FAuST: Striking a Bargain between Forensic Auditing's Security and Throughput

Muhammad Adil Inam
University of Illinois at Urbana-Champaign
mainam2@illinois.edu

Akul Goyal
University of Illinois at Urbana-Champaign
akulg2@illinois.edu

Jason Liu
University of Illinois at Urbana-Champaign
jdliu2@illinois.edu

Jaron Mink
University of Illinois at Urbana-Champaign
jaronmm2@illinois.edu

Noor Michael
University of Illinois at Urbana-Champaign
nsm2@illinois.edu

Sneha Gaur
University of Illinois at Urbana-Champaign
sg2@illinois.edu

Adam Bates
University of Illinois at Urbana-Champaign
batesa@illinois.edu

Wajih Ul Hassan
University of Virginia
hur7wv@virginia.edu

ABSTRACT

System logs are invaluable to forensic audits, but grow so large that in practice fine-grained logs are quickly discarded – if captured at all – preventing the real-world use of the provenance-based investigation techniques that have gained popularity in the literature. Encouragingly, forensically-informed methods for reducing the size of system logs are a subject of frequent study. Unfortunately, many of these techniques are designed for offline reduction in a central server, meaning that the up-front cost of log capture, storage, and transmission must still be paid at the endpoints. Moreover, to date these techniques exist as isolated (and, often, closed-source) implementations; there does not exist a comprehensive framework through which the combined benefits of multiple log reduction techniques can be enjoyed.

In this work, we present FAuST, an audit daemon for performing streaming audit log reduction at system endpoints. After registering with a log source (e.g., via Linux Audit's `auditsp` utility), FAuST incrementally builds an in-memory provenance graph of recent system activity. During graph construction, log reduction techniques that can be applied to local subgraphs are invoked immediately using event callback handlers, while techniques meant for application on the global graph are invoked in periodic epochs. We evaluate FAuST, loaded with eight different log reduction modules from the literature, against the DARPA Transparent Computing datasets. Our experiments demonstrate the efficient performance of FAuST and identify certain subsets of reduction techniques that are synergistic with one another. Thus, FAuST dramatically simplifies the evaluation and deployment of log reduction techniques.

ACSAC '22, December 5–9, 2022, Austin, TX, USA

ACM ISBN 978-1-4503-9759-9/22/12…$15.00
https://doi.org/10.1145/3564625.3567990

CCS CONCEPTS

• **Security and privacy → Intrusion/anomaly detection and malware mitigation**.

KEYWORDS

Auditing, Log Reduction

ACM Reference Format:
Muhammad Adil Inam, Akul Goyal, Jason Liu, Jaron Mink, Noor Michael, Sneha Gaur, Adam Bates, and Wajih Ul Hassan. 2022. FAuST: Striking a Bargain between Forensic Auditing's Security and Throughput. In *Annual Computer Security Applications Conference (ACSAC '22), December 5–9, 2022, Austin, TX, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3564625.3567990

1 INTRODUCTION

System logs form the basis for most reactive security measures in modern-day systems, describing fine-grained manipulations of kernel objects such as process creation, file access, and registry updates. These events are analyzed by Endpoint Detection and Response (EDR) systems [11] to alert analysts to potential intrusions, e.g., when system events match against the MITRE ATT&CK knowledge-based of adversarial tactics, techniques, and procedures [45]. Such logs are also essential to novel provenance-based auditing techniques (e.g., [2, 18, 25, 34, 36, 46]), where events are processed into causal dependency graphs so that the relationships between system entities can be easily identified. Even with alerting systems, log-based investigations are an essential step to system defense due to the high rate of false alarms observed in today's security products [6, 10].

Unfortunately, system logs can quickly grow to unwieldy sizes – while volume varies depending on machine load, logs generation rates have been reported to be anywhere between 1 GB [30] to 33

GB [34] per day per machine. These costs quickly add up, to the point that many large organizations are simply unwilling to pay for long-term retention of system logs. Products will often store logs in a ring buffer that is allocated to provide just a few months [38], or even days [53, 54], of storage. Worse, some products may not store recent logs at all unless explicitly and swiftly asked to do so by an analyst in response to a particular incident. These retention periods are simply insufficient when considering the "dwell times" observed by attackers in high-profile data breaches (e.g., [4, 26, 27, 43, 44, 49, 51]), contributing to the difficulty of effectively detecting and responding to threats.

In recognition of this difficulty, a rich literature has emerged that explores methods of reducing log data (e.g., [15, 21, 30, 39, 55, 59]). These techniques are often forensically-informed, translating insights about how logs are used in investigations into algorithms that selectively remove log events with little forensic value. For example, Lee et al.'s pioneering LogGC leverages the insight that log events describing temporary file I/O can be safely removed if those events do not impact the present state of the system [30]. Alternately, Xu et al.'s notion of *Causality-Preserving Reduction* (CPR) observes many log events describing I/O are redundant from an information flow perspective and can thus be removed [59]. While, trivially, retaining an unreduced log is better for organizational security than a reduced log, researchers have nonetheless demonstrated that reduced logs can often satisfy the needs of analysts during threat investigations.

Given the pressing need for highly-efficient log retention methods, why limit ourselves to one log reduction technique? Unfortunately, a variety of obstacles prevent such an approach. First, as a practical consideration, many log reduction techniques described in the literature or either proprietary or otherwise closed-source. Second, even if code for all of these techniques was available, they were designed for different log formats and under different deployment models; for example, Hossain et al.'s DPR system is designed as an offline mechanism that operates on a "completed" log [22], while CPR performs a local calculation and thus is suitable for an online setting [60]. Finally, and perhaps most importantly, there has been only limited exploration of whether these techniques can be "pushed down" to endpoints, focusing instead on a centralized log server model. Techniques applied at a central server may reduce the costs of long-term storage, but do not avoid the costs of local storage or network transmission. Further, many enterprises use third party cloud-based security products that ingest logs directly from endpoints. Since these security products charge by the GB (e.g., Splunk [52]), delayed log reduction not only increases the network transmission cost, but also incurs sizable financial costs due to expensive security licenses.

In this work, we present FAuST, a logging daemon that can perform *transparent and modular* audit log reduction directly at the *endpoints* where voluminous log events are generated. FAuST is comprised of modular parsers that interpret different audit log formats to create a uniform in-memory provenance graph representation. From there, modular filters can be applied to the graph to make reduction decisions that are then attributed back to the audit log. Locally-applicable log reduction algorithms are directly invoked on the graph as it is constructed in an event-driven fashion while globally-applicable log reduction algorithms are converted to a streaming setting by being invoked periodically in an epochal fashion. Our FAuST implementation can be registered with Linux Audit using the `audisp` utility. It is also compatible with the DARPA Transparent Computing's Common Data Model and can be easily extended to support alternate logging frameworks.

Using FAuST, we re-implement 8 exemplar log reduction techniques from the literature, then evaluate their performance against DARPA Transparent Computing's TRACE, THEIA, and CADETS systems using both Engagements 3 and 5 (i.e., six total datasets). We find that reduction performance can vary considerably based on the workloads of the target machine. We also observe a ceiling effect when attempting to apply these techniques in tandem; the most aggressive technique (S-DPR [22]) reduces log size by 87.3%, as compared to 90.7% when all techniques are applied. Moreover, our evaluation reveals that techniques that might not offer significant individual reduction (e.g., [14, 31, 56]) can still provide value-addition when applied with other techniques due to the unique and non-overlapping data patterns exploited by them. Through open-sourcing FAuST and associated log reduction implementations, we hope to make log reduction more accessible and also simplify the process of evaluating future techniques.

The main contributions of this paper are as follows:

- We propose FAuST, an audit daemon that transparently combines various log reduction techniques and applies them to streaming audit logs at the endpoints. To the best of our knowledge, we are the first ones to identify a unified set of requirements for 8 disparate log reduction algorithms and create a novel platform for these techniques to be transparently applied.
- We present a modular design for FAuST that enables users to seamlessly add and remove different reduction techniques.
- We evaluate the performance, utility, and effectiveness of FAuST using DARPA TC datasets. Additionally, we conduct experiments to uncover the reduction variance across different datasets and workloads, similarity and overlap across techniques and the benefits of using multiple techniques together, etc.

2 LOG REDUCTION TECHNIQUES

Prior work [14, 22, 31, 40, 56, 60] has attempted to develop techniques that reduce the audit log size without sacrificing the forensically-relevant evidence present in the audit logs. These log reduction techniques leverage data provenance to identify and remove redundant events from the audit logs. Data provenance describes the totality of system execution by parsing audit logs into dependency graphs called provenance graphs. In this graph, the vertices represent system entities, such as processes and files, and the edges represent causal relationships between system entities. Given an attack symptom (e.g., malicious file), the security analyst can query the provenance graph to identify an attack's root cause (backward tracing query) and ramifications (forward tracing query).

