LogGC: Garbage Collecting Audit Log

Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu
Department of Computer Science and CERIAS, Purdue University
West Lafayette, IN, 47907, USA
kyuhlee@purdue.edu, xyzhang@cs.purdue.edu, dxu@cs.purdue.edu

ABSTRACT

System-level audit logs capture the interactions between applications and the runtime environment. They are highly valuable for forensic analysis that aims to identify the root cause of an attack, which may occur long ago, or to determine the ramifications of an attack for recovery from it. A key challenge of audit log-based forensics in practice is the sheer size of the log files generated, which could grow at a rate of Gigabytes per day. In this paper, we propose LogGC, an audit logging system with garbage collection (GC) capability. We identify and overcome the unique challenges of garbage collection in the context of computer forensic analysis, which makes LogGC different from traditional memory GC techniques. We also develop techniques that instrument user applications at a small number of selected places to emit additional system events so that we can substantially reduce the false dependences between system events to improve GC effectiveness. Our results show that LogGC can reduce audit log size by 14 times for regular user systems and 37 times for server systems, without affecting the accuracy of forensic analysis.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Unauthorized access (e.g., hacking, phreaking) ; Invasive software(e.g., viruses, worms, Trojan horses)*; D.4.2 [**Operating System**]: Storage Management—*Garbage Collection*

Keywords

Attack Provenance; Audit Log; Garbage Collection; Reverse Engineering

1. INTRODUCTION

System-level audit logs record the interactions between applications and the underlying operating system (OS), such as file opens, reads and writes; socket reads and write; and process creations and terminations. Each of these is recorded as an event in the audit log, consisting of process/user/group id involved in the event, as well as the type and parameters of the event. In attack forensics, audit logs are critical to the construction of causal graphs. A causal graph shows the causality relations between system-level objects (e.g., files and sockets) and subjects (e.g., processes), *including those that existed in the past*. For instance, a process p_1 is causally dependent on another process p_2 if p_2 spawns p_1; a process p is causally dependent on a file f if p reads f; and a file f is dependent on p if f is created or modified by p. Causal graphs can be used to track the root cause of an attack. Upon observing a suspicious symptom (e.g., a zombie process) in the system, the administrator can use the casual graph to determine what had initially led to the presence of this process, which could be that a careless user viewed a phishing email, clicked an embedded URL, visited a malicious web site, triggered a driver-by download which compromised the system. Note that the initial attack may have happened days or even weeks before the symptom is observed. Hence the audit log and the derived causal graph are needed to disclose the attack path. Furthermore, the causal graph will disclose the damages or contaminations caused by the attack.

Recent work has focused on generating accurate, complete causal graphs from audit logs. Traditional causal graphs [13, 16] may suffer from imprecision caused by dependence explosion, where an event is unnecessarily dependent on too many other events and the corresponding causal graph is excessively large for human inspection. BEEP [20] involves program instrumentation to reduce the granularity of subjects from processes to "units" for fewer false-positive dependences. Other efforts propose using timestamps [12] or file offsets [23] to capture dependences more accurately.

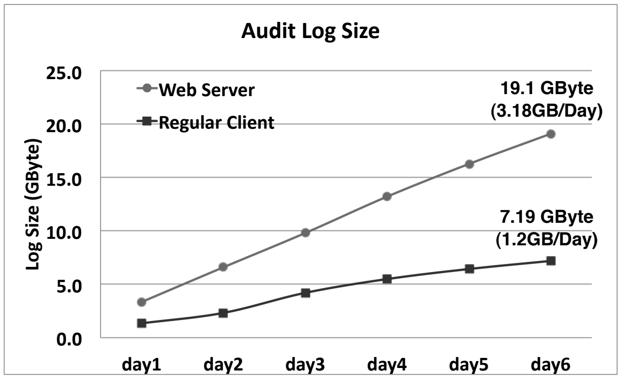


Figure 1: Audit log growth.

However, a major hindrance of audit log-based attack forensics in practice is the sheer size of audit logs. According to [16], even compressed audit logs grow at the rate of 1.2GB/day. Our own ear-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS'13, November 4–8, 2013, Berlin, Germany.
Copyright 2013 ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516731 .

lier work [20] shows an audit log growth rate of 800MB/day. Fig. 1 shows how audit log size grows over time in moderately loaded server and client machines, respectively in our experiments. We can see that the audit logs grow at an average rate of 3.18GB/day (server) and 1.2GB/day (client), incurring excessive space and processing overhead. Unfortunately, the reduction of audit log volume has not received sufficient research attention.

To fundamentally reduce audit log volume, we make a key observation: Many event entries in an audit log can be removed without affecting future forensic analysis. Such entries are about operations on system objects (e.g., files, sockets) that *neither influence nor are influenced by other processes or system objects*. We call such objects *unreachable objects*. We monitored a lab machine for a period of six days and observed that each day more than 94% of the objects accessed were destroyed (or terminated) and over 80% of those destroyed objects had very short lifetime, usually exclusively within a single process. These objects are likely to be unreachable, which suggests room for log reduction.

The log reduction problem we address shares some conceptual similarity to garbage collection (GC) in heap memory management. We hence call our solution LogGC . However, the two problems are technically different as memory GC is an instantaneous problem, namely, it focuses on removing unreachable heap memory objects in a *snapshot* of the memory. In contrast, we are trying to remove redundancy in audit logs that record *history* over a long period of time. We also face challenges that are caused by the unique requirements of forensic analysis and the coarse object granularity in audit logging.

The main contributions of LogGC are the following.

- We develop a basic GC algorithm that works directly on audit logs, each of which is a flat sequence of events. The algorithm can be invoked at any moment during system execution, taking the current audit log and generating a new and reduced audit log (Section 2).

- An important requirement in forensic analysis is to understand attack ramifications, which entails supporting *forward* causal analysis. The basic GC algorithm, which is an adaptation of a classic reachability-based memory GC algorithm, is incapable of supporting forward analysis. We hence propose a new extension to the algorithm (Section 3).

- To improve GC effectiveness, we leverage our previous technique BEEP [20] to partition a process to multiple *execution units*. We also propose a new technique to partition a data file into *logical data units*, by instrumenting user applications at a small set of code locations to emit additional system events. As such, better precision can be achieved and many more unreachable events can be exposed and garbage-collected (Section 4).

