Dependence-Preserving Data Compaction for Scalable Forensic Analysis[*]

Md Nahid Hossain, Junao Wang, R. Sekar, and Scott D. Stoller
{mdnhossain,junawang,sekar,stoller}@cs.stonybrook.edu
Stony Brook University, Stony Brook, NY, USA.

Abstract

Large organizations are increasingly targeted in long-running attack campaigns lasting months or years. When a break-in is eventually discovered, forensic analysis begins. System audit logs provide crucial information that underpins such analysis. Unfortunately, audit data collected over months or years can grow to enormous sizes. Large data size is not only a storage concern: forensic analysis tasks can become very slow when they must sift through billions of records. In this paper, we first present two powerful event reduction techniques that reduce the number of records by a factor of 4.6 to 19 in our experiments. An important benefit of our techniques is that they provably preserve the accuracy of forensic analysis tasks such as backtracking and impact analysis. While providing this guarantee, our techniques reduce on-disk file sizes by an average of 35× across our data sets. On average, our in-memory dependence graph uses just 5 bytes per event in the original data. Our system is able to consume and analyze nearly a million events per second.

1 Introduction

Many large organizations are targets of stealthy, long-term, multi-step cyber-attacks called Advanced Persistent Threats (APTs). The perpetrators of these attacks remain below the radar for long periods, while exploring the organization's IT infrastructure and exfiltrating or compromising sensitive data. When the attack is ultimately discovered, a forensic analysis is initiated to identify the entry points of the attack and its system-wide impact. The spate of APTs in recent years has fueled research on efficient collection and forensic analysis of system logs [13, 14, 15, 9, 16, 17, 18, 22, 42, 30, 10].

Accurate forensic analysis requires logging of system activity across the enterprise. Logs should be detailed enough to track dependencies between events occurring on different hosts and at different times, and hence needs to capture all information-flow causing operations such as network/file accesses and program executions. There are three main options for collecting such logs: (1) instrumenting individual applications, (2) instrumenting the operating system (OS), or (3) using network capture techniques. The rapid increase in encrypted traffic has greatly reduced the effectiveness of network-capture based forensic analysis. In contrast, OS-layer logging is unaffected by encryption. Moreover, OS-layer logging can track the activities of *all processes* on a host, including any malware that may be installed by the attackers. In contrast, application-layer logs are limited to a handful of benign applications (e.g., network servers) that contain the instrumentation for detailed logging. For these reasons, we rely on OS-based logging, e.g., the Linux audit and Windows ETW (Event Tracing for Windows) systems.

1.1 Log Reduction

APT campaigns can last for many months. With existing systems, such as Linux auditing and Windows ETW, our experience as well as that of previous researchers [42] is that the volume of audit data is in the range of gigabytes *per host per day*. Across an enterprise with thousands of hosts, total storage requirements can easily go up to the petabyte range in a year. This has motivated a number of research efforts on reducing log size.

Since the vast majority of I/O operations are reads, ProTracer's [22] reduction strategy is to log only the writes. In-memory tracking is used to capture the effect of read operations. Specifically, when a process performs a read, it acquires a taint identifier that captures the file, network or IPC object read. If the process reads n files, then its taint set can be of size $O(n)$. Write operations are logged, together with the taint set of the process at that point. This means that write records can, in general, be of size $O(n)$, and hence a process performing m writes can produce a log of size $O(mn)$. This contrasts with the $O(m+n)$ log size that would result with traditional OS-level logging of both reads and writes. Thus, for ProTracer's strategy to reduce log size, it is necessary to narrow the size of taint sets of write operations to be close to 1. They achieve this using a fine-grained taint-tracking technique called *unit-based execution partitioning* [17], where a *unit* corresponds to a loop iteration. MPI [21] proposes a new form of execution partitioning, based on annotated data structures instead of loops. However, fine-grained taint-tracking via execution partitioning would be difficult to deploy on the scale of a large enterprise running hundreds of applications or more. Without fine-grained taint-tracking, the analysis above, as well as our experiments, indicate that this strategy of "alternating tainting with logging" leads to *substantial increases* in log size.

LogGC [18] develops a "garbage collection" strategy, which identifies and removes operations that have no persistent effect. For instance, applications often create temporary

[*]This work was primarily supported by DARPA (contract FA8650-15-C-7561) and in part by NSF (CNS-1319137, CNS-1421893, CCF-1414078) and ONR (N00014-15-1-2208, N00014-15-1-2378, N00014-17-1-2891).

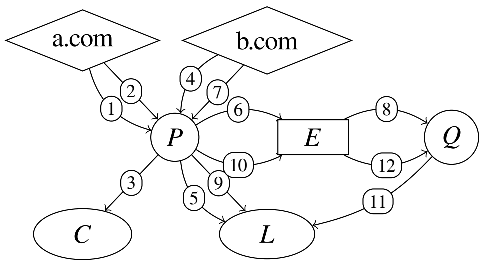

Fig. 1: An example (time-stamped) dependence graph.

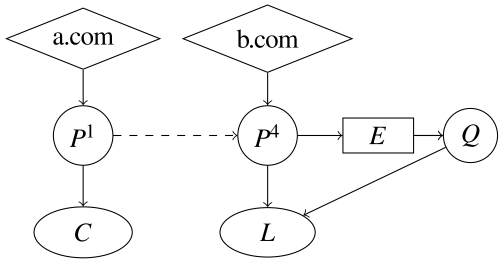

Fig. 2: Dependence graph resulting after our FD log reduction. SD reduction will additionally remove the edge from Q to L. In this reduced graph, dependence can be determined using standard graph reachability. Edge timestamps are dropped, but nodes may be annotated with a timestamp.

files that they subsequently delete. Unless these files are accessed by other processes, they don't introduce any new dependencies and hence aren't useful for forensic analysis. However, some temporary files do introduce dependencies, e.g., malware code that is downloaded, executed and subsequently removed by another attack script. Operations on such files need to be logged, so LogGC introduces a notion of exclusive ownership of files by processes, and omits only the operations on exclusively owned files. Although they report major reductions in log size using this technique, this reduction is realized only in the presence of the unit instrumentation [17] described above. If only OS-layer logging is available, which is the common case, LogGC does not produce significant reductions. (See the "Basic GC" column in Table 5 of [18].)

While LogGC removes all events on a limited class of objects, Xu et al [42] explore a complementary strategy that can remove some (repeated) events on any object. To this end, they developed the concept of *trackability equivalence* of events in the audit log, and proved that, among a set of equivalent events, all but one can be removed without affecting forensic analysis results. Across a collection of several tens of Linux and Windows hosts, their technique achieved about a 2× reduction in log size. This is impressive, considering that it was achieved without any application-specific optimizations.

While trackability equivalence [42] provides a sufficient basis for eliminating events, we show that it is far too strict, limiting reductions in many common scenarios, e.g., communication via pipes. The central reason is that trackability is based entirely on a local examination of edges incident on a single node in the dependence graph, without taking into account any global graph properties. In contrast, we develop a more general formulation of dependence preservation that can leverage global graph properties. It achieves 3 to 5 times as much reduction as Xu et al.'s technique.

- In Section 3, we formulate *dependence-preserving log reduction* in terms of reachability preservation in the *dependence graph*. As in previous works (e.g., [13, 42]), nodes in our dependence graph represent *objects* (files, sockets and IPCs) and *subjects* (processes), while edges represent operations (also called *events*) such as read, write, load, and execute. Edges are timestamped and are oriented in the direction of information flow. We say that a node v depends on node u if there is a (directed) path from u to v with non-decreasing edge timestamps. In Fig. 1, P denotes a process that connects to a.com, and downloads and saves a file C. It also connects to b.com, and writes to a log file L and a pipe E. Process Q reads from the pipe and also writes to the same log file. Based on timestamps, we can say that C depends on a.com but not b.com.
- Based on this formulation, we present two novel dependency preserving reductions, called *full dependence preservation (FD)* and *source dependence preservation (SD)*. We prove that FD preserves the results of backward as well as forward forensic analysis. We also prove that SD preserves the results of the most commonly used forensic analysis, which consists of running first a backward analysis to find the attacker's entry points, and then a forward analysis from these entry points to identify the full impact of the attack.
- Our experimental evaluation used multiple data sets, including logs collected from (a) our laboratory servers, and (b) a red team evaluation carried out in DARPA's Transparent Computing program. On this data, FD achieved an average of **7**× reduction in the number of events, while SD achieved a **9.2**× reduction. In comparison, Xu et al.'s algorithm [42], which we reimplemented, achieved only a 1.8× reduction. For the example in Fig. 1, our technique combines all edges between the same pair of nodes, leading to the graph shown in Fig. 2, while Xu et al's technique is able to combine only the two edges with timestamps 1 and 2.

1.2 Efficient Computation of Reductions

Our log reductions (FD and SD) rely on global properties of graphs such as reachability. Such global properties are expensive to compute, taking time that is linear in the size of the (very large) dependence graph. Moreover, due to the use of timestamped edges, reachability changes over time, and hence the results cannot be computed once and cached for subsequent use.

To overcome these computational challenges posed by timestamped graphs, we show in Section 4 how to transform them into standard graphs. Fig. 2 illustrates the result of this conversion followed by our FD reduction. Note how the edge timestamps have been eliminated. Moreover, P has been split into two *versions* connected by a dashed edge, with each version superscripted with its timestamp. Note the absence of a path from a.com to C, correctly capturing the reachability information in the timestamped graph in Fig. 1.

Versioning has been previously studied in file system and provenance research [31, 26, 25]. In these contexts, versioning systems typically intervene to create file versions that provide increased recoverability or reproducibility. Provenance capture systems may additionally intervene to break cyclic dependencies [24, 25], since cyclic provenance is generally considered meaningless.

In our forensic setting, we cannot intervene, but can only observe events. Given a timestamped event log, we need to make sound inferences about dependencies of subjects as well as objects. We then encode these dependencies into a standard graph in order to speed up our reduction algorithms. The key challenge in this context is to minimize the size of the standard graph without dropping any existing dependency, or introducing a spurious one. Specifically, the research described in Section 4 makes the following contributions:

- *Efficient reduction algorithms.* By working with standard graphs, we achieve algorithms that typically take constant time per event. In our experiments, we were able to process close to a million events per second on a single-core on a typical laptop computer.
- *Minimizing the number of versions.* We present several optimization techniques in Section 4.2 to reduce the number of versions. Whereas naive version generation leads to an explosion in the number of versions, our optimizations are very effective, bringing down the average number of versions per object and subject to about 1.3. Fig. 2 illustrates a few common cases where we achieve substantial reductions by combining many similar operations:
  - multiple reads from the same network connection (a.com, b.com) interleaved with multiple writes to files (C and L),
  - series of writes to and reads from pipes (E), and
  - series of writes to log files by multiple processes (L).
- *Avoiding spurious dependencies.* While it is important to reduce the space overhead of versions, this should not come at the cost of inaccurate forensic analysis. We therefore establish formally that results of forensic analysis (specifically, forward and backward analyses) are fully preserved by our reduction.
- *Optimality.* We show that edges and versions retained by our reduction algorithm cannot be removed without introducing spurious dependencies.