Below, we describe eight examplar log reduction techniques, which are also featured in our study.

LogGC. Lee et al. [31] proposed LogGC in 2013 and since then this technique has been adopted by various auditing systems [29, 35, 37]. The key idea behind LogGC is to identify and "garbage collect" system entities from the provenance graph that do not have persistent effects on the system. Consider an application that

writes to a temporary file and later deletes that temporary file. In this case, this file event will introduce any causal dependencies in the provenance graph; therefore, it is safe to remove this event from the audit logs. LogGC's reduction technique requires a complete provenance graph; however, we found that their technique can be applied to local subgraphs without affecting the accuracy of the technique.

NodeMerge. Tang et al. [56] proposed a template-based technique for audit log reduction called NodeMerge. NodeMerge is based on the observation that certain applications, during each invocation, read a specific set of read-only files for actions such as loading libraries, reading program configuration, and accessing read-only resources. These sets of frequent events are the same across all invocations and can be replaced with a single template. The technique consists of two phases: 1) the training phase and 2) the reduction phase. In the training phase, the system uses existing audit logs to learn the templates of frequent reads of read-only files for each process using FP trees. In the reduction phase, the system applies these templates on new unseen audit logs to replace a set of frequent reads with a single template read. The training phase of NodeMerge requires the global provenance state, whereas the reduction phase can be applied to local subgraphs.

CPR. The goal of the Causality Preserving Reduction (CPR) technique [60] is to remove redundant information from the audit logs as long as the complete information flow and causality are preserved in the underlying provenance graph. Rather than naïvely eliminate every repeated system call between a source and destination entity, CPR tests for *interleaved flows*, i.e., whether any new inputs have been received at the source between the two system calls. An interleaved flow indicates that the system call may not actually be redundant and thus should be preserved. In other words, if a process performs multiple writes to a file without reading any other object between them, it just preserves the first write event and removes all the subsequent write events from the audit logs. This is because the process state does not change between subsequent writes, and hence the new writes do not add any new information flow.

PCAR. Process-centric Causality Approximation Reduction (PCAR) technique [60] extends the idea of CPR by aggressively eliminating redundant events when a "bursty" process exceeds some number of system calls per second. This technique is based on the observation that certain applications, such as system daemons produce a burst of semantically similar events. These bursts of events indicate a single high-level application event and contain many interleaved dependencies. For instance, a process scanning for PCI devices may repeatedly interleave read and write accesses to a specific set of files in a burst. PCAR first identifies these bursts by identifying the processes that interact with a large number of objects in a short time. After that, PCAR identifies the neighbor sets and the information inflows or outflows for these processes to achieve data reduction without impacting the forensic analysis.

F-DPR and S-DPR. Dependency Preserving Reduction (DPR) technique was proposed by Hossain et al. [22] in 2018. The idea behind DPR is that a reduced provenance graph is sufficient as long as it can identify the same entities as the full provenance graph when performing backward tracing and forward tracing. To achieve this, the DPR systems selectively drop flow events that are not necessary to correctly traverse every entity's ancestors (S-DPR), or ancestors and successors (F-DPR). For instance, if two paths in the provenance graph lead back to the same root-cause, S-DPR only preserves one of the paths as long as the other path is not preserving the dependencies for another event.

Winnower. Hassan et al. [14] proposed Winnower to reduce the audit log size by summarizing the provenance graphs of hundreds or thousands of replicated cloud applications. Winnower leverages graph grammar, a rule-based mechanism for analyzing and generating a graph, to model behaviors of several endpoints in a cluster. Using these models, Winnower identifies if the audit logs from different endpoints in the cloud are operationally equivalent (i.e., highly redundant) and therefore can be removed from the audit logs before transmitting them to the central storage server. Winnower's reduction technique is locally-applicable to the streaming audit logs.

LogApprox. Michael et al. [40] proposed the LogApprox technique to efficiently summarize repeated executions of the same program on a single host. It performs bounded regular expression learning over file I/O events, approximating typical behaviors while atypical behaviors are losslessly retained. LogApprox consists of two phases - a regex learning phase and a reduction phase. In the regex learning phase, the technique creates groups of filenames based on path similarities and generates regexes corresponding to each group. The regexes are crafted such that they avoid overgeneralization and do not describe unique attack patterns. In the reduction phase, the regexes are applied to past, and new audit logs, such that each group corresponds to a single approximated regex entry. Since the information flow across a regex group remains the same, approximated causality is preserved. The regex learning phase of LogApprox requires the global provenance state, whereas the reduction phase can be applied locally or globally.

3 SYSTEM DESIGN

3.1 Threat Model & Assumptions

The attack surface we consider in this work is that of a typical organization with several connected endpoints. After initial intrusion, the attacks may attempt to perform privilege escalation, lateral movement, and data exfiltration to achieve their goals. Like other work in this space (e.g., [17, 29, 35, 37, 42, 58]), we assume that the audit daemon running on the endpoint is not compromised. While the adversaries can compromise user-level programs, we assume that the system audit data is still provided and protected by the kernel, i.e., events generated by user-level programs are faithfully recorded and reported. We also note that maintaining the integrity of audit logs [24, 47] is an important problem; however, we consider this problem out of the scope of this work.

3.2 Design Goals

Given the challenges discussed in Section 1, we set out to design a system with the following design goals:

- **Online Log Reduction** Our system must combine all the relevant audit log reduction techniques and apply those techniques to the streaming audit logs generated at the endpoints.

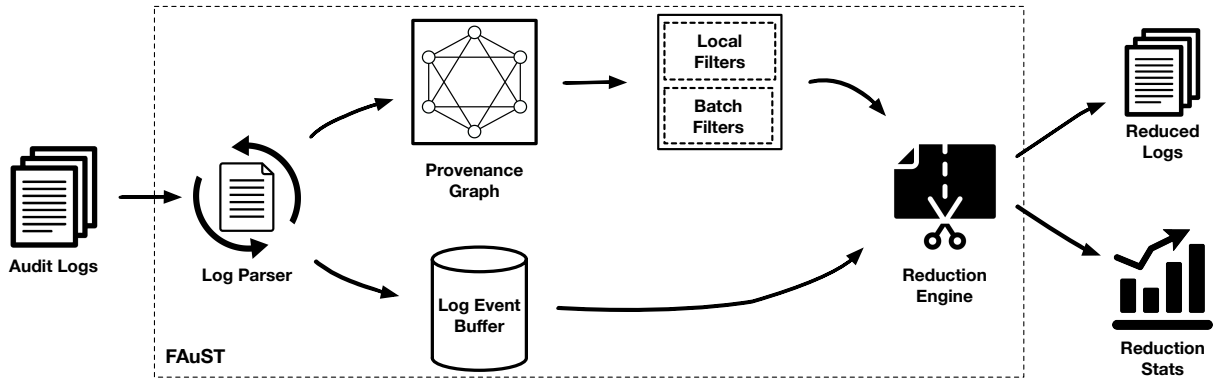

Figure 1: FAuST architecture overview. First, the streaming audits logs are parsed by our log parsers to generate provenance graphs. After that, our local filters apply locally-applicable reduction techniques in an event-driven fashion, and our batch filters apply globally-applicable reduction algorithms in an epochal fashion on the generated provenance graphs.

- **Minimal Log Transmission** Our system should prevent endpoints from sending redundant audit logs event to central storage, i.e., only transmits logs after applying log reduction techniques.
- **Generality** Our system design should be independent of the underlying operating system and applications used by the endpoints.
- **Interoperable and Extensible.** Our system must support a variety of audit logging frameworks and should be able to easily accommodate the incorporation of new log reduction techniques.

3.3 Overview

To achieve these goals, we present FAuST, a logging daemon that can perform transparent and modular audit log reduction directly at the endpoints where voluminous log events are generated. An overview of FAuST is shown in Figure 1. In the first step, FAuST is registered with a log source (e.g., via Linux Audit's `audisp` utility). The raw audit logs are then fed to a log parser module that interprets different log formats and incrementally creates two different in-memory representations of log data – the log event buffer and the corresponding provenance graph. The log event buffer stores a uniform in-memory representation of each system-level entry present in the audit logs. For each entry, it stores all necessary fields required for analysis along with the complete log event in the original format. The in-memory provenance graph representation has nodes corresponding to system entities, whereas the edges in the provenance graph correspond to individual log entries. From there, modular filters are applied to the provenance graph to make reduction decisions.