- We propose to leverage applications' own log files to further remove event entries in audit logs (Section 5).

- We conduct extensive evaluation of LogGC on a pool of real-world applications. Our results show that LogGC can garbage-collect 92.89% of the original audit logs for client systems and 97.35% for server systems. Furthermore, through a number of case studies, we show that the reduced logs are equally effective in forensic analysis.

Assumptions and Limitations of LogGC First, we trust the OS as LogGC collects, stores, and reduces audit logs at the kernel level. Hence, a kernel-level attack could disable LogGC and tamper with the audit log. However, LogGC can be implemented at the hypervisor level to mitigate such risks.

Second, we assume that, when LogGC initially starts, all user programs and files in the system are "clean". Attacks against these programs and files will hence be logged by LogGC . If LogGC begins with a compromised program, the program could disrupt LogGC by generating bogus system-level events. Note that while a program that gets compromised when LogGC is active may perform the same attack, the attack will be captured by LogGC . Our first two assumptions are standard for many existing system-level auditing techniques [12, 15, 16, 17, 20, 23].

Third, LogGC instruments user programs to partition executions and data files. While applicable to most application *binaries*, for some applications (e.g., `mysql`), our current file partitioning technique requires the user to inspect the application's *source code* using a profiler and select the instrumentation points from a small number of options provided by the profiler. The manual efforts are minor – only 15 statements were selected and instrumented for `mysql`. More importantly, this is a one-time effort. The user also needs to provide a small set of training inputs for LogGC to determine the instrumentation points. However, the instrumentation points are not sensitive to the inputs.

Finally, while LogGC can preprocess a large pool of commonly used applications, users may install new applications. If a user installs a new long running application (e.g. servers or UI programs), he/she may need to instrument the application for identification of execution units. If the user installs an application that induces a large number of dependences through files (i.e., by writing to a file and reading it later), such as a database application, he/she may also need to instrument the application for partitioning a file into data units. In the worst case where a newly installed application is not instrumented, LogGC can still garbage collect logs from the existing applications without affecting the un-reduced log entries generated from the new application.

2. BASIC DESIGN

Redundancy abounds in audit logs. Many applications create and operate on temporary files during execution. Such files are destroyed after the applications terminate. As a result, no future system behavior will be affected by these files. Keeping their provenance is hence unnecessary. An application may only read files or receive information from remote hosts, and send the contents to display without saving them. After the application terminates, its provenance is of no interest for future forensic analysis.

In this section, we present a basic algorithm that garbage-collects redundant entries in an audit log. It is analogous to garbage collection in memory management. We define *root objects* as the live processes and files at the time LogGC is invoked. We then traverse backward in the audit log. If any root object is directly or transitively dependent on a logged event, the event is marked reachable. In the end, all unreachable log entries are removed from the audit log. Different from classic GC, our algorithm will operate on the log file, which is a linear sequence of events, instead of a reference graph of memory cells. Moreover, the dependences between event entries are often not explicit. For example, assume a process receives a packet and saves it to a file. This leads to two event entries, one is a socket read and the other is a file write. It is not easy to infer the dependence between the two events from the audit log. A conservative approximation made by many existing audit logging techniques (also called provenance tracing) [12, 15, 16, 17, 20, 23] is to assume that an event is dependent on all the preceding input events for the same process.

Event type	Events
Input	file read; socket read
Output	file write (excluding `stdout`); socket write; process spawn; chmod; chown; link; truncate; create
Dead-end	write to `stdout`; file deletion*; process kill*

* denotes destruction events.

Table 1: Classification of event types

Before explaining the algorithm, we first classify logged events into three categories (Table 1) as the algorithm will behave differently based on type of events.

- An *input* event is one that receives data from input devices.
- An *output* event is one that creates influence on some other system object and such influence will persist beyond the completion of this event.
- A *dead-end* event is one that has effect only on objects directly involved in the event and the effect will *not* create dependences in subsequent execution of the system. For example, writes to `stdout` will not influence any system object.

Algorithm 1 Basic Audit Log Garbage Collection Algorithm

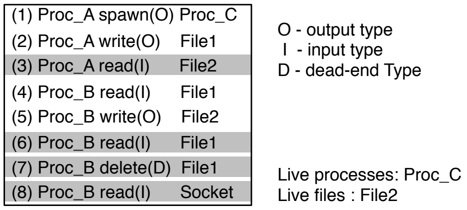
```
Input:      L - the audit log          P - the current live processes.
            F - the current live files.
Output:     L' - the new audit log.
Definition  ReachableProc - the set of reachable processes.
            ReachableObj - reachable system objects (e.g. files).

 1: $L' \leftarrow$ nil
 2: $ReachableProc \leftarrow P$
 3: $ReachableObj \leftarrow F$
 4: for each event $e \in L$ in reverse order do
 5:     if $e$ is an output event involving an object in $ReachableObj$ then
 6:         $L' \leftarrow e \cdot L'$
 7:         $ReachableProc \leftarrow ReachableProc \cup$ the process of $e$
 8:     else if $e$ is an input event involving a process in $ReachableProc$ then
 9:         $L' \leftarrow e \cdot L'$
10:         $ReachableObj \leftarrow ReachableObj \cup$ the objects operated by $e$
```

The basic algorithm is Algorithm 1. It takes the audit log, and the current set of live processes and live files as input, and produces a reduced audit log. It leverages two data structures *ReachableProc* and *ReachableObj* to maintain the set of reachable processes and system objects, respectively. We say a process, either alive or terminated, is reachable if a live process or file is directly or transitively dependent on it. Reachable system objects are similarly defined.

The algorithm first initializes *ReachableProc* with the set of live processes and *ReachableObj* with the set of live files. It then starts traversing the events in L in the reverse order. If an output event e in the log operates on a reachable object (line 5), the event becomes reachable and gets inserted to the output log L'. In the mean time, the process involved in the event is set to reachable. According to lines 8, 9, and 10, all the preceding input events in the process become reachable too. Note that setting the process reachable only affects the events preceding e, all the events that happen after e are not reachable even though they involve the same reachable process.

The algorithm is presented with a high level of abstraction. At the implementation level, LogGC also handles reuse of file ids and socket ids, and supports various system objects beyond files and sockets.

