent the step at which the node has been added to the graph: *red* nodes are added in step 1, followed by *white* nodes and then *gray* nodes.

6.2.3 FTP Backdoor Attack. The user has seen a sudden restart in the system after admins are notified that a new shell file is found in the home directory; this information and the logs are passed to the forensics expert. The following steps show how the attack can be traced back to its source:

- *Step 1.* The expert locates the shell file by executing the command 'select * from * where name is myshell.sh'
- *Step 2.* The expert performs a back select from the file to see what processes led to the creation of the file. 'back select * from * where name is myshell.sh'
- *Step 3.* The expert finds an "sh" process which started a shell that wrote the shell file. Starting from that process, they do a forward trace to find other activities from that "sh" process; this leads to a socket on a remote computer, with IP 192.168.10.20. 'forward select * from * where pid is 24456 ,also, name is sh '

Using the steps listed above, the expert is able to reconstruct the attack's path. Based on the observed events, the expert concludes that the intrusion was achieved by creating a backdoor from the "vsftp" server; by accessing that backdoor, the attacker was able to drop a shell file into the system and execute it.

6.2.4 System Monitoring. In this task, the administrator is looking to see if any new executable has been downloaded on the computer and if they have been used to make changes to the contents of a file in the system.

- *Step 1.* The user looks for recent writes to files in the downloads directory. "back select write from * where file name has /home/user1/Downloads/ ,also, date has 2019-02-03"
- *Step 2.* The user has the list of recently downloaded files. The next step is to find which one they choose to trace.
- *Step 3.* The user chooses an executable file and performs a forward trace from that file. "forward select * from * where name is /home/user1/Downloads/dash".

With the steps mentioned above, the user will obtain a graph that shows the downloaded files and whether they have made changes to other files on that computer. Fig. 13 shows the steps of this experiment.

7 RELATED WORKS

Tools such as Elastic Stack are widely adopted in industry and academia for their agility and high performance when dealing with logs. Plaso [4] and SOF ELK [28] are two of the tools built on top of Elastic Stack; these tools have rich visualization and log parsing features; however, they provide statistics-based reports, do not work with provenance graph models, and do not provide users with much queryabilty beyond filtering features. These tools can be highly effective for monitoring or modeling the system state but are not as effective when it comes to provenance-tracking forensics querying.

Other tools such as [4, 8, 16] provide models based on artificial intelligence which will assist the forensics expert in both monitoring the system and detecting malicious behaviours based on known patterns; however, these tools are not designed for manual forensics tasks such as whole system provenance tracking and are often bound to one scheme of proprietary data stream. This limits their application by a human analyst by 1) imposing criteria on the data they present, such as a limited number of system calls, and 2) limiting freedom of exploration while querying.

GrAALF is designed to give the user more flexibility in both data source and data exploration while maintaining features present in other works such as pattern matching on the streaming data and provenance tracking capabilities. Table 1 shows an overview of some related works by their capabilities. We compare these works by:

- streaming analysis capabilities such as pattern matching or live monitoring and tracking.
- provenance backward and forward tracking; we use ■ when tracking on long sequences is possible and □ when the tool provides tracking functionality but it is not designed for long sequences from provenance graphs or can do tracking of events but does not provide a graph representation of provenance paths.
- flexibility in schema is the ability of the tool to support more than one type of input; e.g. Elastic Stack can process logs of arbitrary formats as long as they are provided in a key-value pair.
- open-sourced and freely available, where □ represents tools which are partially open-source or provide free versions.
- support for granularity smaller than process, which is unique to GrAALF and helps prevent dependency explosion; other similar tools stop their analysis granularity at process or thread level, but GrAALF supports finer grained units.
- queryability shows whether a tool provides an interface for users to sort through logs or not; ■ represents presence of query languages, and □ means there are some filtering capabilities that are not as advanced as a query language.
- presence of automated anomaly detection capabilities.

We also discuss some related works in more detail below.

SAQL [12] presents an online stream analysis tool that can detect predefined patterns in a fast stream of data. Patterns in this tool are defined using a query language that is capable of establishing time windows and calculating aggregations of events in those windows. The query language supports the definition of events separately, then connecting them to form a pattern. While the query language is very well designed to cover many tasks, the tool lacks constructs that focus on lineage of events (e.g., backward and forward tracking). Also, the query language is more complex than GrAALF's.