Locally-applicable log reduction algorithms are directly invoked on the graph as it is constructed using event callback handlers. On the other hand, globally-applicable log reduction algorithms are converted to a streaming setting by being invoked periodically in an epochal fashion. Lastly, the reduction engine is invoked at the end of each epoch. It takes as input both the filter decisions and the log event buffer containing the original log entries. From there, it partially writes out the log event buffer to the disk to create the reduced log and also tabulates statistics about reduction during each epoch.

In the security operations center (SOC) pipeline, FAuST is deployable as a transparent proxy to endpoint telemetry sensors. It invisibly feeds logs into existing tools. Once registered with a log source (e.g via Linux Audit's audisp utility), the logs generated by the system's audit daemon (e.g, Linux auditd) are directly fed to FAuST for reduction. FAuST can then output the reduced logs for downstream SOC tasks, including storing logs to disk or directly transmitting to log servers (e.g., syslog).

3.4 Log Parsing

FAuST is designed to support multiple audit log formats across different frameworks. FAuST can be registered with Linux Audit using the `audisp` utility and is also compatible with the DARPA Transparent Computing's Common Data Model. Due to the extensible nature of FAuST, it can be easily extended to support alternate logging frameworks. The raw audit logs from the source are fed to the log parser module that parses the audit logs and constructs two uniform in-memory representations of log data.

First, the module enqueues each log entry within the log stream in an event buffer. The buffer contains a unique event ID, all necessary event description fields required for reduction analysis along with raw log entry associated with each log event. In certain cases, the event description for a log entry may be dependent on multiple other entries. For instance, while the log entries for `read` and `write` system calls specify the file descriptors, the proper resolution of these events require the `inode` information specified in the log entry of the corresponding `open` system call. To handle such event descriptions, the parsing module maintains all possible mappings between different log entries.

Second, the parser is responsible for incrementally constructing an in-memory representation of the provenance graph from individual event descriptions. In the provenance graph representation,

Handler	Source	Destination	Edge Relation	Description
execve/fork	Child process	Parent process	Forked_by	An edge is created between the child and parent process.
open	File	Process	Opened_by	An edge is created and the inode is added to the process-fd map.
close	File	Process	Closed_by	An edge is created and the inode is removed from the process-fd map.
read	Process	File	Used	A read edge is created by extracting the inode from the process-fd map.
write	File	Process	Generated_by	A write edge is created by extracting the inode from the process-fd map.
unlink	File	Process	Unlinked_by	An unlink edge is created and the file node within the graph is marked as dead
exit	Child process	Parent process	Exited_by	An unlink edge is created and the process node within the graph is marked as dead
connect	Socket	Process	Opened_by	An edge is created and the socket address (saddrr) is added to the process-fd map
recv	Process	Socket	Used	A recv edge is created by extracting the saddr from the process-fd map.
send	Socket	Process	Generated_by	A send edge is created by extracting the saddr from the process-fd map.

Table 1: Implementation details of various event handlers within FAuST.

nodes correspond to processes, files, and other file-like objects (e.g., VFS and network sockets), whereas the edges correspond to the interactions (events) between these nodes. For each event in the log event buffer, the parser identifies the subject and object entities along with the interaction between them. For example, in the case of file-IO events, the process issuing the system-call acts as the subject, the target file acts as the object, and the interaction between them is determined by the type of operation on the file (open, read, write, close, etc.). For any of the previously unencountered subject and object entities, the parser adds a new node in the provenance graph. Similarly, the interaction between the subject and object (e.g, OPENED_BY, CLOSED_BY, FORKED_BY, etc.) is represented by adding a new edge between the subject and the object nodes. Each edge in the provenance graph representation also contains an event ID that points back to associated log entries in the event buffer.

3.5 Log Filtering

The core filtering logic of FAuST is handled by two modules: event-driven filters and batch filters. The event-driven filters are applied locally as the provenance graph is being constructed using event callback handlers. These filters do not require the global graph state and operate on the local subgraphs constructed thus far. On the other hand, the batch filters are applied to the global provenance graph for the full batch of logs at the end of each epoch. For each edge (event) in the provenance graph, FAuST maintains a mapping from event ID to active reduction filters. The mapping results are passed on to the reduction engine at the end of each epoch.

Because our tool is based on an in-memory database, it was necessary to manage memory overheads by partitioning the logs into batches. Since the reduction techniques are applied per batch at the end of each epoch, the redundant events between epochs are not identified in FAuST. This makes the observed log reduction rates in our evaluation an underestimation of the optimal reduction rates. Fortunately, in experimenting with different epoch sizes, we observed that the differences in log reduction rates were negligible, and that a 100k batch size was sufficient to achieve similar log reduction rates to those reported in prior work. We further discuss this in Section 5.

3.6 Reduction Engine

The reduction engine is the final module in FAuST's pipeline. It is responsible for outputting the reduced logs along with the reduction statistics. At the end of each epoch, the reduction engine examines the filter decisions for both event-driven and batch-based filters. It then cross-references the filter decisions with the log event buffer to output the reduced audit logs. The reduction engine can be configured to reduce logs using any combination of the eight reduction techniques. This module is also responsible for mangling or modifying any log entries before they are written to disk. For instance, in the case of LogApprox [40], the file paths in certain log entries are replaced with appropriate regexes. Similarly, for NodeMerge [56], the file paths are replaced with corresponding node templates, and the templates are also stored on disk. In addition to the reduced logs, the reduction engine also outputs various reduction statistics, including filter decisions per event, total size reduction, throughput and memory footprint of different techniques, etc. We report these statistics for the DARPA Transparent Computing datasets in Section 5.

4 IMPLEMENTATION

FAuST is implemented in C++ using 7500 LoC (calculated with cloc [8]). To foster future research, we have open-sourced our implementation[1]. For online streaming reduction, the log parsing module is registered with the Linux Audit subsystem using the `audisp` utility. We modified the `audisp` configuration such that the logs generated by `auditd` are directly fed to FAuST without being written to the disk. Additionally, FAuST can be configured to work offline and read the log files from local storage to reduce and evaluate existing datasets. Our offline implementation currently supports the DARPA TC data formats for both Engagement 3 and Engagement 5. Note that users can extend FAuST to work with other log formats by registering an event callback function that can parse the respective data format. To generate in-memory provenance graphs, FAuST leverages the SNAP graph library [32]. The implementation details of various event handlers within FAuST for provenance graph generation are outlined in Table 1.

[1] https://bitbucket.org/sts-lab/faust

Techniques	Filter Type	Training Required?	Event Mangling?
LogGC [31]	Local	X	X
NodeMerge [56]	Batch	✓	X
CPR [60]	Local	X	X
F-DPR [22]	Batch	X	X
S-DPR [22]	Batch	X	X
PCAR [60]	Local	X	✓
LogApprox [40]	Batch	✓	✓
Winnower [14]	Batch	✓	✓

Table 2: Categorization of different filters implemented within FAuST.

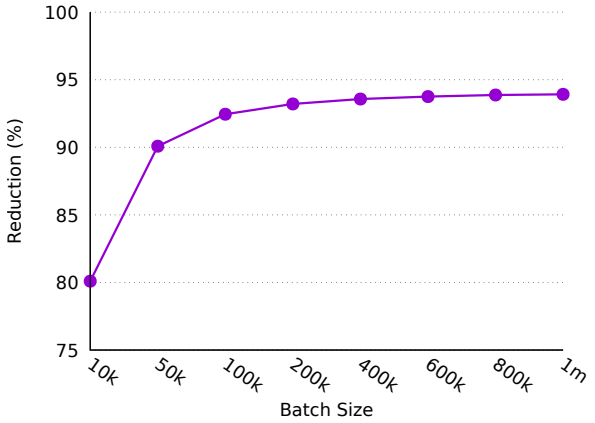

Figure 2: Observed reduction rates for the E3-Theia dataset across different batch sizes for all techniques combined.

4.1 Provenance Graph Filters

We reimplemented the eight reduction techniques discussed in Section 2, based on their descriptions in the original papers. We implement the techniques as faithfully and staunchly as possible. Note that because we did not have access to the authors' source code, our implementation of these techniques might slightly deviate from the original systems. After comprehensively reviewing each technique and its implementation, we are confident that our implementations align with the methodologies and the designs discussed in the original papers. A summary of implementation categorization is shown in Table 2. Three out of the eight techniques are implemented as local filters, whereas the remaining five are implemented as batch filters. The implementation details of these filters and the corresponding algorithms can be found in Appendix A.