Example. Consider the example in Fig. 2. The live file and process are shown on the right bottom of the figure. The algorithm traverses backward. Log entries 8, 7, 6 are garbage-collected as they do not operate on any reachable objects or involve a reachable process.



```
(1) Proc_A spawn(O)  Proc_C
(2) Proc_A write(O)   File1
(3) Proc_A read(I)    File2
(4) Proc_B read(I)    File1
(5) Proc_B write(O)   File2
(6) Proc_B read(I)    File1
(7) Proc_B delete(D)  File1
(8) Proc_B read(I)    Socket
```

Figure 2: An example of the basic GC algorithm. An event entry consists of the process id, the event name (event type), and the system object being operated on.

Entry 5 is an output event with live file `File2` so it is inserted to the output log file. In the mean time, `Proc_B` becomes reachable. Hence, all the input events with `Proc_B` preceding entry 5 become reachable including the read of `File1` in entry 4. As a result, `File1` becomes a reachable object, which leads to entry 2 being inserted to the new log. Entry 1 is the creation of a live process and hence retained. It is worth mentioning that although entry 3 involves a reachable object `File2`, it is not an output event and hence garbage-collected.

Despite its simplicity, such a basic design is insufficient in practice. We will discuss how we handle various challenges in the following sections.

3. SUPPORTING FORWARD ANALYSIS

The basic design enables inspection of the history of any live object, which is a backward analysis. An equally important usage of audit logs is to facilitate understanding of attack ramifications, namely identification of damages that have been inflicted during an attack. We call this a *forward analysis* as it inspects the audit log in a forward fashion to look for ramifications of an event (e.g., the root attack event).

Fig. 3 (a) shows a causal graph generated from the audit log of the following hypothetical attack, with the numbers representing the corresponding events in the graph: (1) The user visited a malicious web site at "`x.x.x.x:80`"; `Firefox` is exploited such that (2) it spawns a backdoor process and (3) downloads a malicious dynamic library file `x.so`. (4) Later, the user launches the `ls` command in `bash`. (5) The `ls` process makes use of the malicious `x.so`. As a result, (6) while the process emits to the screen as usual, it also (7) removes an important file `.permission` and (8) the malicious library file `x.so`.

Assume that the user later notices the backdoor process, which is the only trail left by the attack (without analyzing the audit log). By performing backward analysis on the audit log, he can backtrack to the earlier visit to the malicious web site. However, to identify the damages caused by the attack, he also needs to perform forward analysis starting from the site visit event (1). As such, the past existence of the malicious library `x.so` and its removal of the important file and the library itself can be disclosed.

However, the basic design presented in Section 2 is insufficient for forward analysis because it garbage-collects all dead-end events since they will not affect any live system object in the future. However, *some dead-end events (e.g. file removal and process termination) become important when performing forward analysis.*

Revisiting the example in Fig. 3 (a), at the time of GC, only `bash` and the backdoor processes are alive. Files `x.so` and `.permission` have been removed from the file system. After applying Algorithm 1, only events 1 and 2 are left. While this is sufficient to disclose the causality of the backdoor, it loses information about the malicious library and the damages it caused. Note that `stdout`

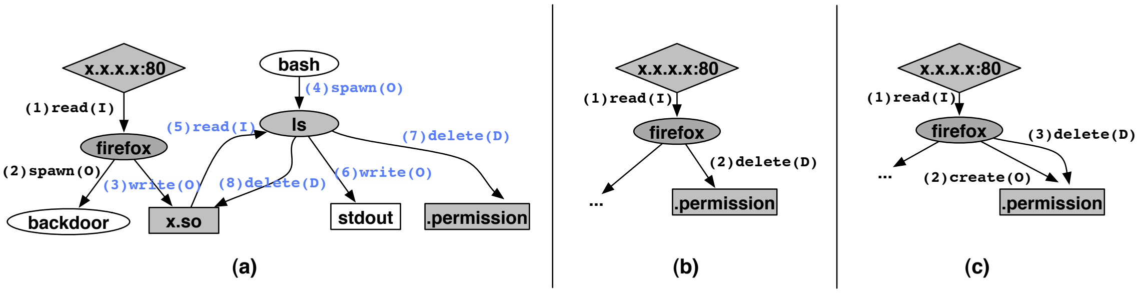


Figure 3: Causal graph examples. Ovals, diamonds and boxes represent processes, sockets and files, respectively. Edges denote events, annotated with numbers representing their order in the audit log, event names and types (i.e., *I* for input, *O* for output, and *D* for dead-end). The shaded shapes represent those no longer live at the GC time. Events in light blue denote those garbage-collected following Algorithm 1.

is a special file such that the "output-to-stdout" event at step (6) is considered a dead-end event (Table 1) and hence garbage-collected.

We cannot simply keep all dead-end events. Intuitively, keeping a dead-end event means that it may be of importance for forensic analysis. As such, events that may have led to the event must also be important. In other words, we cannot garbage-collect the input events preceding the dead-end event. Consider the example in Fig. 3 (b). It shows another hypothetical scenario in which (1) firefox visited a malicious web site and was exploited and (2) the infected firefox removed an important file .permission as part of the malicious payload. Suppose firefox has been terminated when GC is performed. If we keep the dead-end event 2, we need to keep the preceding input event 1 in order to understand its cause. However, if we were to keep all dead-end events and their preceding input events, garbage collection would not be effective at all.

One observation is that we do not need to consider all kinds of dead-end events as not all of them are of interest to forensic analysis. In particular, only those that cause destructive effects on the system are of interest, such as file removal and process termination. We call these events *destruction events*, such as the ones annotated with '*' in Table 1.

Log	Accessed Files	Deleted Files	
		Total	Temp Files
User1	14,909	11,981 (80.36%)	10,118 (84.45%)
User2	2,373	1,211 (51.03%)	1,197 (98.92%)
User3	2,991	2,046 (68.41%)	1,985 (97.02%)
User4	7,611	4,902 (64.41%)	2,610 (53.24%)
User5	2,988	1,416 (47.39%)	1,401 (98.94%)
Total	30,872	21,556 (69.82%)	17,311 (80.31%)

Table 2: Number of deleted files and temporary files

Unfortunately, considering destruction events only is still too expensive as many programs generate and delete a large number of files in their life time. Note that since a file deletion is a destruction event, it prevents us from garbage-collecting all the preceding input events. Fig. 3 (c) shows a typical execution pattern in firefox: it involves creating and removing a large number of files. These files are used to save/cache pages temporarily. They will be deleted when firefox is closed. In this scenario, if we retain all file deletions, there would not be many events that can be garbage-collected.