AIQL [13] presents a query tool that is capable of querying large sums of system provenance data. The query language is capable of backtracking from events in small steps through temporal orders. The system also provides a window-based aggregation model that allows users to query for abnormal behaviors, mostly in terms of frequency of events. The query language in this tool, much like in [12], is not designed for tracing of long sequences of system events but instead focuses on events that are within a close circle of forensics steps. Also, the tool works by leveraging a highly optimized database backend.

Winnower [15] is designed to cluster running tools like docker; it optimizes storage and reduces nodes by finding common graphs among different images. This relies on the fact that jobs running on these machines are heavily repetitive and similar. This allows the program to prune similar graphs from the network, which consists of many machines doing the same task. The tool also uses a query system capable of backward and forward tracking of provenance graphs with a Cypher-like query grammar. This tool is more oriented toward tasks with highly repeated components such as a computer cluster running the same or similar task; this idea does not apply to realistic program behaviors in an enterprise today.

Vast [40] focuses on a network-level log paradigm, but does handle logs in designing a distributed storage system that will keep and query network events. This system produces a complete graph of nodes in a distributed fashion where each node maintains an index and an archive of its own data. HDFS and bitmap index are being used to store the data and indexes. While GrAALF can work with a distributed data store as well, the integrity of system logs prevents us from trusting a computer with its logs; thus logs should be transported out as fast as possible. The query model in this work is also less flexible than that of the others.

There exist a number of causality analysis techniques [14, 20–23] that use system call loggers to record important system events at runtime and analyze recorded events to find causal relations between system subjects (e.g., process) and system objects (e.g., file or network socket). For instance, BackTracker [21] and Taser [14] propose backward and forward analysis techniques in order to analyze system call logs and construct causal graphs for effective attack investigation. Recently, a series of works [25, 26, 30–32] have proposed to provide accurate and fine-grained attack analysis. They divide long-running processes into multiple autonomous execution units and identify causal dependencies between them. LDX [24] proposes a dual execution-based causality inference technique. When a user executes a process, LDX automatically starts a slave execution by mutating input sources. Then LDX identifies causal dependencies between the input source and outputs by comparing the outputs from the original execution and the slave execution. MCI [25] leverages LDX [24] to build a model-based causality inference technique for audit logs to infer causal relations between system calls.

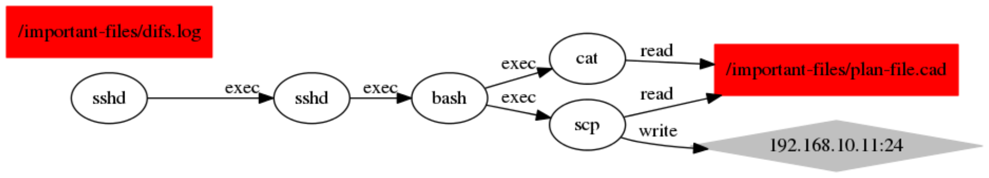


Figure 11: Attack 1. Graph, node colors represent the step of the investigation they have been added at; *red* nodes are added by first query, *white* nodes in the second, and *gray* ones in the third query.

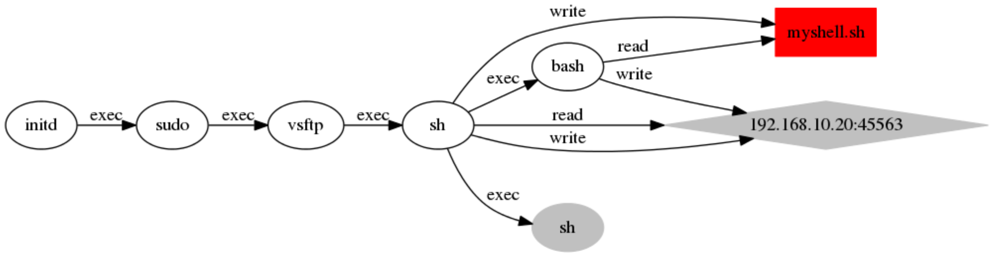


Figure 12: Attack 2. Graph, node colors represent the step of the investigation they have been added at; *red* nodes are added by first query, *white* nodes in the second, and *gray* ones in the third query.

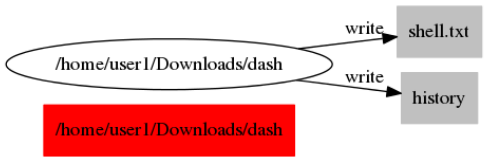


Figure 13: Monitoring Task 1. Graph, node colors represent the step of the investigation they have been added at; *red* nodes are added by first query, *white* nodes in the second, and *gray* ones in the third query.