5 EVALUATION

We now evaluate the performance of FAuST along with its constituent audit log reduction techniques. To do so, we must first select a corpora of audit log datasets that describe security incidents. Ideally, we would be able to benchmark FAuST against audit logs captured in a enterprise the describe real-world intrusions;

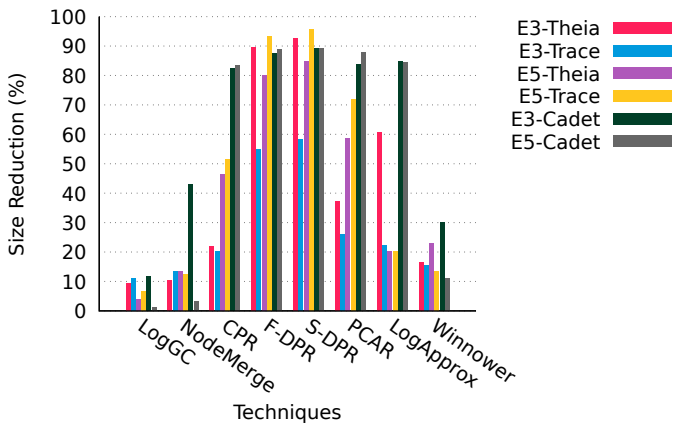

Figure 3: Observed reduction percentages for each technique separately using DARPA datasets.

unfortunately, due to the sensitive nature of logs, such datasets are not publicly available. Instead, we leverage the datasets released by DARPA Transparent Computing (TC) Program, Engagements 3 and 5 [1] conducted in 2018 and 2019, respectively. The DARPA TC engagements contains event streams from multiple hosts and documented ground truth information of the attacks conducted on the machines. We use the four Linux-based datasets from hosts running the Theia and Trace systems. Additionally, we use two FreeBSD-based datasets from hosts running the Cadet system. We refer to these six datasets as E3-Theia, E3-Trace, E5-Theia, E5-Trace, E3-Cadet, and E5-Cadet. In contrast to "real" enterprise logs, these datasets suffer from some notable limitations: first, each dataset describes one week of system activity; second, benign background activities were generated by workload generators and may not be sufficiently representative of real-world behaviors; third, the system loads were designed to reflect typical workstation behavior, not the behaviors of fully saturated servers. Because of this, there is a small risk that our results may not be representative of longer-term use of FAuST, reflect performance for diverse human interactions with real-world systems, or capture the challenges of auditing highly-utilized server machines. This said, the DARPA TC engagements were the most robust and expansive publicly-available datasets at the time of this study. Further, they have been widely used for benchmarking related prior work (e.g., [13, 19, 20, 41]), providing an opportunity to contrast FAuST's performance with other systems.

5.1 Batch Size

As mentioned before, FAuST processes log events in batches to adapt reduction techniques to a streaming setting and reduce memory consumption. In our evaluation, we experimented with different batch sizes (up to 1 million events per batch). The reduction rates for different batch sizes across all techniques combined for the E3-Theia dataset are shown in Figure 2. We observe diminishing returns for batch sizes greater than 100,000 events, i.e., increasing the batch size from 100,000 to 1 million yields less than 1.5% change

Technique 1	Technique 2	Technique 3	Size Red.
S-DPR	PCAR	-	87.57% (8.0X)
S-DPR	Winnower	-	87.81% (8.2X)
S-DPR	LogApprox	-	88.32% (8.6X)
S-DPR	LogGC	-	88.53% (8.7X)
S-DPR	NodeMerge	-	89.17% (9.2X)
S-DPR	NodeMerge	PCAR	89.19% (9.2X)
S-DPR	NodeMerge	LogApprox	89.22% (9.3X)
S-DPR	LogGC	LogApprox	89.48% (9.5X)
S-DPR	NodeMerge	Winnower	89.63% (9.6X)
S-DPR	NodeMerge	LogGC	90.24% (10.3X)
All Techniques Combined			90.70% (10.8X)

Table 3: Top 5 2-sized and 3-sized subsets of techniques (with and without DPR) for all Linux-based datasets combined and their corresponding reduction rates i.e both log reduction percentage (1 - Reduced Log / Raw Log) and in parenthesis the log reduction factor (Raw Log / Reduced Log).

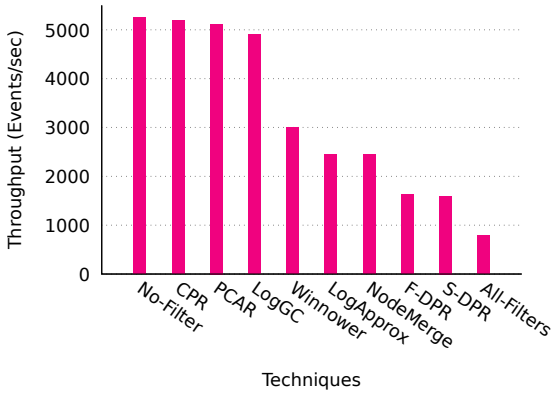

Figure 4: Throughput (events ingested per second) for different filters in FAuST at 100% CPU utilization.

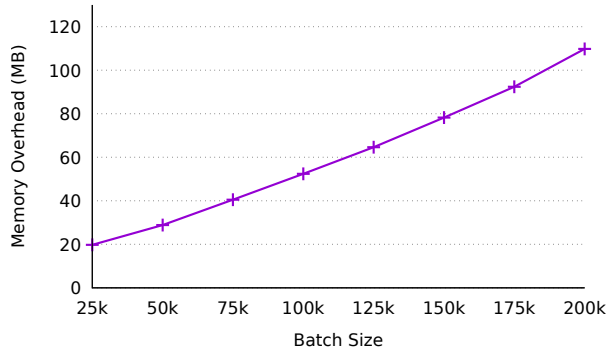

Figure 5: Memory overhead for different batch sizes in FAuST.

in reduction across all techniques combined. Therefore, in the rest of our experiments, we use a batch size of 100,000.

5.2 Storage Reduction

The rate of log reduction across various techniques using DARPA TC datasets is shown in Figure 3.[2] The space efficiency results indicate that the reduction capabilities are significantly impacted by the dependencies and causalities retained across different techniques. Most techniques' performance was roughly consistent with the reported performance from their original papers, with DPR boasting the most substantial reduction rates. In comparison, techniques that do not compromise on the information flow in the provenance graphs and aim to retain either complete causality (CPR) or approximated causality (e.g., PCAR, LogApprox, and Winnower) achieve lesser size reduction. While DPR techniques outperform all other techniques, their reduction typically comes at a cost. Micheal et al. [40] showed that DPR techniques might retain as little as 8% of forensically relevant evidence for certain attack cases.

In our experiments, LogGC, NodeMerge, and Winnower appear to underperform across most datasets compared to original reports. LogGC and NodeMerge both leverage application behaviors, and their reduction capabilities are primarily impacted by the environment and workload of the machines, as suggested by the authors of the respective papers. Additionally, in the case of LogGC, we observe that the DARPA TC datasets chose not to log many of the termination events (e.g., EXIT and CLOSE), preventing the LogGC algorithm from activating. When we tested LogGC against in-lab datasets, we found its performance more consistent with its original evaluation. As far as Winnower is concerned, this technique is proposed by Hassan et al. [14] for reducing provenance graphs from identical containers and is better suited for isomorphic provenance graphs. Therefore, it might not work as effectively in the context of a single host. The range of reduction for the rest of the techniques matches the ones claimed by their respective authors.

[2]We also analyzed space efficiency in terms of events dropped but observed no significant difference in trends compared to size reduction.

We also observe that the performance of the different log reduction techniques varied significantly by dataset. In the most extreme case, DPR techniques achieve less than 60% reduction for the E3-Trace dataset in comparison to the ≈90% reduction across the other five datasets. E3-Trace contains a significantly large proportion of process events as compared to other datasets. Most reduction techniques target I/O behavior, which explains the lower overall reduction rates for E3-Trace. Similarly, CPR and PCAR exhibit significantly higher reduction rates in the FreeBSD-based datasets (E3 and E5 Cadet) than others due to differences in simulated workloads.

Since FAuST performs reduction at endpoints, the storage reduction inherently reduces the cost of log transmission over the network. For a network MTU of 1500 bytes, the size of TCP/IP headers (around 40 bytes) results in an ≈2% overhead on top of the payload. Therefore, a 90% reduction in log size reduces the network transmission cost by approximately 89.8%. Since many enterprises use third party cloud-based security products that charge by the GB (e.g., Splunk [52]), the endpoint reduction performed by significantly reduces the financial costs associated with log transmission and storage.