In Table 2, we have collected and analyzed five different audit logs from machines of different users with different settings (detailed settings in Section 6). Each log corresponds to one day's execution. The table shows the number of accessed files in each execution (column two) and the number of files that are deleted (column three). Observe that most of the accessed files are deleted. If we retain such deletions, the saving from garbage collection would be small.

Fortunately, another observation comes to the rescue: Most of the deleted files are temporary files. *A temporary file is defined as a file whose entire life time belongs exclusively to a single process.* In other words, a temporary file is only accessed (created/read/written/ deleted) by a single process. Besides web browsers, document viewer applications and compilers also use temporary files very often. As shown in the fourth column, 80.31% deleted files are temporary files. Such deletions have little forensic value and hence can be garbage-collected, which will transitively provide other garbage collection opportunities.

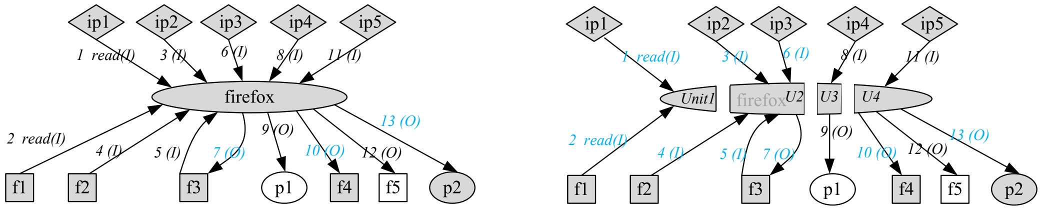
Hence, our final solution that supports forward attack analysis is as follows. *Upon a dead-end event, we check if it is a destruction event. If not, it will be garbage-collected. Otherwise, we will further determine if it is a file deletion event. If so, it will be retained and its process is set reachable, except when it is a temporary file deletion, which will be garbage-collected.* We perform a forward preprocessing to identify all temporary files before each GC procedure. When we retain a destruction event, we do not set the object being destroyed reachable, because the content of the object is not important but rather the deletion action itself is. As such, the preceding outputs to the object are still eligible for garbage collection.

One possible concern is that the attacker may exploit an application in such a way that the compromised application downloads a malicious library file, executes it and finally removes it. Since all these events occur within the same process, according to our policy, we consider the malicious library a temporary file and may garbage-collect its history. In this case, if the malicious library has ever affected or changed the system, such as changing other files or sending packets, its events will be marked as reachable by our traversal algorithm and thus retained. Otherwise, the event will never appear in any causal graph even though it is the deletion of a malicious library file. Therefore, it is safe to remove the event.

4. PROVENANCE TRACING WITH FINER GRANULARITY

In the basic design, attack provenance tracing is conducted at the system level, meaning that audit log entries are events captured by the OS. This is also the default setting of the Linux audit system. However, we observe that system-level tracing is overly coarse-grained, which substantially limits the effectiveness of audit log GC. In particular, the following two problems are dominant.

First, since we cannot afford tracing instruction level dependences in practice, we have to conservatively assume that an event is dependent on all preceding *input* events involving the same pro-



(a) before partition

(b) after partition

Figure 4: Partitioning a process execution into units. Event names may be omitted if inferable from context.

cess. Consequently, during garbage collection, if an event of a process becomes reachable, meaning that it may directly/transitively affect live processes or objects, the process becomes reachable, making all the preceding input events of the process reachable. It further implies that the objects operated in those events will also become reachable. This is particularly problematic for long running processes.

Consider a sample execution of `firefox` in Fig. 4. It accessed a number of URLs with different IP addresses, read and wrote a number of files, and spawned two processes. At the time of garbage collection, the process has terminated and many of the files accessed are also removed. The child process `p1` and file `f5` are still alive. Following the basic GC algorithm, live file `f5` makes `firefox` process reachable when the backward traversal reaches event 12, preventing garbage-collection of any input event before that. Consequently, only events 7, 10, and 13 are removed. In practice, audit logs are dominated by event entries from long running processes that may run for hours or even days.

Second, treating a file as a single object such that any read of the file is considered dependent on all preceding writes is too coarse-grained for some applications, especially those that need to repeatedly read/write to a file. For example, `mysql` creates an index file and a data file for each table. Upon receiving a query, `mysql` usually first reads the index file to find the location of a tuple (in the data file) given the tuple id. Then it reads the tuple from the data file and sends it back to the user. The index and data files need to be frequently updated. They also stay alive as long as the table is alive. According to our basic GC strategy, none of the reads/writes to these files can be garbage-collected.

Fig. 5 (a) shows three SQL queries executed by three `mysql` processes. In the first update query, `mysql` first reads the index file, which stores ids and locations of tuples in a B-Tree as shown in the figure. The corresponding table is shown in the data file. In this case, `mysql` first reads the root node in the index file which is the requested node (id=3) and finds the location of the tuple (event 1). Then it updates both the index file and the data file. In the second select query, `mysql` first reads the root node from the index file (event 4), which does not satisfy the `where` clause. It then further reads the child node in the index file (event 5), which satisfies the clause. Then `mysql` reads the data file to get the tuple (event 6) and returns it (event 7). In the third delete query, `mysql` reads individual tuples from the data file (events 8 and 10) and then checks them against the `where` condition. If the condition is satisfied, tuple deletion is performed by updating the index file (events 9 and 11).

According to Algorithm 1, since both the data and index files are alive, none of the writes to them can be garbage-collected, making the preceding reads non-garbage-collectable. Consequently, none of the events can be removed. Note that a simple idea that uses file offsets [23] to detect more precise dependences does not work here because `mysql` tends to read many input tuples to memory to process a query, but only a (possibly small) subset of them are used in computing the result tuples (e.g., the delete query in the previous example).

We have conducted an experiment in which we applied the basic GC algorithm to audit logs of `firefox` and `mysql`. Our results show that only 0.07% and 0.0004% of them can be garbage-collected, respectively. And the log entries from `mysql` can be as large as 90.68% of the overall audit log in a server system.