An interesting aspect of our work is that we use versioning to reduce storage and runtime, whereas versioning is normally viewed as a performance cost to be paid for better recoverability or reproducibility.

1.3 Compact Graph and Log Representations

A commonly suggested approach for forensic analysis is to store the dependence graph in a graph database. The database's query capabilities can then be used to perform backward or forward searches, or any other custom forensic analysis. Graph databases such as OrientDB, Neo4j and Titan are designed to provide efficient support for graph queries, but experience suggests that their performance degrades dramatically on graphs that are large relative to main memory. For instance, a performance evaluation study on graph databases [23] found that they are unable to complete simple tasks, such as finding shortest paths on graphs with 128M edges, even when running on a computer with 256GB main memory and sufficient disk storage. Log reduction techniques can help, but may not be sufficient on their own: our largest dataset, representing just one week of data, already contains over 70M edges. Over the span of an APT (many months or a year), graph sizes can approach a billion edges *even after log reduction.* We therefore develop a compact in-memory representation for our versioned dependence graphs.

- Section 5.2 describes our approach for realizing a compact dependence graph representation. By combining our log reduction techniques with compact representations, our system achieves very high density: it uses about 2 bytes of main memory per event on our largest data set. This dataset, with 72M edges, is comparable in size to the 128M edges used in the graph database evaluation [23] mentioned above. Yet, our memory utilization was just 111MB, in comparison with the 256GB available in that study.

- We also describe the generation of compact event logs based on our event reduction techniques (Section 5.1). We began with a space-efficient log format that was about **8**$\times$ smaller than a Linux audit log containing roughly the same information. With FD reduction, it became **35.3**$\times$ smaller, while SD increased the reduction factor to about **41.4**$\times$. These numbers are before the application of any data compression techniques such as `gzip`, which can provide further reductions.

1.4 Paper Organization

We begin with some background on forensic analysis in Section 2. The formulation of dependence-preserving reductions, together with our FD and SD techniques, are presented in Section 3. Efficient algorithms for achieving FD and SD are described in Section 4, together with a treatment of correctness and optimality. Section 5 summarizes a compact main-memory dependence graph and offline event log formats based on our event reductions. Implementation and experimental evaluation are described in Section 6, followed by related work discussion in Section 7 and concluding remarks in Section 8.

2 Background

Dependence graphs. System logs refer to two kinds of *entities: subjects* and *objects.* Subjects are processes, while objects correspond to passive entities such as files, network connections and so on. Entries in the log correspond to *events,* which represent actions (typically, system calls) performed by subjects, e.g., read, write, and execute.

In most work on forensic analysis [13, 15, 42], the log contents are interpreted as a *dependence graph:* nodes in the graph correspond to entities, while edges correspond to events. Edges are oriented in the direction of information flow and have timestamps. When multiple instances of an event are aggregated into a single instance, its timestamp becomes the interval between the first and last instances. Fig. 1 shows a sample dependence graph, with circles denoting subjects, and the other shapes denoting objects. Among objects, network connections are indicated by a diamond, files by ovals, and pipes by rectangles. Edges are timestamped, but their names omitted. Implicitly, in-edges of subjects denote reads, and out-edges of subjects denote writes.

Backward and Forward Analysis. Forensic analysis is concerned with the questions of *what, when* and *how.* The *what* question concerns the *origin* of a suspected attack, and the entities that have been impacted during an attack. The origin can be identified using *backward analysis,* starting from an entity flagged as suspicious, and tracing backward in the graph. This analysis, first proposed in BackTracker [13], uses event timestamps to focus on paths in dependence graphs that represent causal chains of events. A backward analysis from file C at time 5 will identify P and a.com. Of these, a.com is a source node, i.e., an object with no parent nodes, and hence identified as the likely entry point of any attack on C.

Although b.com is backward reachable from C in the standard graph-theoretic sense, it is excluded because the path from b.com to C does not always go forward in time.

The set of entities impacted by the attack can be found using *forward analysis* [43, 1, 15] (a.k.a. *impact analysis*), typically starting from an entry point identified by backward analysis. In the sample dependence graph, forward analysis from network connection a.com will reach all nodes in the graph, while a forward analysis from b.com will leave out C.

The *when* question asks when each step in the attack occurred. Its answer is based on the timestamps of edges in the subgraph computed by forward and backward analyses. The *how* question is concerned with understanding the steps in an attack in sufficient detail. To enable this, audit logs need to capture all key operations (e.g., important system calls), together with key arguments such as file names, IP addresses and ports, command-line options to processes, etc.

3 Dependence Preserving Reductions

We define a *reduction* of a time-stamped dependence graph G to be another graph G' that contains the same nodes but a subset of the events. Such a reduction may remove "redundant" events, and/or combine similar events. As a result, some events in G may be dropped in G', while others may be aggregated into a single event. When events are combined, their timestamps are coalesced into a range that (minimally) covers all of them.

A log reduction needs to satisfy the following conditions:

- it won't change forensic analysis results, and
- it won't affect our understanding of the results.

To satisfy the second requirement, we apply reductions only to *read, write*[1], and *load* events. All other events, e.g., *fork*, *execve*, *remove*, *rename* and *chmod,* are preserved. Despite being limited to reads, writes and loads, our reduction techniques are very effective in practice, as these events typically constitute over 95% of total events.

For the first requirement, our aim is to preserve the results of forward and backward forensic analysis. We ensure this by preserving forward and backward reachability across the original graph G and the reduced graph G'. We begin by formally defining reachability in these graphs.

3.1 Reachability in time-stamped dependence graphs

Dependence graph G is a pair (V,E) where V denotes the nodes in the graph and E denotes a set of directed edges. Each edge e is associated with a start time $start(e)$ and an end time $end(e)$. Reachability in this graph is defined as follows:

Definition 1 (Causal Path and Reachability) *A node v is reachable from another node u if and only if there is (directed) path $e_1, e_2, \ldots, e_n$ from u to v such that:*

$$\forall 1 \le i < n \quad start(e_i) \le end(e_{i+1}) \tag{1}$$

We refer to a path satisfying this condition as a *causal path.* It captures the intuition that information arriving at a node through event e_i can possibly flow out through the event e_{i+1}, i.e., successive events on this path $e_1, e_2, \ldots, e_n$ *can* be causally related. In Fig. 1, the path consisting of edges with timestamps $1, 6, 8$ and 11 is causal, so L is reachable from a.com. In contrast, the path corresponding to the timestamp sequence $4, 3$ is not causal because the first edge occurs later than the second. Hence C is unreachable from b.com.

In forensics, we are interested in reachability of a node at a given time, so we extend the above definition as follows:

Definition 2 (Forward/Backward Reachability at t)

- *A node v is forward reachable from a node u at time t, denoted $u@t \longrightarrow v$, iff there is a causal path $e_1, e_2, \ldots, e_n$ from u to v such that $t \le end(e_i)$ for all i.*
- *A node u is said to be backward reachable from v at time t, denoted $u \longrightarrow v@t$, iff there is a causal path $e_1, e_2, \ldots, e_n$ from u to v such that $t \ge start(e_i)$ for all i.*

[1] There can be many types of read or write events, some used on files, others used on network sockets, and so on. For example, Linux audit system can log over a dozen distinct system calls used for input or output of data. For the purposes of this description, we map them all into reads and writes.

Intuitively, $u@t \longrightarrow v$ means u's state at time t can impact v. Similarly, $u \longrightarrow v@t$ means v's state at t can be caused/explained by u. In Fig. 1, $P@6 \longrightarrow Q$, but $P@11 \not\longrightarrow Q$. Similarly, a.com $\longrightarrow C@3$ but b.com $\not\longrightarrow C@3$.

Based on reachability, we present three dependency-preserving reductions: CD, which is close to Xu et al's full trackability, and FD and SD, two new reductions we introduce in this paper.

3.2 Continuous dependence (CD) preservation

This reduction aims to preserve forward and backward reachability at every instant of time.

Definition 3 (Continuous Dependence Preservation) *Let G be a dependence graph and G' be a reduction of G. G' is said to preserve continuous dependence iff forward and backward reachability is identical in both graphs for every pair of nodes at all times.*

In Fig. 3, S reads from a file F at $t = 2$ and $t = 4$, and writes to another file F' at $t = 3$ and $t = 6$. Based on the above definition, continuous dependence is preserved when the reads by S are combined, as are the writes, as shown in the lower graph.

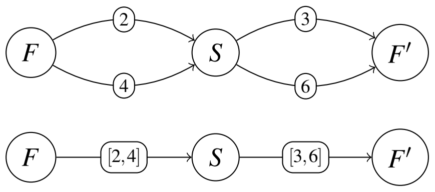

Fig. 3: Reduction that preserves continuous dependence.

Fig. 4 shows a reduction that does *not* preserve continuous dependence. In the original graph, $F@3 \not\longrightarrow H$: the earliest time $F@3$ can affect S is at $t = 4$, and this effect can propagate to $F'@6$, but by this time, the event from F' to H has already terminated. In contrast, in the reduced graph, $F@3$ affects $H@5$.

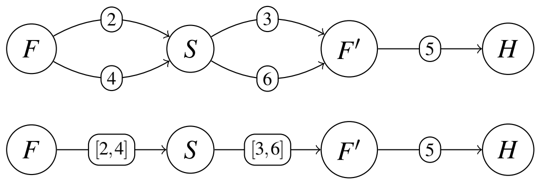

Fig. 4: Reduction that violates continuous dependence.

Our definition of continuous dependence preservation is similar to Xu et al.'s definition of full trackability equivalence [42]. However, their definition is a bit stricter, and does not allow the reductions shown in Fig. 3. They would permit those reductions only if node S had (a) no incoming edges between its outgoing edges and (b) no outgoing edges between its incoming edges[2].

Their stricter definition was likely motivated by efficiency considerations. Specifically, their definition ensures that reduction decisions can be made *locally,* e.g., by examining the edges incident on S. Thus, their criteria does not permit the combination of reads in either Fig. 3 or Fig. 4, since they share the same local structure at node S. In contrast, our continuous dependence definition is based on the more powerful *global* reachability properties, and hence can discriminate between the two examples to safely permit the aggregation in Fig. 3 but not Fig. 4. The downside of this power is efficiency, as continuous dependence may need to examine every path in the graph before deciding which edges can be removed.

Although the checking of global properties can be more time-consuming, the resulting reductions can be more powerful (i.e., achieve greater reduction). This is why we devote Section 4 to development of efficient algorithms to check the more powerful global properties used in the two new reductions presented below.