Data	% Reduction	TP	FP	TN	FN	Precision (%)	Recall (%)	% Attack Edges Retained
Unfiltered	0	27	24	48253	4	52.94	87.10	100
NodeMerge	10.4	27	24	48253	4	52.94	87.10	100
CPR	22.04	25	29	48248	6	46.30	80.65	98
S-DPR	92.45	4	78	5104	24	4.88	14.29	13.5
CPR + NodeMerge	30.14	25	33	48244	6	43.10	80.65	98
S-DPR + NodeMerge	93.74	4	78	5104	24	4.88	14.29	13.5

Table 4: Detection results of the DeepLog anomaly detection system for NodeMerge, CPR and S-DPR reduced datasets. The first row represents the baseline numbers for the unfiltered dataset.

5.3 Collective Usage of Multiple Techniques

We also analyze the reduction statistics reported by FAuST to investigate the benefits of using multiple techniques together. Table 3 shows the size reduction when two, three, or all techniques are applied to the Linux-based datasets (Theia and Trace). We report size reduction in terms of both percentage of original log size as well as the "log reduction factor," which can be interpreted as the number of times the reduced log can fit into the storage footprint of the original log. We observe that the techniques that do not offer significant individual reduction (e.g., Winnower, NodeMerge, and LogGC) can still provide value-addition due to the non-overlapping nature of data patterns exploited by them. The total reduction increases from 88-89% with two techniques and to 89-90% with three techniques combined. However, there are diminishing returns as more techniques are applied. With all techniques combined, we achieve around 90.7% reduction. This suggests that to strike a better balance between reduction performance and overhead, FAuST should realistically be configured with 2 to 3 active reduction filters.

5.4 Throughput Benchmark

An important performance consideration for FAuST is to make sure that our tool can keep pace with the speed of audit log generation at the endpoint. To evaluate this, we first measure the event generation rate of the E3-Theia dataset that remains under 500 events per second for each batch throughout the dataset. We then measure our tool's event throughput rate, i.e., events processed by FAuST per second. Figure 4 compares the event throughput rate of FAuST for the E3-Theia dataset under three different configurations: 1) when no filters are applied, 2) when filters are applied individually, and 3) when all filters are applied together. We observed that locally-applicable filters(e.g., LogGC, CPR, and PCAR) applied to provenance graphs only minimally impacted the event throughput rate. On the other hand, filters applied periodically at the end of the batch significantly impact the event throughput rate since they perform complex graph operations. *Note that even with all the filters applied, the average throughput rate of FAuST (≈790 events/sec) far outpaces the number of audit logs generated during the DARPA TC engagement (<500 events/sec).*

5.5 Memory Overhead

FAuST makes use of an in-memory database to store uniform representations of the log event buffer and the provenance graph. To manage memory overheads, FAuST partitions the logs into batches and clears the corresponding in-memory database once the batch is processed. The memory overhead associated with FAuST for different batch sizes is shown in Figure 5. As expected, the memory consumption by FAuST directly correlates with the size of the batch. At an optimal batch size of 100k (as discussed in Section 5.1), the memory consumption by FAuST remains well under 60 MB, which is easily manageable on most commodity systems. This further illustrates the practicality of our tool to operate on commodity systems with realistic workloads.

5.6 Case Study

To understand the effectiveness of FAuST during threat investigations when multiple filters are applied, we present a case study from DARPA TC engagement 3. Figure 6 first depicts the original provenance graph generated from system-level audit logs during the Firefox backdoor attack. The green subgraph represents a simplified version of benign system activity, whereas the red subgraph is a simplified version of the provenance graph that describes the attack. A victim machine, unknowingly running a vulnerable Firefox 54.0.1, establishes a connection with a malicious ad server located at 146.153.68.151. The server exploits a backdoor in Firefox and injects a binary executable "Drakon" into its process memory. Drakon subsequently spawns a new process (/home/admin/clean) with root privileges that connects to the attacker's server at 161.116.88.72, giving the attacker full access to the victim machine. We configure FAuST with three active filters (S-DPR, CPR and NodeMerge) and replay the audit logs generated during the attack engagement.

The original provenance graph consists of multiple repeated low-level events between OS objects, such as various read events between a process and a file, and multiple send/receive events between a process and a socket. Since CPR merges system call events between a source and a destination entity as long as there are no interleaved flows, some of these repeated low-level events (send, recv, read, etc.) are removed by the CPR filter as shown in the second graph of Figure 6. Since S-DPR removes all events not necessary to correctly traverse every entity's ancestors, only single edges are preserved between the same source and destination as shown in the third graph of Figure 6. Moreover, the original provenance graph consists of specific sets of read-only files that are repeatedly read during each invocation of the firefox process. Since NodeMerge identifies and generates templates for the sets of frequent events that are the same across all program invocations, the repeated sets of read-only files in the original graph are replaced with individual templates, as shown in the second and third graph of Figure 6. Given

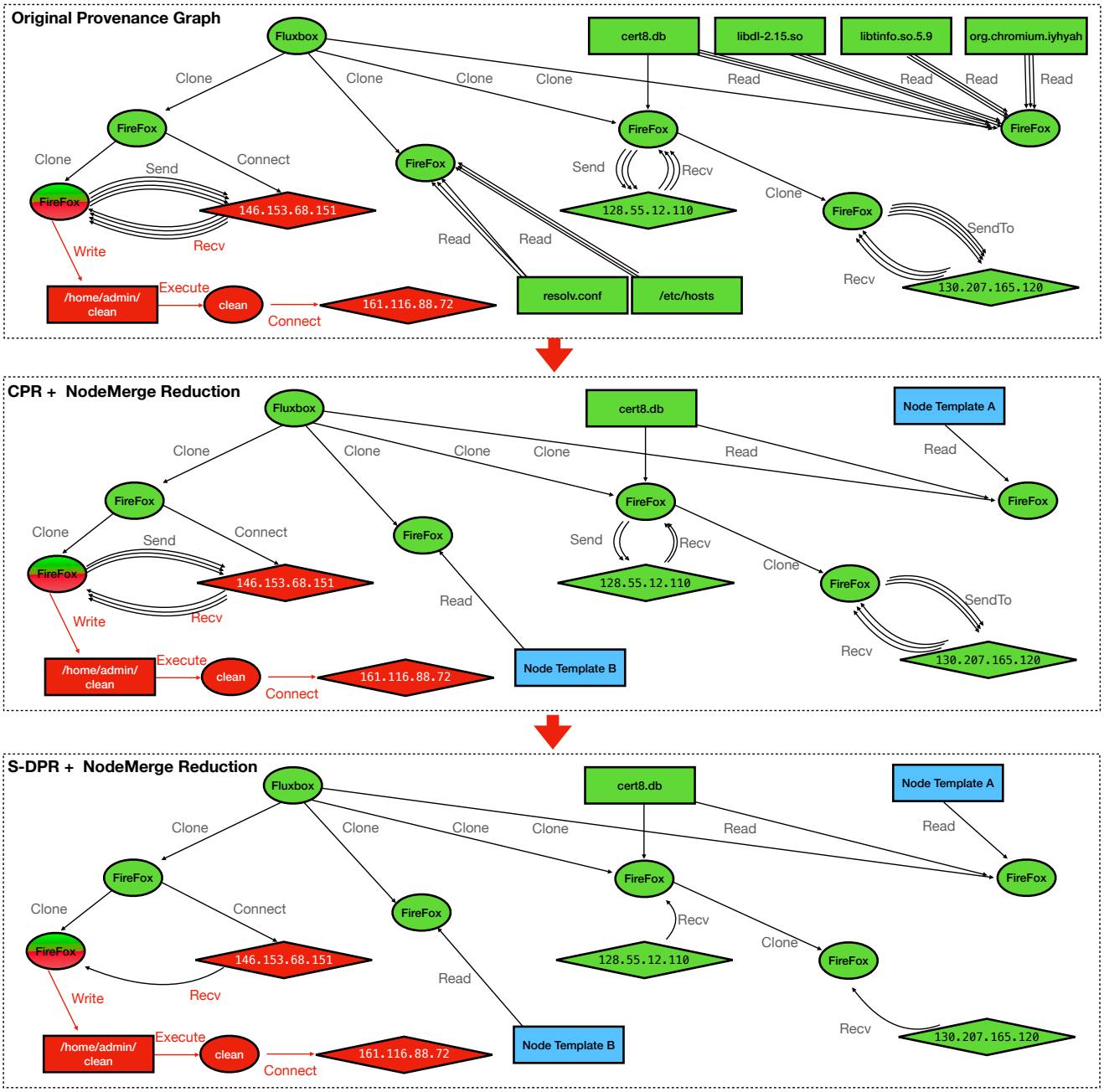

Figure 6: Provenance graph describing the Firefox backdoor APT attack from the DARPA TC datasets. The graph is first filtered by applying CPR and NodeMerge reduction techniques. The graph is then further reduced by applying the S-DPR and NodeMerge reduction technique. (node merge templates are shown as blue nodes)

a symptom of the attack, the reduced provenance graph presented to a security analyst. Both CPR/DPR and NodeMerge exploit different data access patterns for reducing log size, highlighting the utility of using multiple filters in FAuST during threat investigations.