4.1 Dividing Process into Execution Units

To overcome the first problem discussed earlier in this section (i.e., treating a process as a single subject), we adopt our earlier solution called BEEP [20] by dividing a process into execution units. The basic idea is that the execution of a long running program is dominated by event handling loops. According to our study of over 100 applications [20], event handling loops are present in most of those applications and each loop iteration can be considered as a logical execution unit that handles an individual external request. Hence, instead of considering an event dependent on all preceding input events in the life time of an entire process, BEEP considers that it is only dependent on the preceding input events in the *same unit*.

Consider the example in Fig. 4 (b). The execution of `firefox` in (a) is divided into four units corresponding to four event handling loop iterations, each accessing an individual URL. As such, events 1-6 can be garbage-collected. This is because the reachable output events 12 and 9 only cause the input events in their units (i.e., `u4` and `u3`) to be marked as reachable. Note that the history of live objects, i.e., process `p1` and file `f5`, is correctly preserved.

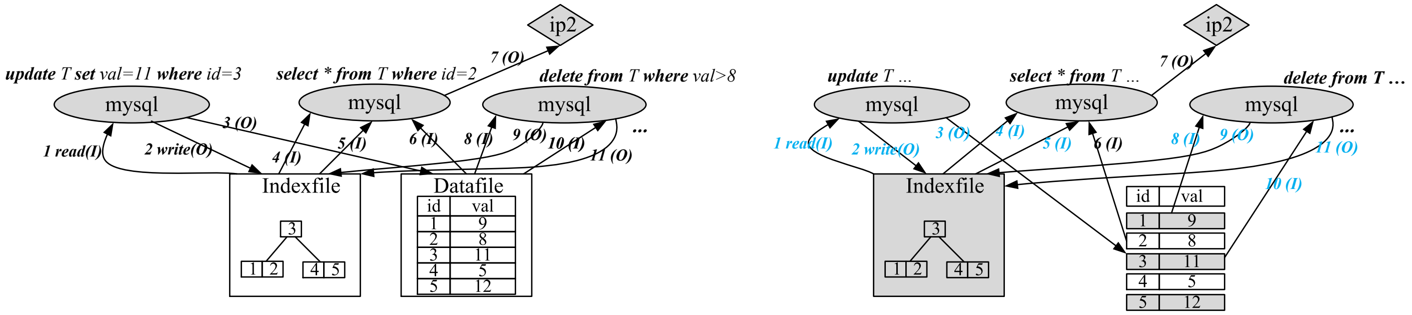
We adapt the binary profiling technique in BEEP to identify event handling loops in a binary program. Loop entries and exits are instrumented to emit special system events that serve as unit delimiters. We also identify workflows between units that may introduce cross-unit dependences. Details can be found in [20]. Since these are not our contribution, details are elided.

4.2 Dividing File into Data Units

To address the second problem (i.e., treating a file as a single object), we propose to divide a file into data units such that dependences can be defined with respect to the smaller data units.

Consider the example in Fig. 5 (b). The data file is divided into units with each unit being a tuple. Furthermore, since index files are completely internal to `mysql`, reads and writes on them are of no forensic interest and hence removed. We are now able to garbage-collect all events except 6 and 7. Observe that with the finer granularity, we can claim that tuple 3 is not reachable hence event 3 can be garbage-collected.

However, dividing files into data units is highly challenging. One may have to have knowledge about the internal structure of a file. Also, one has to identify all operations on data units. When an



(a) before partition

(b) after partition

Figure 5: Partitioning data files into units improves garbage collection.

operation is performed, the specific units being operated on need to be figured out. For instance, `mysql` has 1.2M lines of code. It is prohibitively difficult for a non-`mysql` developer to identify such information manually. Instruction-level tracing may determine data provenance precisely down to the byte level. However, it is too expensive to be useful for production systems.

Solution Overview. Our solution is to leverage a *profiler* to identify and instrument a small number of places in the application program such that special system events will be emitted by our instrumentation to disclose the provenance of data units. Take `mysql` as an example, given a query, the result tuples are computed from some input tuples that may come from multiple tables. We identify the code locations that *exclusively* access those input tuples used in computing the result tuples and instrument the code to emit the ids of the input tuples to the audit log. When an output tuple is written to disk, we also emit a write event tagged with its tuple id. Since all such events occur within the execution unit handling a single query, by following the default strategy that considers an output event (i.e., tuple write) as dependent on all preceding input events (i.e., input tuple reads), we can correctly determine the tuple-level dependences and allow more garbage collection.

Consider the example in Fig. 6 (a). It shows a piece of pseudo-code[1] that models the procedure that `mysql` handles the query on top of the figure, which performs a join query between tables `t1` and `t2` and writes the result table to `t3`. `Mysql` first loads all the tuples of `t1` and stores them to the cache. It then loads individual tuples of `t2` and compares their value fields with those in the cache. If there is a matching pair, the corresponding data fields are extracted (lines 12-14) and used to compose the result tuple. Finally, the result tuple is written at line 18.

With our solution, we will identify and instrument lines 12 and 13 for input tuples. Particularly, we will add a special system call after line 12 to emit the tuple id of `t1` and another one after line 13 to emit the tuple id of `t2`. According to the semantics of the code snippet, since lines 12 and 13 are within the true branch of the comparison at line 11, the tuples represented by the two lines must be the input tuples used to compute the output tuple. The result tuple composition at line 17 is also instrumented to emit the output tuple id. As such, we can associate the output tuple with the corresponding input tuples via the default audit log analysis.

In the following, we explain our techniques to identify those instrumentation points. Our discussion will be mainly focused on `mysql` as it is the core database engine for many web service applications.

Definitions: $Pv[\&x]$ - the provenance of a variable x, which is a set of input tuple ids.
pc: op (y_1, ..., y_n) - an instruction at pc that has op as the opcode and y_1,..., y_n the operand variables.
$S[pc]$ - aggregated provenance set for the instruction at pc.