Because of the similarity of Xu et al's full trackability and our continuous dependence, we will henceforth refer to their approach as *local continuous dependence (LCD) preservation*. We end this discussion with examples of common scenarios where LCD reduction is permitted:

- *Sequence of reads without intervening writes:* When an application reads a file, its read operation results in multiple `read` system calls, each of which is typically logged as a separate event in the audit log. As long as there are no write operations performed by the application at the same time, LCD will permit the reads to be combined.
- *Sequence of writes without intervening reads:* The explanation in this case mirrors the previous case.

However, if reads and writes are interleaved, then LCD does not permit the reads (or writes) to be combined. In contrast, the FD notion presented below can support reductions in cases where an application is reading from one or more files while writing to one or more files.

3.3 Full Dependence (FD) Preservation

CD does not permit the reduction in Fig. 4, because it changes whether the state of F at $t = 3$ propagates to H. But does this difference really matter in the context of forensic analysis? To answer this question, note that there is no way for F to become compromised at $t = 3$ if it was not already compromised before. Indeed, there is no basis for the state of F to change between $t = 0$ and $t = 6$ because nothing happens to F during this period.

More generally, subjects and objects don't spontaneously become compromised. Instead, compromises happen due to input consumption from a compromised entity, such as a network connection, compromised file, or user[3]. This

[2]In particular, as per Algorithm 2 in [42], the period of the incoming edge in Fig. 3 should not overlap the period between the end times of the two edges out of S; per their Algorithm 3, the period of the S to F' edge must not overlap the period between the start times of the two edges out of F.

[3]We aren't suggesting that a compromised process must *immediately*

observation implies that keeping track of dependencies between entities at times strictly in between events is unnecessary, because nothing relevant changes at those times. Therefore, we focus on preserving dependencies at times when a node could become compromised, namely, when it acquires a new dependency.

Formally, let $Anc(v,t)$ denote the set of ancestor nodes of v at time t, i.e., they are backward reachable from v at t.

$$Anc(v,t) = \{u \mid u \longrightarrow v@t\}.$$

Let $NewAnc(v)$ be the set of times when this set changes, i.e.:

$$NewAnc(v) = \{t \mid \forall t' < t,\ Anc(v,t) \supset Anc(v,t')\}.$$

We define $NewAnc(v)$ to always include $t = 0$.

Definition 4 (Full Dependence (FD) Preservation) *A reduction G' of G is said to preserve full dependence iff for every pair of nodes u and v:*

- *forward reachability from $u@t$ to v is preserved for all $t \in NewAnc(u)$, and*
- *backward reachability of u from $v@t$ is preserved at all t.*

In other words, when FD-preserving reductions are applied:

- the result of backward forensic analysis from any node v will identify the exact same set of nodes before and after the reduction.
- the result of forward analysis carried out from any node u will yield the exact same set of nodes, as long as the analysis is carried out at any of the times when there is a basis for u to get compromised.

To illustrate the definition, observe that FD preservation allows the reduction in Fig. 4, since (a) backward reachability is unchanged for every node, and (b) $NewAnc(F) = \{0\}$, and $F@0$ flows into S, F' and H in the original as well as the reduced graphs.

3.4 Source Dependence (SD) Preservation

We consider further relaxation of dependence preservation criteria in order to support more aggressive reduction, based on the following observation about the typical way forensic analysis is applied. An analyst typically flags an entity as being suspicious, then performs a backward analysis to identify likely root causes. Root causes are *source* nodes in the graph, i.e., nodes without incoming edges. Source nodes represent network connections, preexisting files, processes started before the audit subsystem, pluggable media devices, and user (e.g., terminal) input. Then, the analyst performs an impact (i.e., forward) analysis from these source nodes. To carry out this task accurately, we need to preserve only information flows from source nodes; preserving dependencies between all pairs of internal nodes is unnecessary.

exhibit suspicious behavior. However, in order to fully investigate the extent of an attack, forensic analysis needs to focus on the earliest time a node could have been compromised, rather than the time when suspicious behavior is spotted. Otherwise, the analysis may miss effects that may have gone unnoticed between the time of compromise and the time suspicious behavior was observed.

Definition 5 (Source Dependence (SD) Preservation) *A reduction G' of G is said to preserve source dependence iff for every node v and a* ***source node*** *u:*

- *forward reachability from $u@0$ to v is preserved, and*
- *backward reachability of u from $v@t$ is preserved at all t.*

Note that SD coincides with FD applied to source nodes. The second conditions are, in fact, identical. The first conditions coincide as well, when we take into account that $NewAnc(u) = \{0\}$ for any source node u. (A source node does not have any ancestors, but since we have defined $NewAnc$ to always include zero, $NewAnc$ of source nodes is always $\{0\}$.)

Fig. 5 shows a reduction that preserves SD but not FD. In the figure, F and F' are two distinct files, while S, S' and S'' denote three distinct processes. Note that FD isn't preserved because a new flow arrives at S' at $t = 2$, and this flow can reach F' in the original graph but not in the reduced graph. However, SD is preserved because the reachability of S, S', S'' and F' from the source node F is unchanged.

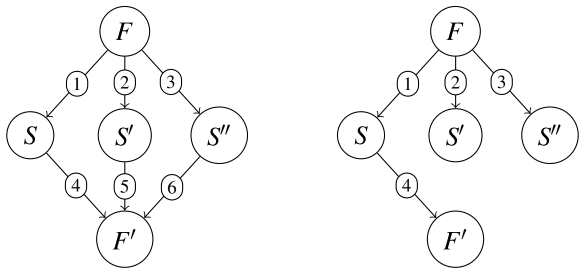

Fig. 5: Source dependence preserving reduction.

Note that the first condition in Defn. 5 is redundant, as it is implied by the second: If u is backward reachable from a node v at t, then, by definition of backward reachability, there exists a causal path from $e_1, e_2, \ldots, e_n$ from u to v. Since 0 is the smallest possible timestamp, $0 \leq end(e_i)$ for all i, and hence, using the causal path $e_1, e_2, \ldots, e_n$ and the first part of Defn. 2, we conclude $u@0 \longrightarrow v$, thus satisfying the first condition. We also point out that the first condition does not imply the second. To see this, note that if we only need to preserve forward reachability from $F@0$ in Fig. 5, then we can drop any two of the three edges coming into F'. However, the backward reachability condition limits us to dropping the edges from S' and S'', as we would otherwise change backward reachability of the source node F from $F'@4$.

Despite being unnecessary, we kept the first condition in Defn. 5 because its presence makes the forensic analysis preservation properties of SD more explicit. (Unlike Defn. 5, there is no redundancy in Defn. 4.)

4 Efficient Computation of Reductions

Full dependence and source dependence reductions rely on global properties of graph reachability. Such global properties are expensive to compute, taking time that can be linear in the size of the (very large) dependence graph. Moreover, due to the use of timestamped edges, reachability changes over time and hence must be computed many times. This mutability also means that results cannot be computed once and cached for subsequent use, unlike standard graphs, where we can determine once that v is a descendant of u and reuse this result in the future.

To overcome these computational challenges posed by timestamped graph, we show how to transform them into standard graphs. The basic idea is to construct a graph in which objects as well as subjects are *versioned.* Versioning is widely used in many domains, including software configuration management, concurrency control, file systems [31, 26] and provenance [25, 24, 4, 29]. In these domains, versioning systems typically intervene to create file versions, with the goal of increased recoverability or reproducibility. In contrast, we operate in a forensic setting, where we can only observe the order in which objects (as well as subjects) were accessed. Our goal is to (a) make sound inferences about dependencies through these observations, and (b) encode these dependencies in a standard (rather than time-stamped) graph. This encoding serves as the basis for developing efficient algorithms for log reduction. Specifically, this section addresses the following key problems.

- Formally establishing that versioned graphs produce the same forensic analysis results as timestamped graphs.
- Developing a suite of optimizations that reduce the number of versions while preserving dependencies.
- Showing that our algorithms generate the optimal number of versions while preserving FD or SD.

Using versioning, we realize algorithms that are both faster and use less storage than their unversioned counterparts. Specifically, we realize substantial reduction in the size of the dependence graph by relying on versioning. Runtime is also reduced because the reduction operations typically take constant time per edge (See Section 6.6.1). In contrast, a direct application of Defn. 4 on timestamped graphs would be unacceptably slow[4].

4.1 Naive Versioned Dependence Graphs

The simplest approach for versioning is to create a new version of a node whenever it gets a new incoming edge, similar to creating a new file version each time the file is written. Fig. 6 shows an example of an unversioned graph and its corresponding naive versioned graph. Versions of a node are stacked vertically in the example so as to make it easier to see the correspondence between nodes in the timestamped and versioned graphs.

[4]In order to determine if an edge e is redundant, we would potentially have to consider every path in the graph containing e; the number of such paths can be exponential in the size of the graph.

Note that timestamps in versioned graphs are associated with nodes (versions), not with edges. A version's start time is the start time of the event that caused its creation. We show this time using a superscript on the node label.

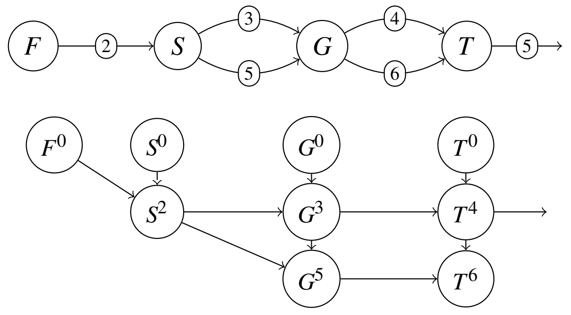

Fig. 6: A timestamped graph and equivalent naive versioned graph.

4.1.1 Algorithm for naive versioned graph construction

We treat the contents of the audit log as a timestamped graph $G = (V, E_T)$. The subscript T on E is a reminder that the edges are timestamped. The corresponding (naive) versioned graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ is constructed using the algorithm shown below. Without loss of generality, we assume that every edge in the audit log has a unique timestamp and/or sequence number. We denote a directed edge from u to v with timestamp t as a triple (u, v, t). Let $u^{<t}$ denote the latest version of u in the versioned graph before t.

```
1. $BuildVer(V, E_T)$
2.   $\mathbf{V} = \{v^0 | v \in V\}$; $\mathbf{E} = \{\}$;
3.   for each $(u, v, t) \in E_T$
4.       add $v^t$ to $\mathbf{V}$
5.       add $(u^{<t}, v^t)$ to $\mathbf{E}$
6.       add $(v^{<t}, v^t)$ to $\mathbf{E}$
7.   return $(\mathbf{V}, \mathbf{E})$
```

We intend *BuildVer* and its optimized versions to be *online algorithms,* i.e., they need to examine edges one-at-a-time, and decide immediately whether to create a new version, or to add a new edge. These constraints are motivated by our application in real-time attack detection and forensic analysis.

For each entity v, an initial version v^0 is added to the graph at line 2.[5] The for-loop processes log entries (edges) in the order of increasing timestamps. For an edge (u, v) with timestamp t, a new version v^t of the target node v is added to the graph at line 4. Then an edge is created from the latest version of u to this new node (line 5), and another edge created to link the last version of v to this new version (line 6).