In our case study, we additionally investigate if reduction techniques impact the forensic utility of the audit log. To do so, we analyze the reduced logs using an exemplar anomaly detection system i.e., DeepLog[3] [9]. To create comprehensive ground truth labels for the attack events in the dataset, we performed back traces on each of the attack steps described in the E3 documentation, then

[3]We utilized a 3rd party implementation of DeepLog for our experiments: https://github.com/nailo2c/deeplog

manually pruned the results to remove false dependencies. We then applied NodeMerge, CPR and S-DPR reduction techniques on the labeled dataset. Using the optimal configurations reported in the original papers, we then trained DeepLog using a 70/30 train/test split. DeepLog performed classification at the granularity of individual log sequence windows of 1000 ms of log data. We report TP, FP, TN, FN, recall and precision for this model.[4] The results are shown in Table 4. Across all systems, we observe an inverse relation between storage efficiency and anomaly detection performance. The low-reduction techniques are not associated with any change in detection performance, while the most aggressive technique (S-DPR) sees a dramatic drop in the model's ability to detect attack sequences and avoid false positives. Our anomaly detection experiment uncovers a trend in which aggressive log reduction leads to reduced intrusion detection performance. To further explore this issue, we recommend that future research on audit log reduction conduct security analyses not just through demonstrative anecdotal examples, but also through data-driven analysis of the impacts on security monitoring software.

6 RELATED WORK

6.1 Provenance Analysis

There has been a lot of work to leverage data provenance analysis for threat investigation [2, 12, 19, 23, 30, 41, 50, 57, 61]. Backtracker [25] introduced the concepts of backward tracing and forward tracing on kernel-level audit logs for intrusion investigations. Later several systems[28, 29, 35, 37] were introduced to solve the problem of dependency explosion in the provenance graphs and enable accurate root cause analysis using provenance graphs. PrioTracker [33] accelerates the forward tracing by prioritizing abnormal events. Zhou et al. [62] designed SNoOPy a provenance-based forensic system for distributed systems that can work under adversarial settings.

Chen et al. [7] introduced the concept of differential provenance to perform precise root-cause analysis by reasoning about differences between provenance trees. Elsewhere in the literature, a provenance graph visualization [5] technique is also proposed to facilitate data provenance navigation and exploration. As such, all these existing systems focus on improving the accuracy of provenance graph generation instead of reducing the size of audit logs to enhance storage efficiency.

6.2 Log Reduction

Besides the existing log reduction systems that we introduced in Section 2, there are several other log reduction systems proposed in the literature. Ma et al. [34] introduced KCAL to reduce the size of logs by caching event dependencies inside the kernel. KCAL discards all the redundant events indicated by cache hits and caches only the events that introduce new event dependencies. One advantage of performing in-kernel log reduction is that it eliminates the overhead of transferring redundant logs from kernel to userspace and storing them on the disk. However, due to KCAL's dependency on kernel instrumentation, we do not leverage KCAL in our work.

[4]Accuracy can be calculated from the provided data, but is highly misleading due to the large number of true negatives.

Rather than pruning based on a general application behavior, Bates et al.'s ProvWalls [3] examines an application's security policy to determine the subjects and objects on the system that form its Trusted Computing Base(TCB), and then prunes all events that fall outside the TCB. Since it is an application-specific approach, we do not leverage ProvWalls in our work.

CamQuery [48] translated log analysis to a streaming model on endpoints using a vertex-centric graph query API. While some log reduction techniques could likely have been expressed in CamQuery as programmable query functions, others could not because they rely on a global view of the graph that CamQuery did not support. Hassan et al. [16] also introduce an audit log reduction technique that only preserves audit log events that are necessary to generate dependencies between threat alerts and filters out the rest of the audit logs. However, their proposed technique requires the audit logs from all the endpoints to build a global provenance graph; therefore, it is not applicable to our system.

7 CONCLUSION

FAuST is the first decentralized audit log reduction framework that transparently combines various log reduction techniques to reduce audit log size at the endpoints. FAuST features two types of in-memory log filters: 1) *local filters* that apply locally-applicable log reduction techniques using event callback handlers and 2) *global filters* that apply globally-applicable log reduction techniques in epochal fashion. We have implemented FAuST for the Linux Audit subsystem and evaluated it on DAPRA TC datasets to show that FAuST is modular, highly efficient, and enables users to seamlessly load and combine different log reduction techniques.

REFERENCES

[1] 2020. DARPA Transparent Computing. 2020. Transparent Computing Engagement 3 Data Release. (2020).

[2] Adam Bates, Dave Tian, Kevin R.B. Butler, and Thomas Moyer. 2015. Trustworthy Whole-System Provenance for the Linux Kernel. In *Proceedings of 24th USENIX Security Symposium* (Washington, D.C.).

[3] Adam Bates, Dave Tian, Grant Hernandez, Thomas Moyer, Kevin R.B. Butler, and Trent Jaeger. 2017. Taming the Costs of Trustworthy Provenance through Policy Reduction. *ACM Trans. on Internet Technology* 17, 4 (sep 2017), 34:1–34:21.

[4] Tara Siegel Bernard, Tiffany Hsu, Nicole Perlroth, and Ron Lieber. 2019. Equifax Says Cyberattack May Have Affected 143 Million in the U.S. https://www.nytimes.com/2017/09/07/business/equifax-cyberattack.html.

[5] Steven P. Callahan, Juliana Freire, Emanuele Santos, Carlos E. Scheidegger, Cláudio T. Silva, and Huy T. Vo. 2006. VisTrails: Visualization Meets Data Management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (Chicago, IL, USA) *(SIGMOD '06)*. ACM, New York, NY, USA, 745–747. https://doi.org/10.1145/1142473.1142574

[6] Carbon Black. 2018. Global Incident Response Threat Report. https://www.carbonblack.com/global-incident-response-threat-report/november-2018/. Last accessed 04-20-2019.

[7] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2015. Differential Provenance: Better Network Diagnostics with Reference Events. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets'15)* (Philadelphia, PA).

[8] Albert Danial. 2021. *cloc: v1.92.* https://doi.org/10.5281/zenodo.5760077

[9] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *CCS*.

[10] FireEye, Inc. 2019. How Many Alerts is Too Many to Handle? https://www2.fireeye.com/StopTheNoise-IDC-Numbers-Game-Special-Report.html.

[11] Gartner Peer Insights. 2019. Endpoint Detection and Response Solutions Market. https://www.gartner.com/reviews/market/endpoint-detection-and-response-solutions.

[12] Ashish Gehani and Dawood Tariq. 2012. SPADE: Support for provenance auditing in distributed environments. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer.

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

[14] Wajih Ul Hassan, Lemay Aguse, Nuraini Aguse, Adam Bates, and Thomas Moyer. 2018. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *NDSS*.

[15] Wajih Ul Hassan, Nuraini Aguse, Mark Lemay, Thomas Moyer, and Adam Bates. 2018. Towards Scalable Cluster Auditing through Grammatical Inference over Provenance Graphs. In *Proceedings of the 25th ISOC Network and Distributed System Security Symposium (NDSS'18)*. San Diego, CA, USA.

[16] Wajih Ul Hassan, Adam Bates, and Daniel Marino. 2020. Tactical provenance analysis for endpoint detection and response systems. In *IEEE Symposium on Security and Privacy (SP)*.

[17] 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*.

[18] Md Nahid Hossain, Sadegh M. Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R. Sekar, Scott Stoller, and V.N. Venkatakrishnan. 2017. SLEUTH: Real-time Attack Scenario Reconstruction from COTS Audit Data. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 487–504. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/hossain

[19] Md Nahid Hossain, Sadegh M Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R Sekar, Scott Stoller, and VN Venkatakrishnan. 2017. {SLEUTH}: Real-time attack scenario reconstruction from {COTS} audit data. In *USENIX Security Symposium*.