Instruction	Action
id=$readRecord$("t",&buf,&s...)	for (i=0 to s-1) $Pv[buf+i]$={ id}
pc: x=op (y_1,...,y_n)	$Pv[\&x]$= $Pv[\&y_1] \cup ... \cup Pv[\&y_n]$ $S[pc]$=$S[pc] \cup Pv[\&x]$

Table 3: Instruction-level provenance profiler

Algorithm 2 Identifying instrumentation points for input tuples

```
execute mysql on a given query with the provenance profiler
Let "$pc$: $writeRecord(...,out\_buf,s...)$" be the instruction emitting the output tuple
$pv \leftarrow Pv[out\_buf] \cup ... \cup Pv[out\_buf+s-1]$
$R \leftarrow \{\}$
for each executed instruction $i$ do
    $l \leftarrow$ the program counter of $i$
    if $S[l] \subset pv$ and all tuples in $S[l]$ belong to one table then
        $R \leftarrow R \cup \{l\}$
return the minimum subset(s) of $R$ whose provenance covers $pv$
```

Detecting Output Tuple Writes. The first profiling technique is to identify instrumentation points that disclose writes of output tuples. For each query category (i.e. select, join, aggregation, update, and insert), we have a small set of training runs (2-5). If we observe a write to data file, we manually identify the tuple id in the output buffer and leverage a data dependence tracking technique to backtrack to the earliest point that defines the tuple id value. For example, if a tuple id is generated using a counter "`id=counter++`" at line 1, and then copied at lines 2, 3, and 4, which is the final tuple write, we backtrack to line 1. Essentially, we identify the root of the data dependence chain to the tuple id such that all instruction instances along the chain have the same id value.

Detecting Dependent Input Tuples. The second profiling technique is to identify instrumentation points that can disclose the input tuples that are used to compute individual output tuples. The basic idea is to leverage an *instruction level* provenance tracing technique that can precisely compute the set of ids of the input tuples that are directly or transitively used to compute a value at any program point. As such, given an output tuple, we can precisely know its input tuple provenance. Then we inspect the set of executed instructions to determine a minimal subset such that the union of their provenance sets is precisely the provenance of the output tuple. In other words, each of the instructions in the sub-

[1] The real code is too complex to present and explain with limited space.

insert into t3 (**select** t1.id, t2.id, t2.val **from** t1, t2 **where** t1.val = t2.val);

```
   /*read all t1 tuples and store them into cache */
1  while (...) {
2     id1=readRecord ("t1", &buf1, &s1, …);
3     store_cache(<"t1", buf1, s1>);
4  }
5  /*read each t2 tuple and check against the cached t1
6    tuples. If any pair matches, a new tuple is composed
7    and inserted to t3*/
8  while (…) {
9     id2=readRecord("t2", &buf2, &s2, …);
10    for (each record <"t1", buf1, s1> in cache)
11       if (getField(buf1, "t1.val")==getField(buf2,"t2.val") {
12          v1=getField(buf1, "t1.id");
13          v2=getField(buf2, "t2.id");
14          v3=getField(buf2, "t2.val");
15          buf3=newTuple();
16          insertField(buf3, id3++);
17          insertField(buf3, v1); …; insertField(buf3, v3);
18          writeRecord("t3", buf3, …);
         }
   }
```

(a) Code

t1

id	val
1	9
2	4

t2

id	val
5	9
6	26

	Trace	**Pv[]**	**S[]**
2_1	id1=readRecord ("t1",&buf1, …);	Pv[*a1*...(*a1*+7)]={1}	S[2]={1}
	…		
2_2	id1=readRecord ("t1",&buf1,…);	Pv[*a2*...(*a2*+7)]={2}	S[2]={1,2}
	…		
9_1	id2=readRecord ("t2",&buf2,…);	Pv[*b1*...(*b1*+7)]={5}	S[9]={5}
	…		
11_1	if (true)		
12_1	v1=getField(buf1, "t1.id");	Pv[&v1]=Pv[*a1*]={1}	S[12]={1}
13_1	v2=getField(buf2, "t2.id");	Pv[&v2]=Pv[*b1*]={5}	S[13]={5}
14_1	v3=getField(buf2, "t2.val");	Pv[&v3]=Pv[*b1*+4]={5}	S[14]={5}
16_1	insertField(buf3, id3++);		
17_1	insertField(buf3, v1-v3);	Pv[(*c1+4*)…(*c1*+11)]={1,5}	S[17]={1,5}
18_1	writeRecord("t3",buf3,…);		
	…		
9_2	id2=readRecord ("t2",&buf2,…);	Pv[*b2*...(*b2*+7)]={6}	S[9]={5,6}

(b) Execution with the Profiler

Figure 6: Using profiling to determine instrumentation points. The code snippet models how `mysql` **handles the query on top. Function** `readRecord`**("**`table_name`**",**`&buf, &s`**) reads a tuple from data file, returns it in** `buf` **that is allocated inside the function, the size is set in** `s` **and the tuple id is the return value;** `store_cache`**() stores a tuple to cache;** `getField`**() gets a tuple field from a buffer given the field label;** `insertField`**() inserts a field to a buffer;** `writeRecord`**() writes a tuple (buffer) to a given table. Symbols** $a1$**,** $a2$**,** $b1$**,** $b2$**, and** $c1$ **in (b) denote the different buffer address values. The subscripts in statement labels denote instances. Each data field is 4 byte long.**

set has processed (part of) the dependent input tuples. We hence can instrument these instructions to emit the ids of the tuples that they process. Note that instruction-level tracing is *only used for profiling*.

The semantic rules of the instruction-level provenance tracer are presented in Table 3, which specifies the action the tracer takes upon an instruction. Function `readRecord`() loads a tuple from data file. The result tuple is stored in `buf` with size `s`. The function returns the tuple id. When an invocation of the function is encountered, it is considered a source of input provenance. Therefore, the corresponding action is to set the provenance of each individual byte in `buf`, which stores the data fields of the tuple, to the input tuple id. This function is an abstraction of four functions in `mysql`, namely, `_mi_read_static_record`(), `_mi_read_rnd_static _record`(), `_mi_read_dynamic_record`(), and `_mi_read_rnd _dynamic_record`().

The last row shows the rule for assignments. We normalize the right-hand-side to an operation on a set of operands. The provenance of the left-hand-side is computed as the union of the operand provenance sets. We also compute the aggregated provenance set of the instruction, which is essentially the union of provenance sets of all the executed instances of the instruction.