[5]This is a logical simplification — in reality, initial version of v will be added to the graph at the first occurrence of v in the audit stream.

4.1.2 *Forensic analysis on versioned graphs*

In a naive versioned graph, each object and subject gets split into many versions, with each version corresponding to the time period between two consecutive incoming edges to that entity in the unversioned graph. To flag an entity v as suspicious at time t, the analyst marks the latest version $v^{\leq t}$ of v at or before t as suspicious. Then the analyst can use standard graph reachability in the versioned graph to perform backward and forward analysis. For the theorem and proof, we use the notation $v^{<\infty}$ to refer to the latest version of v so far. In addition, we make the following observation that readily follows from the description of *BuildVer*.

Observation 6 *For any two node versions u^t and u^s, there is a path from u^t to u^s if and only if $s \geq t$.*

Theorem 7 *Let $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ be the versioned graph constructed from $G = (V, E_T)$. For all nodes u, v and times t:*

- *v is forward reachable from $u@t$ iff there is a simple path in $\mathbf{G}$ from $u^{\leq t}$ to $v^{<\infty}$; and*
- *u is backward reachable from $v@t$ iff there is a path in $\mathbf{G}$ from u^0 to $v^{\leq t}$.*

Proof: For uniformity of notation in the proof, let $t = t_0, u = w_0$ and $v = w_n$. The definition of reachability in timestamped graphs (specifically, Definitions 1 and 2), when limited to instantaneous events, states that $w_0@t \longrightarrow w_n$ holds in G if and only if there is a path

$$(w_0, w_1, t_1), (w_1, w_2, t_2), \ldots, (w_{n-1}, w_n, t_n)$$

in G such that $t_{i-1} \leq t_i$ for $1 \leq i \leq n$. For each timestamped edge (w_{i-1}, w_i, t_i), *BuildVer* adds a (standard) edge $(w_{i-1}^{<t_i}, w_i^{t_i})$ to $\mathbf{G}$. In addition, by Observation 6, there is a path from $w_i^{t_i}$ to $w_i^{<t_{i+1}}$. Putting these edges and paths together, we can construct a path in $\mathbf{G}$ from $w_0^{<t_0}$ to $w_n^{t_n}$. Also, by Observation 6, there is a path from $w_n^{t_n}$ to $w_n^{<\infty}$. Putting all these pieces together, we have a path from $w_0^{<t_0} = u^{<t_0}$ to $w_n^{<\infty} = v^{<\infty}$. A path from $u^{<t_0}$ to $v^{<\infty}$ clearly implies a path from $u^{\leq t_0}$ to $v^{<\infty}$, thus satisfying the "only if" part of the forward reachability condition.

Note that the "only if" proof constructed a one-to-one correspondence between the paths in G and $\mathbf{G}$. This correspondence can be used to establish the "if" part of the forward reachability condition as well.

The proof of the backward reachability condition follows the same steps as the proof of forward reachability, so we omit the details. ∎

4.2 Optimized Versioning and FD Preservation

Naive versioning is simple but offers no benefits in terms of data reduction. In fact, it increases storage requirements. In this section, we introduce several optimizations that reduce the number of versions and edges. These optimizations cause node timestamps to expand to an interval. A node v with timestamp interval $[t, s]$ will be denoted $v^{t,s}$.

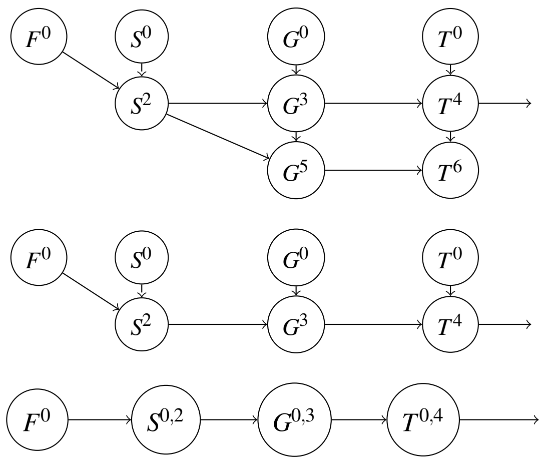

Fig. 7: The naive versioned graph from Fig. 6 (top), and the result of applying redundant edge optimization (REO) (middle) and then redundant node optimization (RNO) (bottom) to it. When adding the edge $(S, G, 5)$, we find that there is already an edge from the latest version S^2 of S to G, so we skip this edge. For the same reason, the edge $(G, T, 6)$ can be skipped, and this results in the graph shown in the middle. For the bottom graph, note that when adding the edge $(F, S, 2)$, S has no descendants, so we simply update S^0 by $S^{0,2}$, and avoid the generation of a new version. For the same reason, we can update G^0 and T^0 as well, resulting in the graph at the bottom.

4.2.1 *Redundant edge optimization (REO)*

Before adding a new edge between u and v, we check if there is already an edge from the latest version of u to some version of v. In this case, the new edge is redundant: in particular, reachability is unaffected by the addition of the edge, so we discard the edge. This also means that no new version of v is generated. Specifically, consider the addition of an event (u, v, t) to the graph. Let $u^{r,s}$ be the latest version of u. We check if there is already an edge from $u^{r,s}$ to an existing version of v. If so, we simply discard this event. We leave the node timestamp unchanged. Thus, for a node $u^{r,s} \in \mathbf{G}$, r represents the timestamp of the first edge coming into this node, while s represents the timestamp of the last. Alternatively, r denotes the start time of this version, while s denotes the last time it acquired a new incoming edge (i.e., an edge that wasn't eliminated by a reduction operation). Fig. 7 illustrates redundant edge (REO) optimization.

4.2.2 *Global Redundant Edge Optimization (REO*)*

With REO, we check whether there is already a *direct edge* from u to v before deciding to add a new edge. With global redundant edge, we generalize to check whether u is an *ancestor* of v. Specifically, before adding an event (u, v, t) to the graph, we check whether the latest version of u is already an ancestor of the latest version of v. If so, we simply discard the event.

The condition in REO* optimization is more expensive to check: it may take time linear in the size of the graph. Also, it did not lead to any significant improvement over REO in our experiments, so we did not evaluate it in detail. However, it is of conceptual significance because the resulting graph is optimal with respect to FD, i.e., any further reduction would violate FD-preservation.

4.2.3 *Redundant node optimization (RNO)*

The goal of this optimization is to avoid generating additional versions if they aren't necessary for preserving dependence. We create a new version v^s of a vertex because, in general, the descendants of v^s could be different from those of v^l, the latest version of v so far. If we overzealously combine v^l and v^s, then a false dependency will be introduced, e.g., a descendant of v^l may backtrack to a node that is an ancestor of v^s but not v^l. This possibility exists as long as (a) the ancestors of v^l and v^s aren't identical, and (b) v^l has non-zero number of descendants. We already considered (a) in designing REO optimizations described above, so we consider (b) here. Note that RNO needs to be checked only on edges that aren't eliminated by REO (or REO*).

Specifically, let $v^{r,s}$ be the latest version of v so far. Before creating a new version of v due to an event at time t, we check whether $v^{r,s}$ has any outgoing edge (i.e., any descendants). If not, we replace $v^{r,s}$ with $v^{r,t}$, instead of creating a new version of v. Fig. 7 illustrates the result of applying this optimization.

RNO preserves dependence for descendants of v, but it can change backward reachability of the node v itself. For instance, consider the addition of an edge at time t from $u^{p,q}$ to $v^{r,s}$. This edge is being added because it is not redundant, i.e., a backward search from $v@s$ does not reach $u^{p,q}$. However, when we add the new edge and update the timestamp to $v^{r,t}$, there is now a backward path from $v@s$ to $u^{p,q}$. The simplest solution is to retain the edge timestamp on edges added with RNO, and use them to prune out false dependencies.[6]

4.2.4 *Cycle-Collapsing Optimization (CCO)*

Occasionally, cyclic dependencies are observed, e.g., a process that writes to and reads from the same file, or two processes that have bidirectional communication. As observed by previous researchers [25, 24], such dependencies can lead to an explosion in the number of versions. The typical approach is to detect cycles, and treat the nodes involved as an equivalence class. A simple way to implement this approach is as follows. Before adding an edge from a version u^r to v^s, we check if there is a cycle involving u and v. If so, we simply discard the edge. Our experimental results show that cycle detection has a dramatic effect on some data sets.

Cycle detection can take time linear in the size of the graph. Since the dependence graph is very large, it is expensive to run full cycle detection before the addition of each edge. Instead, our implementation only checks for cycles involving two entities. We found that this was enough to address most sources of version explosion. An alternative would be to search for larger cycles when a spurt in version creation is observed.

[6]Note that these timestamps need to be used only when an edge added with RNO is the first hop in a backward traversal. If a node v subject to RNO gets a child x, this child would have been added after the end timestamp of v. So, when we do a backward traversal from x, all parents of v should in fact be backward reachable.

4.2.5 *Effectiveness of FD-optimizations*

REO and RNO optimizations avoid new versions in most common scenarios that lead to an explosion of versions with naive versioning:

- *Output files:* Typically, these files are written by a single subject, and not read until the writes are completed. Since all the write operations are performed by one subject, REO avoids creating multiple versions. In addition, all the write operations are combined.
- *Log files:* Typically, log files are written by multiple subjects, but are rarely read, and hence by RNO, no new versions need to be created.
- *Pipes:* Pipes are typically written by one subject and read by another. Since the set of writers does not change, a single version is sufficient, as a result of REO. Moreover, all the writes on the pipe can be combined into one operation, and so can all the reads.

We found that most savings were obtained by REO, RNO, and CCO. As mentioned above, REO* is more significantly more expensive than REO and provided little additional benefit. Another undesirable aspect of REO* (as well as the SD optimization) is that it may change the paths generated during a backward or forward analysis. Such changes have the potential to make attack interpretation more difficult. In contrast, REO, RNO and CCO preserve all cycle-free paths.

4.2.6 *Correctness and Optimality*

Theorem 8 *BuildVer, together with RNO and REO* optimizations, preserves full dependence (FD).*

Proof: We already showed that *BuildVer* preserves forward and backward reachability between the timestamped graph G and the naive versioned graph $\mathbf{G}$. Hence it suffices to show that the edges and nodes eliminated by REO* and RNO don't change forward and backward reachability in $\mathbf{G}$. Now, REO* optimization drops an edge (u,v,t) only if there is already an edge from the latest version of u to the latest or a previous version of v in $\mathbf{G}$. In other words, no new ancestors will result from adding this edge. Since no new ancestors are added, by definition of FD, any additional paths created in the original graph due to the addition of this edge do not have to be preserved. Thus REO* optimization satisfies the forward reachability condition of FD. Moreover, since this edge does not add new ancestors to v, it won't change backward reachability of any node from v or its descendants. Thus, the backward reachability preservation condition of FD is also satisfied.