[20] Md Nahid Hossain, Sanaz Sheikhi, and R Sekar. 2020. Combating dependence explosion in forensic analysis using alternative tag propagation semantics. In *IEEE Symposium on Security and Privacy (SP)*.

[21] Md Nahid Hossain, Junao Wang, R. Sekar, and Scott D. Stoller. 2018. Dependence-preserving Data Compaction for Scalable Forensic Analysis. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, Berkeley, CA, USA, 1723–1740. http://dl.acm.org/citation.cfm?id=3277203.3277331

[22] 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 *USENIX Security Symposium*.

[23] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. 2017. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *CCS*.

[24] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. 2017. SGX-Log: Securing System Logs With SGX. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS '17)*.

[25] Samuel T. King and Peter M. Chen. 2003. Backtracking Intrusions. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. ACM, New York, NY, USA, 223–236. https://doi.org/10.1145/945445.945467

[26] Brendan I. Koerner. 2016. Inside the Cyberattack That Shocked the US Government. https://www.wired.com/2016/10/inside-cyberattack-shocked-us-government/.

[27] George Kurtz. 2010. Operation Aurora Hit Google, Others. Available at http://securityinnovator.com/index.php?articleID=42948&sectionID=25.

[28] 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. 2018. MCI: Modeling-based Causality Inference in Audit Logging for Attack Investigation.. In *NDSS*.

[29] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition.. In *NDSS*.

[30] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. LogGC: Garbage Collecting Audit Log. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security* (Berlin, Germany) *(CCS '13)*. ACM, New York, NY, USA, 1005–1016. https://doi.org/10.1145/2508859.2516731

[31] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. LogGC: garbage collecting audit log. In *CCS*.

[32] Jure Leskovec. 2009. Stanford network analysis package. *Online, http://snap. stanford. edu* (2009).

[33] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. 2018. Towards a Timely Causality Analysis for Enterprise Security.. In *NDSS*.

[34] 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

[35] 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 *USENIX Security Symposium*.

[36] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In *Proceedings of NDSS '16* (San Diego, CA).

[37] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In *NDSS*.

[38] Keith McCammon. 2018. Evaluating Endpoint Products. https://redcanary.com/blog/evaluating-endpoint-products-in-a-crowded-confusing-market/.

[39] Noor Michael, Jaron Mink, Jason Liu, Sneha Gaur, Wajih Ul Hassan, and Adam Bates. 2020. On the Forensic Validity of Approximated Audit Logs. In *Annual Computer Security Applications Conference* (Austin, USA) *(ACSAC '20)*. Association for Computing Machinery, New York, NY, USA, 189–202. https://doi.org/10.1145/3427228.3427272

[40] Noor Michael, Jaron Mink, Jason Liu, Sneha Gaur, Wajih Ul Hassan, and Adam Bates. 2020. On the Forensic Validity of Approximated Audit Logs. In *ACSAC*.

[41] Sadegh M Milajerdi, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. 2019. Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting. In *CCS*.

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

[43] MITRE Corporation. 2019. APT29. https://attack.mitre.org/groups/G0016/.

[44] MITRE Corporation. 2019. APT3. https://attack.mitre.org/groups/G0022/.

[45] MITRE Corporation. 2019. MITRE ATT&CK. https://attack.mitre.org.

[46] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. 2006. Provenance-aware Storage Systems. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference* (Boston, MA) *(Proceedings of the 2006 Conference on USENIX Annual Technical Conference)*.

[47] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher W. Fletcher, Andrew Miller, and Dave Tian. 2020. Custos: Practical Tamper-Evident Auditing of Operating Systems Using Trusted Execution. In *27th ISOC Network and Distributed System Security Symposium (NDSS'20)*.

[48] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, and Margo Seltzer. 2018. Runtime analysis of whole-system provenance. In *CCS*.

[49] Nicole Perlroth and David E. Sanger. 2018. Cyberattacks Put Russian Fingers on the Switch at Power Plants, U.S. Says. https://www.nytimes.com/2018/03/15/us/politics/russia-cyberattacks.html.

[50] D.J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler. 2012. Hi-Fi: Collecting High-Fidelity Whole-System Provenance. In *Proceedings of the 2012 Annual Computer Security Applications Conference (ACSAC '12)*. Orlando, FL, USA.

[51] Michael Riley, Ben Elgin, Dune Lawrence, and Carol Matlack. 2019. Target Missed Warnings in Epic Hack of Credit Card Data. https://bloom.bg/2KjElxM.

[52] Dan Sullivan. 2016. Splunk Enterprise Security: Product overview. https://www.techtarget.com/searchsecurity/feature/Splunk-Enterprise-Security-Product-overview.

[53] Symantec EDR 4.6 Docs. 2022. About purging reports. https://techdocs.broadcom.com/us/en/symantec-security-software/endpoint-security-and-management/endpoint-detection-and-response/4-6/about-reports-v117056913-d38e36074/about-purging-reports-v118097546-d38e36892.html.

[54] Symantec EDR 4.6 Docs. 2022. How Symantec EDR purges data from the Symantec EDR database. https://techdocs.broadcom.com/us/en/symantec-security-software/endpoint-security-and-management/endpoint-detection-and-response/4-6/Settings/how-purges-data-from-the-database-v106460598-d38e46998.html.

[55] 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* (Toronto, Canada) *(CCS '18)*. ACM, New York, NY, USA, 1324–1337. https://doi.org/10.1145/3243734.3243763

[56] 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 *CCS*.

[57] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. 2018. Fear and logging in the internet of things. In *NDSS*.

[58] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Junghwan Rhee, Zhengzhang Chen, Wei Cheng, C Gunter, et al. 2020. You are what you do: Hunting stealthy malware via data provenance analysis. In *NDSS*.

[59] 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* (Vienna, Austria) *(CCS '16)*. ACM, New York, NY, USA, 504–516. https://doi.org/10.1145/2976749.2978378

[60] 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 *CCS*.

Algorithm 2: NodeMerge

```
Input :
rawEvents: Original audit raw events
Output:
reducedEvents: Audit events filtered using NodeMerge
Function NodeMerge():
    // Phase-1 : Learning Templates;
    FAP ← generateFAP(rawEvents);
    FAP ← FAP.filter(); // only keep read-only files for each process;
    root ← new FPTreeNode();
    foreach process, file_list ∈ FAP do
        FPTreeInsert(file_list, root);
    // CFAPS obtained by applying FP Growth algrithm on FP Tree
    CFAPS ← set of files that appears > 2 times together in the FAP;
    // Phase-2 : Reducing Logs;
    reducedEvents ← [];
    FSA ← buildFSA(CFAPS);
    foreach process, file_list ∈ FAP do
        if file_list mathces FSA then
            eventList ← All read events corresponding to FAP entry;
            reducedEvents.append(eventList);
    return reducedEvents;
Function generateFAP(E: events):
    Map FAP ← ∅;
    Map StartTime ← ∅;
    k ← 2; // length of the initial stage in seconds;
    foreach e ∈ E with process p do
        if e is a process start event then
            FAP[p] ← [];
            StartTime[p] ← e.timeStamp;
        else if e is a file read event ∧ (e.timeStamp - StartTime[p]) < k then
            FAP[p].append(e.file);
    return FAP;
Function FPTreeInsert(F: file list, N: node):
    if F[0].file_id == N.file_id then
        N.counter++;
        FPTreeInsert(F[1:], N);
    else
        Add F[0] as a child of N;
        FPTreeInsert(F[1:], F[0]);
Function buildFSA(L: file set):
    FSA,F ← ∅;
    Add an initial state, Init, for F;
    foreach cfap ∈ L do
        Rank items in cfap based on their file IDs in list r;
        forall i ∈ [0,r.length) do
            create state Sr[i] in F;
            Add transition (Sr[i], r[i]) → Sr[i+1] to F;
            Add transition (Sr[i], !r[i]) → Sr[i] to F;
        Add transition (Init, r[0]) → Sr[1] to F;
        Reduce F;
    return FSA;
```

Algorithm 3: Causality Preserved Reduction

```
Input :
e: the edge which is going to be worked on
src: the source node of an event edge
dst: the destination node of an event edge
Function CPR(e: edge, src: node, dst: node):
    // Create edge between source node and destination node;
    e0 ← the edge between (src, dst) with greatest end timestamp;
    if e0 does not exist ∨ ¬ interleavedFlow(e0, src) then
        e.remove(); // remove this edge;
        e0.end ← e.end;
Function interleavedFlow(v: edge, vsrc: node):
    foreach it ∈ the in-edges of vsrc do
        if it.end > v.end then
            return True;
    return False;
```

[61] Carter Yagemann, Mohammad Noureddine, Wajih Ul Hassan, Simon Chung, Adam Bates, and Wenke Lee. 2021. Validating the Integrity of Audit Logs Against Execution Repartitioning Attacks. In *CCS*.