Table 1: GrAALF Compared to Other Related Works

Title	S	PG	F	O	G	Q	A
GrAALF	■	■	■	■	■	■	×
AIQL	×	□	×	×	×	■	×
SAQL	■	□	×	×	×	■	×
SOF ELK	×	×	□	□	×	□	■
Carbon Black CB LiveOps	■	□	×	×	×	□	■
NoDoze	×	□	×	×	×	×	■
Plaso	×	×	□	□	×	□	■

Other common security tools compared to GrAALF by functionalities **S** for Streaming Analysis, **PG** for Provenance Graph Forward And Backward Tracking, **F** for Flexibility in Schema, **O** for Open-Source, **G** for Supporting Granularity Smaller than Process, **Q** for Queryability of the Graph, and, **A** for Intelligent Anomaly Detection. "×" shows lack of support, "□" shows support with limited functionality, and, "■" shows full support.

GrAALF can support all of the proposed logging techniques with the proper definition of subjects, objects, and relations. We can process their logs in real-time and provide GrAALF's query interface to the user to enable interactive investigation and monitoring of the system.

8 CONCLUSION

We present GrAALF, a graphical forensic analysis system for efficient loading, storing, processing, querying, and displaying of causal relations extracted from system events to support computer forensics. GrAALF offers the flexibility of choice between relational database (e.g., PostgreSQL) and graph database (e.g,. Neo4j) for backend storage. GrAALF's in-memory storage can be seamlessly integrated with either backend and provides (near) real-time forensic analysis of streaming system event logs. GrAALF provides a simple query language whose syntax and semantics are close to SQL. The user can simply compose a query to perform a backward and forward analysis to identify causal relations between system subjects and objects. We use an extensive system call audit log that contains realistic attacks to demonstrate the practical utility of GrAALF. We will release the source code of GrAALF as an open-source software.

ACKNOWLEDGEMENTS

This research is funded in part by grant # W911NF-18-1-0288 from the Army Research Office. Authors would also like to thanks Sean Frankum, Aditya Shinde and Muhammed AbuOdeh for their valuable inputs.