Regarding RNO optimization, note that it is applied only when a node v has no descendants. In such a case, preservation of backward and forward reachability from v's descendants holds vacuously. ■

Optimality with respect to FD. We now show that the combination of REO* and RNO optimizations results in reductions that are optimal with respect to FD preservation. This means that any algorithm that drops versions or edges retained by this combination does not preserve full dependence. In contrast, this combination preserves FD.

The main reasoning behind optimality is that REO* creates a new version of an entity v whenever it acquires a new dependency from another entity u. In particular, REO* adds an edge from (the latest version of) u to (the latest version of) v *only when* there is no existing path between them. In other words, this edge corresponds to a time instance when v acquires a new ancestor u. For this reason, reachability from u to v needs to be captured at this time instance for FD preservation. Thus, an algorithm that omits this edge would not preserve FD. On the other hand, if we create an edge but not a new version of v, then there will be a single instance of v in the versioned graph that represents two distinct dependencies. In particular, there will be a path from u^t to v^s, the version of v that existed before the time t of the current event. As a result, u^t would incorrectly be included in a backward analysis result starting at the descendants of v^s. The only way to avoid this error is if v^s had no descendants, the condition specified in RNO. Thus, if either REO* or RNO optimizations were violated, then, forensic analysis of the versioned graph will yield incorrect results.

4.3 Source Dependence Preservation

In this section, we show how to realize source-dependence preserving reduction. Recall that a source is an entity that has no incoming edges. With this definition, sources consist primarily of pre-existing files and network endpoints; subjects (processes) are created by parents and hence are not sources, except for the very first subject. While this is the default definition, broader definitions of source can easily be used, if an analyst considers other nodes to be possible sources of compromise.

We use a direct approach to construct a versioned graph that preserves SD. Specifically, for each node v, we maintain a set $Src(v)$ of source entities that v depends on. This set is initialized to $\{v\}$ for source nodes. Before adding an event (u, v, t) to the graph, we check whether $Src(u) \subseteq Src(v)$. If so, all sources that can reach u are already backward reachable sources of v, so the event can simply be discarded. Otherwise, we add the edge, and update $Src(v)$ to include all elements of $Src(u)$.

Although the sets $Src(v)$ can get large, note that they need to be maintained only for active subjects and objects. For example, the source set for a process is discarded when it exits. Similarly, the source set for a network connection can be discarded when it is closed.

To save space, we can limit the size of Src. When the size limit is exceeded for a node v, we treat v as having an unknown set of additional ancestors beyond $Src(v)$. This ensures soundness, i.e., that our reduction never drops an edge that can add a new source dependence. However, size limits can cause some optimizations to be missed. In order to minimize the impact of such misses, we first apply REO, RNO and CCO optimizations, and skip the edges and/or versions skipped by these optimizations. Only when they determine an edge to be new, we apply the SD check based on Src sets.

Theorem 9 *BuildVer, together with redundant edge and redundant node optimizations and the source dependence optimization, preserves source dependence.*

Proof: Since full dependence preservation implies source dependence preservation, it is clear that redundant edge and redundant node optimizations preserve source dependence, so we only need to consider the effects of source dependence optimization. The proof is by induction on the number of iterations of the loop that processes events. The induction hypothesis is that, after k iterations, (a) $Src(v)$ contains exactly the source nodes that are ancestors of v, and (b) that SD has been preserved so far. Now, in the induction step, note that the algorithm will either add an edge (u, v) and update $Src(v)$ to include all of $Src(u)$, or, discard the event because $Src(v)$ already contains all elements of $Src(u)$. In either case, we can show from induction hypothesis that $Src(v)$ correctly captures all source nodes backward reachable from v. It is also clear that that when the edge is discarded by the SD algorithm, it is because the edge does not change the sources that are backward reachable, and hence it is safe to drop the edge. ■

Optimality of SD algorithm. Note that when SD adds an edge (u, v), that is because $Src(u)$ includes at least one source that is not in $Src(v)$. Clearly, if we fail to add this edge, then source dependence of v is no longer preserved. This implies that the above algorithm for SD preservation is optimal.

5 Compact Representations

In this section, we describe how to use the techniques described so far, together with others, to achieve highly compact log file and main-memory dependence graph representations.

5.1 Compact Representation of Reduced Logs

After reduction, logs can be stored in their original format, e.g., Linux audit records. However, these formats aren't space-efficient, so we developed a simple yet compact format called CSR. CSR stands for Common Semantic Representation, signifying that a unified format is used for representing audit data from multiple OSes, such as Linux and Windows. Translators can easily be developed to translate CSR to standard log formats, so that standard log analyzers, or simple tools such as grep, can be used.

In CSR, all subjects and objects are referenced using a numeric index. Complex data values that get used repeatedly, such as file names, are also turned into indices. A CSR file begins with a table that maps strings to indices. Following this table is a sequence of operations, each of which correspond to the definition of an object (e.g., a file, network connection, etc.) or a forensic-relevant operation such as open, read, write, chmod, fork, execve, etc. Operations deemed redundant by REO, REO* and CCO can be omitted.

Each operation record consist of abbreviated operation name, arguments (mostly numeric indices or integers), and a timestamp. All this data is represented in ASCII format for simplicity. Standard file compression can be applied on top of this format to obtain further significant size reduction, but this is orthogonal to our work.

5.2 Compact Main Memory Representation

Forensic analysis requires queries over the dependence graph, e.g., finding shortest path(s) to the entry node of an attack, or a depth-first search to identify impacted nodes. The graph contains roughly the same information that might be found in Linux audit logs. In particular, the graph captures information pertaining to most significant system calls. Key argument values are stored (e.g., command lines for execve, file names, and permissions), while the rest are ignored (e.g., the contents of buffers in read and write operations).

Nodes in the dependence graph correspond to subjects and objects. Nodes are connected by *bidirectional* edges corresponding to events (typically, system calls). To obtain a compact representation, subjects, objects, and most importantly edges must be compactly encoded. Edges typically outnumber nodes by one to two orders of magnitude, so compactness of edges is paramount.

The starting point for our compact memory representation is the SLEUTH [10] system for forensic analysis and attack visualization. The graph structure used in this paper builds on some of the ideas from SLEUTH, such as the use of compact identifiers for referencing nodes and node attributes. However, we did away with many other aspects of that implementation, such as the (over-)reliance on compact, variable length encoding for events, based on techniques drawn from data compression and encoding. These techniques increased complexity and reduced runtime performance. Instead, we rely primarily on versioned graphs and the optimizations in Section 4 to achieve compactness. This approach also helped improve performance, as we can achieve graph construction rates about three times faster than SLEUTH's. Specifically, the main techniques we rely on to reduce memory use in this paper are:

- *Edge reductions:* The biggest source of compaction is the redundant edge optimization. Savings are also achieved because we don't need timestamps on most edges. Instead, timestamps are moved to nodes (subject or object versions). This enables most stored edges to use just 6 bytes in our implementation, encoding an event name and about a 40-bit subject or object identifier.
- *Node reductions:* The second biggest source of compaction is node reduction, achieved using RNO and CCO optimizations. In addition, our design divides nodes into two types: base versions and subsequent versions. Base versions include attributes such as name, owner, command line, etc. New base versions are created only when these attributes change. Attribute values such as names and command lines tend to be reused across many nodes, so we encode them using compact ids. This enables a base version to be stored in 32 bytes or less.
- *Compact representation for versions:* Subsequent versions derived from base versions don't store node attributes, but just the starting and ending timestamps. By using relative timestamps and sticking to a 10ms timestamp granularity[7], we are able to represent a timestamp using 16-bits in most cases. This enables a version to fit within the same size as an edge, and hence it can be stored within the edge list of a base version. In particular, let S be the set of edges occurring between a version v and the next version appearing in the edge list. Then S is the set of edges incident on version v in the graph.

Edge lists are maintained as vectors that can grow dynamically for active nodes (i.e., running processes and open files) but are frozen at their current size for inactive nodes. This technique, together with the technique of storing versions within the edge list, reduces fragmentation significantly. As a result, we achieve a very compact representation that often takes just a few bytes per edge in the original data.

6 Experimental Evaluation

We begin this section by summarizing our implementation in Section 6.1. The data sets used in our evaluation are described in Section 6.2. In Section 6.3, we evaluate the effectiveness of FD and SD in reducing the number of events, and compare it with Xu et al.'s technique (LCD). We then evaluate the effect of these reductions on the CSR log size and the in-memory dependence graph in Sections 6.4 and 6.5. Runtimes for dependence graph construction and forensic analysis are discussed in Section 6.6. The impact of our optimizations on forensic analysis accuracy is evaluated in Section 6.7.

6.1 Implementation

Our implementation consists of three front-ends and a back-end written in C++. The front-ends together contain about 6KLoC; the back-end, about 7KLoC. The front-ends process data from audit sources. One front-end parses Linux audit logs, while the other two parse Linux and Windows data from the red team engagement. The back-end uses our *BuildVer* algorithm, together with (a) the REO, RNO, and CCO optimizations (Section 4.2) to realize FD preservation, and (b) the source dependence preservation technique described in Section 4.3. It uses the compact main-memory representation presented in Section 5.2. Our implementation can also generate event logs in our CSR format, described in Section 5.1.

[7]This is the granularity typically available on most of our data sets.

The back-end can also read data directly from CSR logs. We used this capability to carry out many of our experiments, because data in CSR format can be consumed much faster than data in Linux audit log format or the OS-neutral format in which red team engagement data was provided. A few key points about our implementation are:

- *Network connections:* We treat each distinct combination of (remote IP, port, time window) as a distinct source node. Currently, time windows are set to about 10 minutes. This means that when we read from any IP/port combination, all reads performed within a 10-minute period are treated as coming from a single source. Thus, FD and SD can aggregate them. After 10 minutes, it is considered a new source, thus allowing us to reason about remote sites whose behavior may change over time (e.g., the site may get compromised). A similar approach is applicable for physical devices.
- *Handling execve:* Execve causes the entire memory image of a process to be overwritten. This suggests that dependences acquired before the execve will be less of a factor in the behavior of the process, compared to dependences acquired after. We achieve this effect by limiting REO from traversing past execve edges.[8]
- *REO* optimization:* Almost all edges in our graph are between subjects and objects. Consider a case when a subject s reads an object o. The only case where o could be an ancestor but not a parent is if o was read by another subject s' that then wrote to an object o' that is being read by s. Since this relationship looks distant, we did not consider that REO* would be very useful in practice.[9]

6.2 Data Sets

Our evaluation uses data from live servers in a small laboratory, and from a red team evaluation led by a government agency. We describe these data sets below.

6.2.1 *Data from Red Team Engagement*

This data was collected as part of the 2nd adversarial engagement organized in the DARPA Transparent Computing program. Several teams were responsible for instrumenting OSes and collecting data, while our team (and others) performed attack detection and forensic analysis using this data. The red team carried out attack campaigns that extended over a period of about a week. The red team also generated benign background activity, such as web browsing, emailing, and editing files.