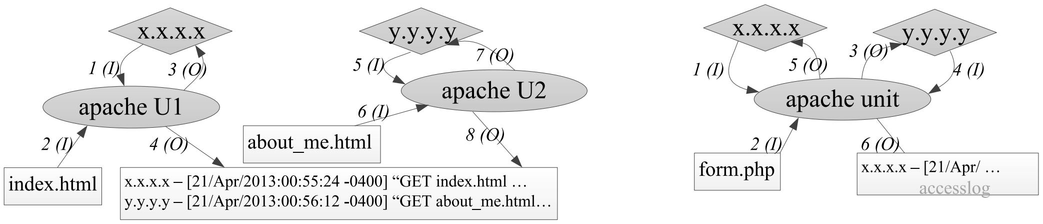
Algorithm 2 shows how to identify the instrumentation points, leveraging the instruction-level tracer. It first executes `mysql` with a given query. Lines 2 and 3 compute the provenance of the output, stored in *pv*. Set *R* denotes the candidate instrumentation points, which are denoted by program counters. In lines 5-8, the algorithm traverses all the executed instructions and determines the candidate instructions. The candidacy requires that the aggregated provenance of an instruction must be a subset of the output provenance. Otherwise, it must be an instruction that does not exclusively process dependent input tuples, such as line 2 in Fig. 6 (a), which processes all input tuples. Furthermore, the instruction should exclusively process input tuples from the same table. Otherwise, it is an instruction that aggregates information from multiple tables. It is usually impossible to acquire the original tuple ids at such an instruction. For example, line 17 in Fig. 6 (a) has exactly the output provenance. However, since it is just a composition of fields from input tuples, it is in general difficult to extract input tuple ids from the composed result. With the candidate set, at line 6, the algorithm identifies the minimal set of instructions that can cover the output provenance. It may return multiple such sets.

Fig. 6 (b) shows a sample profiling run. The input tables are shown on top. One can see that there will be just one result tuple, whose provenance is tuple 1 from `t1` and tuple 1 from `t2`. The figure shows part of the execution trace. The *Pv* and *S* sets are also shown for each executed statement. The program first loads tuples from `t1`, which leads to the provenance sets of the buffer to be set to {1} and {2} in the two respective instances of instruction 2. Then, the program reads `t2` and compares tuples. Since the first tuple of `t2` matches the first tuple of `t1`, lines 12-14 are executed to extract data from input tuples. Hence, their provenance includes the corresponding input tuple id. At the end, the aggregated provenance of line 2 is {1,2}, line 9 is {5,6}, line 12 is {1}, and lines 13 and 14 are {5}. Since the output provenance is {1,5}, the minimal cover set could be lines 12 and 13 or 12 and 14, which are the places reported by the algorithm.

Even though our discussion is driven by a specific query, the algorithm is generic. We apply it to a training set of various types of queries. Since the profiler can only give hints about the instrumentation points, we assume that human users will inspect the source code to eventually determine the instrumentation points from the reported ones. In practice, the human efforts are small and they are one-time efforts. We instrumented a total of 15 places in *mysql* after inspecting about 40 places suggested by the profiler. The instrumentation supports typical select, join, aggregation, update, insert and sub-queries with nesting level of 2. We have validated that it can correctly identify tuple-level dependences in the standard database workloads from RUBiS [5] and SysBench [1].

Discussion. In this work, we do not consider *control dependences* between tuples. For example, in the following query:

select * **from** t1 **where** t1.val > (**select** avg(val) **from** t1);



(a) Leveraging Apache AccessLog

(b) Apache Log Alone is Insufficient

Figure 7: Apache application log examples.

Our instrumentation will emit the ids of tuples whose values are greater than the average. However, all tuples in `t1` are correlated to the output tuples as they are used to compute the average, which is used in the `where` condition. It is analogous to the traditional concept of program control dependence. In this work, we deem such precise tuple control dependences of limited forensic value. Note that even though SQL injection attacks are often achieved by manipulating `where` conditions, knowing the tuples that are used in malicious `where` clauses are unlikely useful as they often serve only as a true condition enabling other malicious actions, which will be captured by LogGC .

If a query generates multiple output tuples, the output tuples will belong to the same execution unit. However, we cannot simply consider a tuple output event as dependent on all the preceding tuple input events in the same unit. Instead, we leverage the fact that output tuples are computed one after another. Hence a tuple output event is only dependent on those preceding tuple input events that are in between this output event and the immediate preceding output event.

Finally, since instrumentation points for data units are produced by the profiler and validated manually by the user, it is possible that they be incomplete and unsound (i.e., inducing false tuple dependences). Also, the profiler takes regular use cases provided by the user to determine the instrumentation points. Our experience does show that our techniques are sufficiently effective. Section 6.3 shows that LogGC with data units effectively reduces log size without affecting forensic analysis of realistic attack scenarios. The results show that the reduced logs (7 times smaller than the original ones) are equally or even more effective in forensic analysis that involves databases. In the future, we plan to develop more automated techniques to statically determine instrumentation points.

5. LEVERAGING APPLICATION LOGS

We observe that many long running applications have the following two characteristics: (1) They have their own log files that record some execution status (we call them the application logs to distinguish from audit logs); (2) They often send data to remote users through socket writes. For instance, `apache` records every remote request it receives and the status of fulfilling the request. It also sends the requested (html) file back to the remote client through socket writes. On one hand, the two characteristics make garbage collection difficult, because both application log file writes and socket writes are output events. According to our earlier discussion, any preceding input events within the same unit may have dependence with those output events hence they cannot be garbage-collected. On the other hand, the two properties also provide an opportunity to prune more events in the audit log if they are already present in the application logs.

Fig. 7 (a) shows two execution units of `apache` that handle requests for `index.html` and `about_me.html`, respectively. They both have the following event pattern: socket read to receive the request; file read to load the file; socket write to send the file; and file write to the application log to record the history. This is also the dominant event pattern for `apache`. Both socket write and file write prevent garbage-collecting any other events. According to our experience, such server programs can generate as much as 71.4% of audit log events.

To address this problem, we propose to leverage the application logs themselves. More specifically, we observe that the aforementioned event pattern (as shown in Fig. 7 (a)) forms a small isolated causal graph independent of the other events in the audit log. It simply receives the request and returns the file. It does not affect other objects in the system except the application log. Furthermore, all events in the graph can be inferred from the application log. In Fig. 7(a), all the four events can be inferred from the application log and hence removed from the audit log.