[62] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. 2011. Secure Network Provenance. In *ACM Symposium on Operating Systems Principles (SOSP)*.

A REDUCTION ALGORITHMS

A.1 LogGC

The implementation of the LogGC filter is shown in Algorithm 1. While the original LogGC system [31] was described as an offline log-based algorithm, we re-implemented the system in an event-driven fashion. This event-driven filter is invoked upon file and process termination events. The filter identifies and prunes temporary file I/O and other dead-end events from the raw audit logs.

Algorithm 1: Garbage Collection (LogGC)

```
Input :
n: The node in the provenance graph
Output:
newEvents: audit events filtered after grabage collection
Function LogGC(n : node):
    if can_perform_gc(n) then
        forall edges e ∈ n.get_edges() do
            e.remove();
        forall node i ∈ n.get_parents_nodes() do
            LogGC(i);
Function can_perform_gc(v : node):
    if is_process(v) ∧ is_dead(v) ∧ not_affect_state(v) then
        return True;
    else if is_file(v) ∧ is_deleted(v) ∧ is_temp_file() then
        return True;
    else if is_file(v) ∧ is_dead_end(v) then
        return True;
    else
        return False;
```

A.2 NodeMerge

The implementation of the NodeMerge filter is presented in Algorithm 2. It is invoked at the end of each batch and consists of three phases. In the first phase, the File Access Pattern (FAP) is generated from the raw audit logs. In the next phase, FAP is utilised to generate the Compressible File Access Pattern (CFAP) for data reduction. In the final phase, CFAP is used to reduce the incoming streamed data.

A.3 CPR

The implementation of the CPR filter is shown in Algorithm 3. The CPR filter is invoked for every read or write event corresponding to an edge between source node s and destination node d. The filter checks that there are no interleaved flows that would result in false data dependencies, if so, the previous edge e_0 and new edge e are merged into a single edge with e_0's start timestamp and e's end timestamp. Otherwise, if no reduction can be performed, both edges are left intact. We inductively know that no edges other than e_0 need to be checked, as e_0 must already be merged as much as possible due to FAuST' streaming architecture.

A.4 F-DPR and S-DPR

The implementation of the S-DPR and F-DPR filters is outlined in Algorithm 4. These algorithms require graph-wide analysis, and are thus applied periodically at the end of each batch. The objective of F-DPR is to reduce the set E to the smallest edge set where forwards

and backwards reachability between all nodes (N) is preserved. S-DPR is more aggressive, and reduces E to the smallest set where only backwards reachability for all nodes in N to the all source nodes in S is preserved. Reachability is determined via breadth-first search (BFS), where nodes are visited in order of timestamps. To maintain backwards reachability, we compute the set of nodes that a given node n can reach; for each of these nodes, n is backwards-reachable. To maintain forward reachability, we compute the reverse, i.e., the set of nodes that can reach n. Once we know the set of reachable nodes R from starting node s, we compute the set of edges required to maintain reachability such that, for each node n in R, we only retain the oldest in-edge from another node in R to n.

Algorithm 4: Full and Source-Dependence Preserving Red.

```
Input :
G: Provenance graph consisting of audit events
E: The set of all edges in the G
N: The set of all nodes in the G
Function SDPR():
    Set S ← {n ∈ N|indegree(n) = 0};
    Set R ← E \ required_edges(S);
    foreach edge r ∈ R do
        r.remove();
Function FDPR():
    Set S ← N;
    Set R_b ← E \ required_edges(S);
    Set R_f ← required_edges_forwards(S); // required_edges_forwards(S) is
      identical to required_edges, but for forward reachability;
    R ← E \ R_b \ R_f;
    foreach edge r ∈ R do
        r.remove();
Function required_edges(S: node set):
    Set E ← ∅;
    foreach node s ∈ S do
        R ← backReach_bfs(s);
        foreach node r in R where r ≠ s do
            e ← oldest edge where r is the in-node ∧ both nodes of e ∈ R;
            E.add(e);
    return E;
Function backReach_bfs(start: node):
    Set R ← ∅;
    Set Q ← PriorityQueue((start, −∞)); // priority queue of (node, timestamp)
      pairs - priority: greatest timestamp;
    while Q neq ∅ do
        n, t ← Q.dequeue();
        R.add(n);
        foreach edge e = (n, u) ∈ n.get_out_edges() where u ∉ R ∧ e.start ≥ t
          do
            Q.enqueue((u, t));
    return R;
```

A.5 PCAR

Algorithm 5 presents the implementation of the PCAR filter. The filter extends the CPR system and is invoked for every read or write event corresponding to an edge between source node s and destination node d. It eliminates redundant events (even for interleaved flows) when a bursty process exceeds a pre-defined number of system calls within a time window. The filter track bursts by keeping a sliding window of recent IO activity, represented as an event set per node. Whenever an edge e is created from source s to destination d, we update this window for both s and d by adding e to the set and discarding any events that no longer fall within the window. Then, if the event set size is large enough to be considered a burst, we set the burst flag for the node. Finally, the filter checks if either s or d have the burst flag set; if so, it merges e with previous edge e_0.

Algorithm 5: Process-centric Causality Approx Red.

```
Input :
e: the edge which is going to be worked on
src: the source node of an event edge
dst: the destination node of an event edge
IBURST_WINDOW: length of the sliding time window for recent IO activity
IBURST_EVENTS: min. number of events that constitute a burst
Function PCAR(e: edge, src: node, dst: node):
    track_burst(e.event, src);
    track_burst(e.event, dst);
    e_0 ← the edge between (src, dst) with greatest end timestamp;
    if e_0 does not exist ∨ ¬ interleavedFlow(e_0, src) ∨ src.is_burst ∨
      dst.is_burst then
        e.remove();
        e_0.end ← e.end;
Function track_burst(v: edge, vsrc: node):
    foreach event e' ∈ vsrc.burst_set do
        if e'.time + IBURST_WINDOW < v.time then
            vsrc.burst_set.remove(e');
    vsrc.burst_set.add(v);
    if vsrc.burst_set.size() ≥ IBURST_EVENTS then
        vsrc.is_burst ← True;
    else
        vsrc.is_burst ← False;
```

Algorithm 6: LogApprox

```
Input :
G: Provenance graph consisting of audit events
E: The set of all edges (events) in the G
N: The set of all nodes in the G
Function LogApprox():
    Set S ← compute_file_groups;
    Set R ← compute_regexes(S);
    foreach group S_i ∈ S do
        forall reads e to file f ∈ S_i do
            if since last read, a write to f happened ∨ a read from p to a file ≠ f then
                Keep e;
                Overwrite f with r_i;
            else
                e.remove();
        forall writes e to file f ∈ S_i do
            if since last read, a write to f happened ∨ a read from p to a file ≠ f then
                Keep e;
                Overwrite f with r_i;
            else
                e.remove();
```

Algorithm 7: Deterministic Finite Automata Winnower

```
Input :
G: Provenance graph consisting of audit events
Output:
removedEvents: audit events filtered through winnower
Function Winnower():
    Set Gram ← ∅;
    List removedEvents ← [];
    foreach node ∈ G do
        τ_prefix ← getPrefixTree(node);
        τ_suffix ← getSuffixTree(node);
        if {τ_prefix, τ_suffix} ∈ Gram then
            // get all events associated with node;
            nodeEvents ← getEvents(node);
            removedEvents.append(nodeEvents);
        else
            Gram ← Gram ∪ {τ_prefix, τ_suffix, vertex.label};
    return removedEvents;
```

A.6 LogApprox

The implementation of the LogApprox filter is shown in Algorithm 6. The filter is invoked periodically at the end of each batch and consists of two phases. First, it generates regular expressions that describe benign process file I/O. Second, for all events that match the same regex, the original filename is replaced with the regex pattern. After the filenames are replaced, all the events that do not alter the information flow are removed from the logs.

A.7 Winnower

The implementation of the Winnower filter is shown in Algorithm 7. The filter is invoked at the end of each batch. It first generates the prefix tree sets and suffix tree sets for each vertex in Graph, and the results are used to create a grammar. Afterward, if the same tuple defines two vertices within the grammar, they are considered equivalent and implicitly merged in the final graph.