Linux Engagement Data (Linux Desktop). Linux data (Linux Desktop) captures activity on an Ubuntu desktop machine over two weeks. The principal data source was the built-in Linux auditing framework. The audit data was transformed into a OS-neutral format by another team and then given to us for analysis. The data includes all system calls considered important for forensic analysis, including open, close, clone, execve, read, write, chmod, rm, rename, and so on. Table 8 shows the total number of events in the data, along with a breakdown of important event types. Since reads and writes provide finer granularity information about dependencies than open/close, we omitted open/close from our analysis and do not include them in our figures.

Dataset	Total Events	Read	Write	Clone/ Exec	Other
Linux Desktop	72.6M	72.4%	26.2%	0.5%	0.9%
Windows Desktop	14.6M	77.1%	14.5%	1.2%	7.2%
SSH/File Server	14.4M	38.2%	58.3%	1.2%	2.3%
Web Server	2.8M	64.3%	30.3%	1.5%	3.9%
Mail Server	3M	70%	23.6%	1.7%	4.7%

Table 8: Data sets used in evaluation.

Windows Engagement Data (Windows Desktop). Windows data covers a period of about 8 days. The primary source of this data is Event Tracing for Windows (ETW). Events captured in this data set are similar to those captured on Linux. The data was provided to us in the same OS-neutral format as the Linux data. Nevertheless, some differences remained. For examples, network reads and network writes were omitted (but network connects and accepts were reported). Also reported were a few Windows-specific events, such as CreateRemoteThread. Registry events were mapped into file operations. From Table 8, it can be seen that the system call distribution is similar as for Linux, except for a much higher volume of "other" calls, due to higher numbers of renames and removes.

6.2.2 *Data From Laboratory Servers*

An important benefit of the red team data is that it was collected by teams with expertise in instrumenting and collecting data for forensic analysis. A downside is that some details of their audit system configurations are unknown to us. To compensate for this, we supplemented the engagement data sets with audit logs collected in our research lab. Audit data was collected on a production web server, mail server, and general purpose file and remote access server (SSH/File Server) used by a dozen users in a small academic research laboratory. All of these systems were running Ubuntu Linux. Audit data was collected over a period of one week using the Linux audit system, configured to record open, close, read, write, rename, link, unlink, chmod, etc.

6.3 Event Reduction: Comparison of LCD, FD and SD

Fig. 9 shows the event reduction factor (i.e., ratio of number of events before and after the reduction) achieved by our two techniques, FD and SD. For comparison, we reimplemented Xu et al.'s full-trackability reduction as described by Algorithms 1, 2 and 3 in [42]. As discussed before, full-trackability equivalence is like a *localized* version

[8]REO, and especially REO*, can be much more effective without this restriction, but such an approach also increases the risk of eliminating significant events from the graph.

[9]Moreover, because the in- and out-degrees of subjects are typically very large, a 3-hop search may end up examining a very large number of edges.

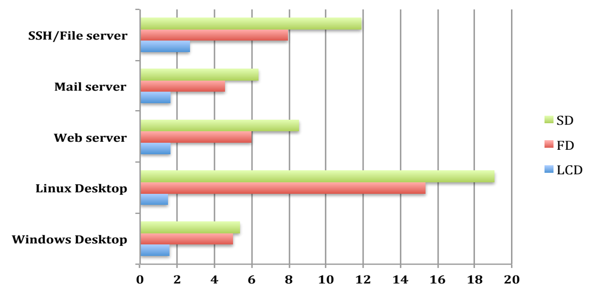

Fig. 9: Event reduction factors achieved by LCD, FD, and SD.

of our continuous dependence preservation criteria, and hence we refer to it as LCD for consistency of terminology. LCD, FD and SD achieve an average reduction factor of 1.8, 7 and 9.2 respectively. Across the data sets, LCD achieves reduction factors between 1.6 and 2.7, FD ranges from 4.6 to 15.4, and SD from 5.4 to 19.1.

As illustrated by these results, FD provides much more reduction than LCD. To understand the reason, consider a simple example of a process P that repeatedly reads file A and then writes file B. The sequence of P's operations may look like $read(A)$; $write(B)$; $read(A)$; $write(B)$; $\cdots$. Note that there is an outgoing (i.e., write) edge between every pair of incoming (i.e., read) edges into P. This violates Xu et al.'s condition for merging edges, and hence none of these edges can be merged. Our FD criteria, on the other hand, can utilize non-local information that shows that A has not changed during this time period, and hence can aggregate all of the reads as well as the writes.

We further analyzed the data to better understand the high reduction factors achieved by FD and SD. We found that on Linux, many applications open the same object multiple times. On average, a process opened the same object approximately two times on the laboratory servers. Since the objects typically did not change during the period, FD was typically able to combine the reads following distinct opens, thus explaining a factor of about 2. Next, we observed that on average, each open was accompanied by 3 to 5 reads/writes. Again, FD was able to aggregate most of them, thus explaining a further factor of 2 to 4. We see that the actual reduction achieved by FD is within this explainable range for the laboratory servers. For Windows desktop, the reduction factor was less, mainly because the Windows data does not include reads or writes on network data. For Linux desktop data set, FD reduction factor is significantly higher. This is partly because long-running processes (e.g., browsers) dominate in this data. Such processes typically acquire a new dependency when they make a new network connection, but subsequent operations don't add new dependencies, and hence most of them can be reduced.

Our implementation of SD is on top of FD: if an edge cannot be removed by FD, then the SD criterion is tried. This is why SD always has higher reduction factor than FD. SD provides noticeable additional benefits over FD.

Dataset	Size on Disk	CSR	Reduction factor	
			FD	SD
Linux Desktop	12.9GB	5.6×	66.1×	76.8×
Windows Desktop	2.1GB	2.4×	4.46×	4.54×
SSH/File server	6.7GB	15.1×	91.5×	122.5×
Web server	1.3GB	13.3×	49.3×	57.9×
Mail server	1.2GB	11.9×	41×	49.2×
Average (Geometric mean)		**8**×	**35.3**×	**41.4**×

Table 10: Log size on disk. The second column reports the log size of original audit data. Each remaining column reports the factor of decrease in CSR log size achieved by the indicated optimization, relative to the size on disk.

6.4 Log Size Reduction

Table 10 shows the effectiveness of our techniques in reducing the on-disk size of log data. The second column shows the size of the original data, i.e., Linux audit data for laboratory servers, and OS-neutral intermediate format for red team engagement data. The third column shows the reduction in size achieved by our CSR representation[10], before any reductions are applied. The next two columns show the size reductions achieved by CSR together with FD and SD respectively.

From the table, it can be seen that the reduction factors from FD and CD are somewhat less than that shown in Fig. 9. This is expected, because they compress only events, not nodes. Nevertheless, we see that the factors are fairly close, especially on the larger data sets. For instance, on the Linux desktop data, where FD produces about 15× reduction, the CSR log size shrinks by about 12× over base CSR size. Similarly, on SSH/File server, FD event reduction factor is 8×, and the CSR size reduction is about 6×. In addition, the log sizes are 35.3× to 41.4× smaller than the input audit logs.

6.5 Dependence Graph Size

Table 11 illustrates the effect of different optimizations on memory use. On the largest dataset (Linux desktop), our memory use with FD is remarkably low: less than two bytes per event in the original data. On the other two larger data sets (Windows desktop and SSH/file server), it increases to 3.3 to 6.8 bytes per event. The arithmetic and geometric means (across all the data sets) are both less than 5 bytes/event.

Examining the Linux desktop and Windows desktop numbers closely, we find that the memory use is closely correlated with the reduction factors in Fig. 9. In particular, for the Linux desktop, there are about 4.7M events left after FD reduction. Each event results in a forward and backward edge, each taking 6 bytes in our implementation (cf. Section 5). Subtracting this 4.7M*12B = 56.4MB from the 111MB, we see that the 1.1M nodes occupy about 55MB, or about 50 bytes per node. Recall that each node takes 32 bytes in our implementation, plus some additional space for storing file names, command lines, etc. A similar analysis of Windows

[10]Recall that CSR is uncompressed, so there is room for significant additional reduction in size, if the purpose is archival storage.

Dataset	Total No. of Nodes	Total Events	FD (MB)	SD (MB)
Linux Desktop	1.1M	72.6M	111	107
Windows Desktop	781K	10.3M	67	67
SSH/File Server	430K	14.4M	45	39
Web Server	141K	2.8M	16	15
Mail Server	189K	3M	21	20
Total	**2.64M**	**103.1M**	**260**	**248**

Table 11: Memory usage. The second column gives the total number of nodes in the dependence graph before any versioning. The third column gives the total number of events. The fourth and fifth columns give the total memory usages for FD and SD. Average memory use across these data sets is less than 5 bytes/event.

data shows that about 2M events are stored occupying about 24MB, and that the 781K nodes take up about 53B/node.

6.5.1 *Effectiveness of Version Reduction Optimizations*

Table 12 shows the number of node versions created with the naive versioning algorithm and our optimized algorithms. The second column shows that naive versioning leads to a version explosion, with about 26 versions per node. However, FD and SD drastically reduce the number versions: with FD, we create just about 1.3 versions per node, on average.

Table 13 breaks out the effects of optimizations individually. Since some optimizations require other optimizations, we show the four most meaningful combinations: (a) no optimizations, (b) all optimizations except redundant node (RNO), (c) all optimizations except cycle-collapsing (CCO), and (d) all optimizations. These figures were computed in the context of FD. When all optimizations other than RNO are enabled, the number of versions falls to about 3.6× from 25.6× (unoptimized). Enabling all optimizations except CCO leads to about 3 versions on average per node. Comparing these with the last column, we can conclude that RNO contributes about a 3× reduction and CCO a 2.4× reduction in the number of versions, with the remaining 2.8× coming from REO. It should be noted that REO and CCO both remove versions as well as edges, whereas RNO removes only nodes.

6.6 Runtime Performance

All results in our entire evaluation were obtained on a laptop with Intel Core i7 7500U running at 2.7GHz with 16GB RAM and 1TB SSD, running Ubuntu Linux. All experiments were run on a single core.

Dataset	Versions per node		
	Naive	FD	SD
Linux Desktop	68.65	1.05	1.02
Windows Desktop	13.9	1.37	1.35
SSH/File Server	34.36	1.31	1.06
Web Server	20.62	1.29	1.10
Mail Server	16.20	1.32	1.22
Average	**25.58**	**1.26**	**1.14**

Table 12: Impact of naive and optimized versioning. Geometric means are reported on the last row of the table.

Dataset	Versions per node			
	None	No RNO	No CCO	FD
Linux Desktop	68.65	4.56	17.75	1.05
Windows Desktop	13.9	2.60	1.38	1.37
SSH/File Server	34.36	4.32	2.21	1.31
Web Server	20.62	3.46	2.15	1.29
Mail Server	16.20	3.57	2.12	1.32
Average	**25.58**	**3.63**	**3.01**	**1.26**

Table 13: Effectiveness of different versioning optimizations. Geometric means are reported on the last row of the table.