REFERENCES

[1] 2019. DTrace. http://dtrace.org/blogs/about. [Online; accessed 25 May 2019].
[2] 2019. Dynatrace. https://www.dynatrace.com. [Online; accessed 25 May 2019].
[3] 2019. Event Tracing for Windows. https://docs.microsoft.com/en-us/windows/desktop/etw/event-tracing-portal. [Online; accessed 25 May 2019].
[4] 2019. Plaso. https://plaso.readthedocs.io/en/latest/. [Online; accessed 25 May 2019].
[5] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. 2015. Trustworthy whole-system provenance for the Linux kernel. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 319–334.
[6] Inc Bryce Merkl Sasaki, Neo4j. 2019. Graph Databases for Beginners: Why a Database Query Language Matters (More Than You Think). https://neo4j.com/blog/why-database-query-language-matters/. [Online; accessed 25 May 2019].
[7] Elasticsearch B.V. 2019. Elasticsearch. https://www.elastic.co/. [Online; accessed 25 May 2019].
[8] Inc. Carbon Black. 2019. CB LiveOps. https://www.carbonblack.com/products/cb-liveops/. [Online; accessed 25 May 2019].
[9] DRAPA Transparent Computing. 2018. Transparent Computing Engagement 3 Data Release. https://github.com/darpa-i2o/Transparent-Computing. [Online; accessed 25 May 2019].
[10] Biplob Debnath, Mohiuddin Solaimani, Muhammad Ali Gulzar Gulzar, Nipun Arora, Cristian Lumezanu, Jianwu Xu, Bo Zong, Hui Zhang, Guofei Jiang, and Latifur Khan. 2018. LogLens: A Real-time Log Analysis System. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, IEEE, 1052–1062.
[11] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, ACM, 1433–1445.
[12] Peng Gao, Xusheng Xiao, Ding Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, Chung Hwan Kim, Sanjeev R Kulkarni, and Prateek Mittal. 2018. {SAQL}: A Stream-based Query System for Real-Time Abnormal System Behavior Detection. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. USENIX, 639–656.
[13] Peng Gao, Xusheng Xiao, Zhichun Li, Fengyuan Xu, Sanjeev R Kulkarni, and Prateek Mittal. 2018. {AIQL}: Enabling Efficient Attack Investigation from System Monitoring Data. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. USENIX, 113–126.
[14] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal De Lara. 2005. The taser intrusion recovery system. *ACM SIGOPS Operating Systems Review* 39, 5 (2005), 163–176.
[15] Wajih Ul Hassan, Lemay Aguse, Nuraini Aguse, Adam Bates, and Thomas Moyer. 2018. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *Network and Distributed Systems Security Symposium*.
[16] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. 2019. NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage.. In *NDSS*.
[17] Florian Holzschuher and René Peinl. 2013. Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*. ACM, 195–204.
[18] Md Nahid Hossain, Junao Wang, Ofir Weisse, R Sekar, Daniel Genkin, Boyuan He, Scott D Stoller, Gan Fang, Frank Piessens, Evan Downing, et al. 2018. Dependence-preserving data compaction for scalable forensic analysis. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 1723–1740.
[19] RedHat Inc. 2019. RedHat Linux Audit. https://people.redhat.com/sgrubb/audit/. [Online; accessed 25 May 2019].
[20] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. 2010. Intrusion Recovery Using Selective Re-execution. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 89–104. http://dl.acm.org/citation.cfm?id=1924943.1924950
[21] Samuel T King and Peter M Chen. 2003. Backtracking intrusions. *ACM SIGOPS Operating Systems Review* 37, 5 (2003), 223–236.
[22] Samuel T. King, Zhuoqing Morley Mao, Dominic G. Lucchetti, and Peter M. Chen. 2005. Enriching Intrusion Alerts Through Multi-Host Causality. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.
[23] Srinivas Krishnan, Kevin Z. Snow, and Fabian Monrose. [n. d.]. Trail of bytes: efficient support for forensic analysis. In *CCS'10*. ACM.
[24] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. [n. d.]. LDX: Causality Inference by Lightweight Dual Execution. In *ASPLOS'16*.
[25] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela F. Cretu-Ciocarlie, Ashish Gehani, and Vinod Yegneswaran. 2018. MCI : Modeling-based Causality Inference in Audit Logging for Attack Investigation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS) (NDSS '18)*.
[26] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition.. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.
[27] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. LogGC: garbage collecting audit log. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 1005–1016.
[28] LLC Lewes Technology Consulting. 2019. SOF-ELKÂő Virtual Machine Distribution. https://github.com/philhagen/sof-elk/blob/master/VM_README.md. [Online; accessed 25 May 2019].
[29] Shiqing Ma, Kyu Hyung Lee, Chung Hwan Kim, Junghwan Rhee, Xiangyu Zhang, and Dongyan Xu. 2015. Accurate, low cost and instrumentation-free security audit logging for windows. In *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 401–410.
[30] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. 2018. Kernel-Supported Cost-Effective Audit Logging for Causality Tracking. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 241–254. https://www.usenix.org/conference/atc18/presentation/ma-shiqing
[31] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2017. {MPI}: Multiple Perspective Attack Investigation with Semantic Aware Execution Partitioning. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 1111–1128.
[32] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting.. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.
[33] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 527–543.
[34] Kiran-Kumar Muniswamy-Reddy, David A Holland, Uri Braun, and Margo I Seltzer. 2006. Provenance-aware storage systems.. In *USENIX Annual Technical Conference, General Track*. 43–56.
[35] Inc Neo4j. 2019. Cypher. https://neo4j.com/docs/cypher-manual/current/. [Online; accessed 25 May 2019].
[36] Joshua OâĂŹMadadhain, Danyel Fisher, Padhraic Smyth, Scott White, and Yan-Biao Boey. 2005. Analysis and visualization of network data using JUNG. *Journal of Statistical Software* 10, 2 (2005), 1–35.
[37] Devin J Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. 2012. Hi-Fi: collecting high-fidelity whole-system provenance. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 259–268.
[38] Inc sysdig. 2019. Sysdig. https://sysdig.com/. [Online; accessed 25 May 2019].
[39] Yutao Tang, Ding Li, Zhichun Li, Mu Zhang, Kangkook Jee, Xusheng Xiao, Zhenyu Wu, Junghwan Rhee, Fengyuan Xu, and Qun Li. 2018. NodeMerge: Template Based Efficient Data Reduction For Big-Data Causality Analysis. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1324–1337.
[40] Matthias Vallentin, Vern Paxson, and Robin Sommer. 2016. {VAST}: A Unified Platform for Interactive Network Forensics. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 345–362.
[41] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. 2016. High fidelity data reduction for big data security dependency analyses. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 504–516.