6.6.1 *Dependence Graph Construction Time with FD*

With our FD-preserving optimizations, this time depends on (a) the size of cycles considered by CCO, and (b) the maximum number of edges examined by REO. For (a), we have not come across cycles involving more than two nodes that meaningfully increased the size or runtime. So, our current implementation only considers cycles of length two. To evaluate the effect of (b), we placed a limit k, called the *FD window size*, on the number of edges examined by REO before it reports that a dependence does not exist; this is safe but may reduce the benefit. With this limit in place, each edge is processed in at most $O(k)$ time, yielding a graph construction algorithm that is linear in the size of the input audit log.

Fig. 14 shows the dependence graph construction time as a function of FD window size. We use the notation $FD = c$ to represent the runtime when k is set to c. We use $k = 1$ as the base, and show the other runtimes relative to this base. Note that runtime can initially dip with increasing k because it leads to significant reductions in memory use, which translates into less pressure on the cache, and consequently, (slightly) improved runtime. But as k is increased beyond 100, the runtime begins to increase noticeably.

The runtime and the reduction factor both increase with window size. Fig. 15 plots the relationship between reduction factor and window size. In particular, FD=1 means that REO can eliminate the edge (u, v) only if the previous edge coming into v is also from u. The average reduction achieved by FD in this extreme case is 1.96, about the same as the maximum rate achieved by LCD. Another observation is that for the laboratory servers, with FD=25, we achieve almost the full reduction potential of FD. For the desktop systems used in the red team engagements, full potential is

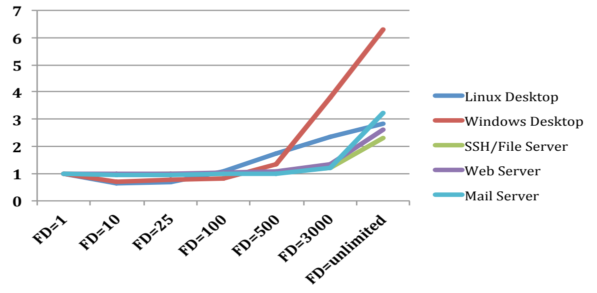

Fig. 14: Dependence graph construction time with different FD window sizes. Y-axis is the normalized runtime, relative to base of FD =1. These base times are 77.54s for Linux desktop, 19.02s for Windows desktop, 11.86s for Web server, 15.11s for Mail server and 41.77s for SSH/File server.

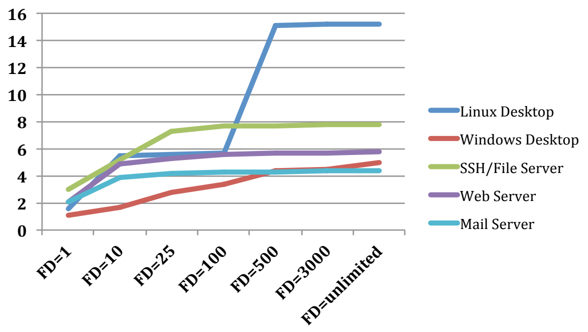

Fig. 15: Effect of FD window size on event reduction factor.

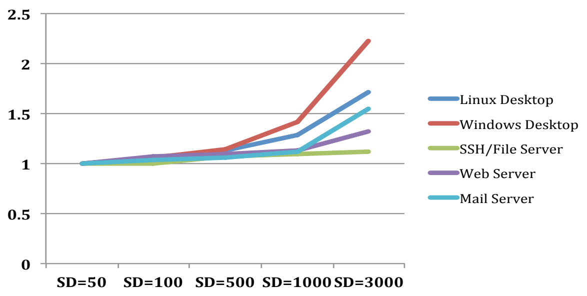

Fig. 16: Dependence graph construction time with different source set size limits. Y-axis is the runtime relative to the runtime with SD=50 (size limit of 50), which is 143.68s for Linux desktop, 23.59s for Windows desktop, 12.86s for Web server, 15.43s for Mail server and 42.81s for SSH/File server.

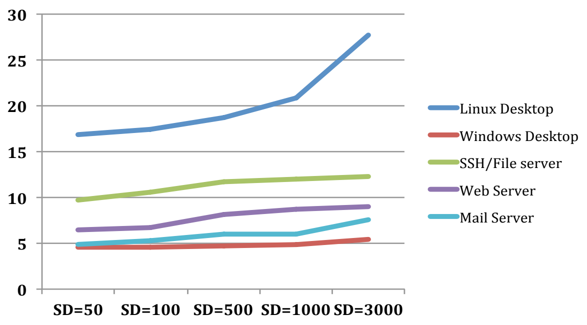

Fig. 17: Effect of source set size limit on event reduction factor.

achieved only at FD=500. We hypothesize that this is partly due to the nature of red team exercises, and partly due to workload differences between desktops and servers.

Comparing the two charts, we conclude that a range of FD=25 to FD=100 represents a good trade-off for a real-time detection and forensic analysis system such as SLEUTH [10], with most of the size reduction benefits realized, and with runtime almost the same as FD=1. At FD=25, our implementation processes the 72M records in the Linux Desktop data set in 84 seconds, corresponding to a rate of 860K events/second. For applications where log size is the primary concern, FD=500 would be a better choice.

6.6.2 *Dependence Graph Construction Time with SD*

For SD, the sizes of *Src* sets become the key factor influencing runtime. SD requires frequent computation of set unions, which takes linear time in the sizes of the sets. Moreover, increased memory use (due to large sets) significantly increases the pressure on the cache, leading to further performance degradation. We therefore studied the effect of placing limits on the maximum size of *Src* sets. Overflows past this limit are treated conservatively, as described in Section 4.3.

Figs. 16 and 17 show the effect of varying the source set size limit on the runtime and reduction factor, respectively. Recall that SD runs on top of FD, so the runtime of FD matters as well. However, since SD is significantly slower than FD, we did not limit the FD window size in these experiments. From the chart, the peak reduction factor is reached by SD=500 for all data sets except Linux desktop. The Linux desktop behaves differently, and we attribute this to the much higher level of activity on it, which means that a single long-running process can acquire a very large number of source dependencies. Nevertheless, the chart suggests that SD=500 is generally a good choice, as the overall runtime is almost unchanged from SD=50.

At SD=500, it takes 144 seconds to process 72M records from Linux, for an event processing rate of about 500K/second. Thus, although SD is slower than FD, it is quite fast in absolute terms, being able to process events at least two orders of magnitude faster than the maximum event production rate observed across all of our data sets.

6.6.3 *Backward and Forward Forensic Analysis*

Once the dependence graph is constructed, forensic analysis is very fast, because the whole graph currently resides in memory. To evaluate the performance, we randomly tagged 100K nodes in the dependence graph for the Linux desktop system. From each of these nodes, we performed

- a backward analysis to identify the source node closest to the tagged node. This search used a shortest path algorithm.
- a forward analysis to identify the nodes reachable from the tagged node. In case of searches that could return very large graphs, we terminated the search after finding 10K nodes (in most cases, the search terminated without hitting this limit).

This entire test suite took 112 seconds to run. In other words, each forward plus backward analysis on a dependence graph corresponding to 72M events took just 1.12 milliseconds on average.

6.7 Preserving Forensic Analysis Results

6.7.1 *Reproducing Analysis Results from* SLEUTH *[10]*

In our previous work [10], we performed real-time attack detection and forensic analysis of multi-step APT-style attack campaigns carried out in the 1st adversarial engagement in the DARPA Transparent Computing program. As described in Table 6 in [10], there were 8 distinct attack campaigns, each of which involved most of the seven stages in APT life cycle, including drop & load, intelligence gathering, backdoor insertion, privilege escalation, data exfiltration, and cleanup.

Dataset	Attack Scenario	Analysis Type	Number of Entities		
			Naive	FD	SD
Linux Desktop	A	Backward	7	7	7
		Forward	15	15	15
	B	Backward	3	3	3
		Forward	10	10	10
Windows Desktop	A	Backward	4	4	4
		Forward	17	17	17
	B	Backward	2	2	2
		Forward	9	9	9
	C	Backward	4	4	4
		Forward	7	7	7

Table 18: Results of forward and backward analyses carried out from the entry and exit points of attacks used in the red team attacks. The exact same set of entities were identified with and without the FD and SD event reductions.

SLEUTH assigns integrity and confidentiality tags to objects. These tags propagate as a result of read, write and execute operations. It detects attacks using tag-based policies that were developed in the context of our earlier work on whole-system integrity protection [19, 34, 35, 36] and policy-based defenses [39, 32]. It then uses a backward analysis to identify the entry point, and then a forward analysis to determine attack impact, and then a set of simplification passes to generate a graph depicting the attack, and to list the entities involved. Across these 8 attacks, a total of 176 entities were identified as relevant by the red team, and our original analysis in [10] identified 174 of them.

We carried out the investigation again, with FD and SD reductions in place. We were able to obtain the same results as in [10], showing that FD and SD reductions do not affect forensics results. This should come as no surprise, given that we proved that they both preserve the results of backward analysis followed by forward analysis. Nevertheless, the experimental results are reassuring.

6.7.2 Forensic Analysis Results on Table 8 Data Set

We then turned our attention to the Engagement 2 data set. (We did not use Engagement 1 data set in our reduction experiments because it was far smaller in size than Engagement 2.) There were 2 attacks within the Linux dataset and 3 attacks within the Windows data set. For each attack, we ran a forward analysis from the attack entry point, and then a backward analysis from attack exfiltration point (which is one of the last steps in these attacks). As shown Table 18, these analyses identified the exact same set of entities, regardless of whether any data reduction was used.

7 Related Work

Information-flow Tracking. Numerous systems construct dependence graphs [13, 9, 15, 22] or provenance graphs [25, 24, 8, 4, 29] that capture information flow at the coarse granularity of system calls. In particular, if a subject reads from a network source, then all subsequent writes by the subject are treated as (potentially) dependent on the network source. This leads to a *dependence explosion,* especially for long-running processes, as every output operation becomes dependent on every input operation. Fine-grained taint tracking [28, 41, 2, 12] can address this problem by accurately tracking the source of each output byte to a single input operation (or a few). Unfortunately, these techniques slow down programs by a factor of 2 to 10 or more. BEEP [17, 21] developed an alternative fine-grained tracking approach called *unit-based execution partitioning* that is much more efficient. However, as compared to taint-tracking techniques, execution partitioning generally requires some human assistance, and moreover, makes optimistic assumptions about the program behavior.

The main drawback shared by all fine-grained tracking approaches is the need for instrumenting applications. In enterprises that run hundreds of applications from multiple vendors, this instrumentation requirement is difficult to meet, and hence it is much more common for enterprises to rely on coarse-grained tracking.

Log Reduction. BackTracker [13, 14, 15] pioneered the approach of using system logs for forensic investigation of intrusions. Their focus was on demonstrating effectiveness of attack investigation, so they did not pursue log reduction beyond simple techniques such as omitting "low-control" (less important) events, such as changing a file's access time.

LogGC [18] proposed an interesting approach for log reduction based on the concept of garbage collection, i.e., removing operations involving removed files ("garbage"). Additional restrictions were imposed to ensure that files of interest in forensic analysis, such as malware downloads, aren't treated as garbage. They report remarkable log reduction with this approach, provided it is used in conjunction with their unit instrumentation. Without such fine-grained instrumentation, the savings they obtain are modest. To further evaluate the potential of this approach, we analyzed the data set used in this paper (Table 8). We found that less than 3% of the operations in this data set were on files that were subsequently removed. Although not all of these files satisfy their definition of "garbage," 3% is an upper bound on the savings achievable using this garbage collection technique on our data.

ProTracer [22] proposed another new reduction mechanism that was based on logging only the write operations. Read operations, as well as some memory-related operations tracked by their unit instrumentation, were not logged. In the presence of their unit instrumentation, they once again show a dramatic reduction in log sizes using their strategy. However, as discussed in the introduction, this strategy of selective logging of writes can actually increase log sizes in the absence of unit instrumentation. Indeed, our experiments with this strategy[11] resulted in more than an order of magnitude *increase* in log sizes.

[11] In our experiment, we implemented the method detailed in Table I of their paper [22]. Our implementation incorporated obvious optimizations such as avoiding the logging of multiple write records when the subject's taint set hasn't changed.

Xu et al.'s notion of full-trackability equivalence (LCD-preservation in our terminology) [42] is similar to our CD-preservation, as discussed in Section 3.2. We implemented their LCD-preserving reduction algorithm and found that our FD and SD optimizations achieve significantly more reduction, as detailed in Section 6.3. The reasons for this difference were also discussed in Section 6.3.

Provenance capture systems, starting from PASS [25], incorporate simple reduction techniques such as the removal of duplicate records. PASS also describes the problem of cyclic dependencies and their potential to generate a very large number of versions. They avoid cycles involving multiple processes by merging the nodes for those processes. Our cycle-collapsing optimization is based on a very similar idea.

ProvWalls [5] is targeted at systems that enforce Mandatory Access Control (MAC) policies. It leverages the confinement properties provided by the MAC policy to identify the subset of provenance data that can be safely omitted, leading to significant savings on such systems.

Winnower [38] learns compact automata-based behavioral models for hosts running similar workloads in a cluster. Only the subset of provenance records that deviate from the model need to be reported to a central monitoring node, thereby dramatically reducing the network bandwidth and storage space needed for intrusion detection across the cluster. These models contain sufficient detail for intrusion detection but not forensics. Therefore, Winnower also stores each host's full provenance graph locally at the host. In contrast, our system generates compact logs that preserve all the information needed for forensics.

File Versioning. The main challenge for file versioning systems is to control the number of versions, while the challenge for forensic analysis is to avoid false dependencies. Unfortunately, these goals conflict. Existing strategies that avoid false dependencies, e.g., creating a new version of a file on each write [33], generate too many versions. Strategies that significantly reduce the number of versions, e.g., open-close versioning [31],[12] can introduce false dependencies.

Many provenance capture systems use versioning as well. Like versioning file systems, they typically use either simple versioning that creates many versions (e.g., [4, 29]) or coarse-grained versioning that does not accurately preserve dependencies (e.g., [25]). In contrast, we presented an approach that provably preserves dependencies, while generating only a small number of versions in practice.

Provenance capture systems try to avoid cycles in the provenance graph, since cyclic provenance is meaningless. Causality-based versioning [24] discusses two techniques for cycle avoidance. The first of these performs global cycle detection across all objects and subjects on a system. The second operates with a view that is local to an object. It uses a technique similar to our redundant edge optimization, but is aimed at cycle avoidance rather than dependency preservation. They do not consider the other techniques we discuss in this paper, such as REO*, RNO, and SD preservation, nor do they establish optimality results.

[12] With this technique, the first open of an existing file for writing causes a new version to be generated. While the file remains open, subsequent opens all update the same version.

Graph Compression and Summarization. Several techniques have been proposed to compress data provenance graphs by sharing identical substructures and storing only the differences between similar substructures, e.g., [6, 40, 7]. Bao et al. [3] compress provenance trees for relational query results by optimizing the selection of query tree nodes where provenance information is stored. These compression techniques, which preserve every detail of the graph, are orthogonal to our techniques, which can drop or merge edges.

Graph summarization [27, 37] is intended mainly to facilitate understanding of large graphs but can also be regarded as lossy graph compression. However, these techniques are not applicable in our context because they do not preserve dependencies.

Attack Scenario Investigation. Several recent efforts have been aimed at recreating the full picture of a complex, multi-step attack campaign. HERCULE [30] uses community discovery techniques to correlate attack steps that may be dispersed across multiple logs. SLEUTH [10] assigns trustworthiness and confidentiality tags to objects, and its attack detection and reconstruction are both based on an analysis of how these tags propagate. PrioTracker [20] speeds up backward and forward analysis by prioritizing exploration of paths involving rare or suspicious events. RAIN [11] uses record-replay technology to support on-demand fine-grained information-flow tracking, which can assist in detailed reconstruction of low-level attack steps.

8 Conclusion

In this paper, we formalized the notion of dependency-preserving data reductions for audit data and developed efficient algorithms for dependency-preserving audit data reduction. Using global context available in a versioned graph, we are able to realize algorithms that are optimal with respect to our notions of dependency preservation. Our experimental results demonstrate the power and effectiveness of our techniques. Our reductions that preserve full dependence and source dependence reduce the number of events by factors of 7 and 9.2, respectively, on average in our experiments, compared to a factor 1.8 using an existing reduction algorithm [42]. Our experiments also confirm that our reductions preserve forensic analysis results.

References

[1] Paul Ammann, Sushil Jajodia, and Peng Liu. Recovery from malicious transactions. *IEEE Transactions on Knowledge and Data Engineering*, 2002.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick

McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 2014.

[3] Zhifeng Bao, Henning Köhler, Liwei Wang, Xiaofang Zhou, and Shazia Sadiq. Efficient provenance storage for relational queries. In *CIKM*, 2012.

[4] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. Trustworthy whole-system provenance for the Linux kernel. In *USENIX Security*, 2015.

[5] Adam Bates, Dave (Jing) Tian, Grant Hernandez, Thomas Moyer, Kevin R. B. Butler, and Trent Jaeger. Taming the costs of trustworthy provenance through policy reduction. *ACM Trans. Internet Technol.*, 2017.

[6] Adriane P. Chapman, H. V. Jagadish, and Prakash Ramanan. Efficient provenance storage. In *ACM SIGMOD*, 2008.

[7] Chen Chen, Harshal Tushar Lehri, Lay Kuan Loh, Anupam Alur, Limin Jia, Boon Thau Loo, and Wenchao Zhou. Distributed provenance compression. In *ACM SIGMOD*, 2017.

[8] Ashish Gehani and Dawood Tariq. Spade: support for provenance auditing in distributed environments. In *International Middleware Conference*, 2012.

[9] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The Taser intrusion recovery system. In *SOSP*, 2005.

[10] Md Nahid Hossain, Sadegh M Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R Sekar, Scott D Stoller, and VN Venkatakrishnan. Sleuth: real-time attack scenario reconstruction from cots audit data. In *USENIX Security*, 2017.

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

[12] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. *SIGPLAN Not.*, 2012.

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

[14] Samuel T. King and Peter M. Chen. Backtracking intrusions. *ACM Transactions on Computer Systems*, 2005.

[15] Samuel T. King, Zhuoqing Morley Mao, Dominic G. Lucchetti, and Peter M. Chen. Enriching intrusion alerts through multi-host causality. In *NDSS*, 2005.

[16] Srinivas Krishnan, Kevin Z. Snow, and Fabian Monrose. Trail of bytes: Efficient support for forensic analysis. In *ACM CCS*, 2010.

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

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

[19] Zhenkai Liang, Weiqing Sun, V. N. Venkatakrishnan, and R. Sekar. Alcatraz: An Isolated Environment for Experimenting with Untrusted Software. In *TISSEC*, 2009.

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

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

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

[23] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A Bader. A performance evaluation of open source graph databases. In *PPAA*, 2014.

[24] Kiran-Kumar Muniswamy-Reddy and David A Holland. Causality-based versioning. *ACM Transactions on Storage (TOS)*, 2009.

[25] Kiran-Kumar Muniswamy-Reddy, David A Holland, Uri Braun, and Margo I Seltzer. Provenance-aware storage systems. In *USENIX ATC*, 2006.

[26] Kiran-Kumar Muniswamy-Reddy, Charles P Wright, Andrew Himmer, and Erez Zadok. A versatile and user-oriented versioning file system. In *USENIX FAST*, 2004.

[27] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. Graph summarization with bounded error. In *ACM SIGMOD*, 2008.

[28] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[29] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. In *SoCC*, 2017.

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

[31] Douglas S Santry, Michael J Feeley, Norman C Hutchinson, Alistair C Veitch, Ross W Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. In *SOSP*, 1999.

[32] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. 2003.

[33] Craig A Soules, Garth R Goodson, John D Strunk, and Gregory R Ganger. Metadata efficiency in a comprehensive versioning file system. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 2002.

[34] Weiqing Sun, R. Sekar, Gaurav Poothia, and Tejas Karandikar. Practical Proactive Integrity Preservation: A Basis for Malware Defense. In *IEEE S&P*, 2008.

[35] Wai-Kit Sze and R Sekar. A portable user-level approach for system-wide integrity protection. In *ACSAC*, 2013.

[36] Wai Kit Sze and R Sekar. Provenance-based integrity protection for windows. In *ACSAC*, 2015.

[37] Yuanyuan Tian, Richard A. Hankins, and Jignesh M. Patel. Efficient aggregation for graph summarization. In *ACM SIGMOD*, 2008.

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

[39] V. N. Venkatakrishnan, Peri Ram, and R. Sekar. Empowering mobile code using expressive security policies. In *New Security Paradigms Workshop*, 2002.

[40] Yulai Xie, Dan Feng, Zhipeng Tan, Lei Chen, Kiran-Kumar Muniswamy-Reddy, Yan Li, and Darrell D.E. Long. A hybrid approach for efficient provenance storage. In *CIKM*, 2012.

[41] Wei Xu, Sandeep Bhatkar, and R. Sekar. Practical dynamic taint analysis for countering input validation attacks on web applications. Technical Report SECLAB-05-04, Department of Computer Science, Stony Brook University, May 2005.

[42] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. High fidelity data reduction for big data security dependency analyses. In *ACM CCS*, 2016.

[43] Ningning Zhu and Tzi-cker Chiueh. Design, implementation, and evaluation of repairable file service. In *Dependable Systems and Networks*, 2003.