One important criterion for replacing audit log events with application log events is that the events are self-contained. They do not have dependence with any past event or induce any dependence with future events. For example, if `about_me.html` in Fig. 7 (a) has been modified in the past by some user request, which is recorded in the audit log, events in the second unit will not be garbage-collected. In case of a malicious modification, this will allow us to back-track to the malicious change and determine its ramifications. Furthermore, if a unit produces side effects on the system, such as writing to a file or writing to a database (e.g., via socket `y.y.y.y` to the `mysql` server in Fig. 7.(b)), its events will not be garbage collected.

We apply this strategy to server programs (e.g., web server and ftp server) and user applications that generate their own logs, with the exception of `mysql` because we rely on the tuple-level events to perform garbage collection.

6. EVALUATION

We have implemented a LogGC prototype composed of training, profiling, instrumentation, log analysis and garbage collection components. The training and profiling components are implemented on PIN [21], instrumentation is through a binary rewriting tool PEBIL [19]. The log analysis and garbage collection components are implemented in C++.

Table 4 shows a list of sample applications we have installed and preprocessed for our evaluation. For each application, we have trained and instrumented it to support process and file partitioning. Columns 3 and 4 indicate applications that make use of temporary files and have application logs, respectively. Recall that we garbage-collect temporary file deletions, which enables removing other dependent events. The last column indicates if an application needs to be instrumented for data unit partition. Currently, we only instrument `mysql` to achieve more log reduction.

	Logs	Total	Basic GC		Execution Unit		Temp file		Application Logs		Data Unit	
		# logs	# logs	(%)	# logs	(%)	# logs	(%)	# logs	(%)	# logs	(%)
One day execution	User1	1,159,680	745,424	64.28%	667,104	57.52%	308,144	26.57%	172,688	14.89%	-	-
	User2	985,185	984,261	99.91%	685,233	69.55%	602,715	61.18%	197,718	20.01%	108,172	10.98%
	User3	1,329,854	1,246,804	93.75%	618,572	46.51%	134,859	10.14%	83,595	6.29%	-	-
	User4	1,921,038	1,920,363	99.96%	247,409	12.88%	228,428	11.89%	61,100	3.18%	-	-
	User5	973,789	592,521	60.85%	300,824	30.89%	238,577	24.5%	1,403	0.14%	-	-
	Average	1,273,909	1,097,875	86.18%	503,828	39.55%	302,545	23.75%	108,470	8.51%	90,668	7.11%
Web server benchmark	Rubis1	8,676,521	8,676,260	100.00%	8,467,229	97.59%	8,355,045	96.29%	7,886,531	90.90%	259,139	2.99%
	Rubis2	8,433,427	8,433,345	100.00%	8,433,142	100.00%	8,323,800	98.70%	7,812,483	92.64%	166,968	1.98%
	Rubis3	8,352,924	8,352,842	100.00%	8,352,630	100.00%	8,243,034	98.68%	7,731,299	92.56%	206,786	2.48%
	Rubis4	8,900,179	8,897,798	99.97%	8,126,108	91.30%	8,011,739	90.02%	7,508,254	84.36%	326,705	3.67%
	Rubis5	7,933,792	7,933,710	100.00%	7,933,501	100.00%	7,829,530	98.69%	7,209,876	90.88%	161,523	2.04%
	Average	8,459,368	8,458,791	99.99%	8,262,522	97.67%	8,152,629	96.37%	7,629,688	90.19%	224,224	2.65%
Applications	Firefox	3,260,948	3,258,628	99.93%	1,518,640	46.57%	999,160	30.64%	371,336	11.39%	-	-
	MC	16,020	15,940	99.50%	2,380	14.86%	1,685	10.52%	5	0.03%	-	-
	Mplayer	452,400	452,338	99.99%	252,296	55.77%	0	0.00%	0	0%	-	-
	Pidgin	47,018	47,018	100%	9,475	20.15%	8,605	18.30%	231	0.49%	-	-
	Pine	91,215	91,215	100%	44,982	49.31%	41,769	45.79%	970	1.06%	-	-
	Proftpd	51,906	51,404	99.03%	20,044	38.62%	20,044	38.62%	3,156	6.08%	-	-
	Sendmail	7,321	7,176	98.02%	6,376	87.09%	4,560	62.29%	684	9.34%	-	-
	Sshd	148,885	148,885	100%	148,885	100%	148,885	100%	72	0.05%	-	-
	Vim	98,791	98,791	100%	98,142	99.34%	4,213	4.26%	4,191	4.24%	-	-
	W3m	128,569	127,330	99.04%	127,302	99.01%	80,934	62.95%	32,914	25.60%	-	-
	Wget	80112	80112	100%	40,344	50.36%	40,344	50.36%	40,344	50.36%	-	-
	Xpdf	76,015	76,015	100%	27,401	36.05%	0	0%	0	0%	-	-
	Yafc	477	477	100%	477	100%	477	100%	477	100%	-	-
	Audacious	216,428	216,427	100%	9,031	4.17%	867	0.4%	867	0.4%	-	-
	Bash	1,325	111	8.38%	109	8.23%	109	8.23%	109	8.23%	-	-
	Apache	3,951,923	3,951,523	99.99%	3,951,130	99.98%	3,951,130	99.98%	490,023	12.40%	-	-
	Mysqld	38,344,585	38,344,430	100%	37,363,478	97.44%	37,363,395	97.44%	36,865,575	96.14%	631,115	1.65%

Table 5: Number of log entries after garbage collection.

	Applications	Temp	Log	Data unit
Servers	Sshd-5.9	N	Y	N
	Sendmail-8.12.11	N	N	N
	Proftpd-1.3.4	N	Y	N
	Apache-2.2.21	N	Y	N
	Cherokee-1.2.1	N	Y	N
	Squid-3.2.6	Y	Y	N
	Gmediaserver-0.13.0	N	Y	N
	MySQL-5.1.66	Y	Y	Y
UI Programs	Wget-1.13	N	N	N
	W3m-0.5.2	Y	Y	N
	Pine-4.64	N	N	N
	MidnightCommand-4.6.1	Y	Y	N
	Vim-7.3	Y	Y	N
	Bash-4.2	N	Y	N
	Firefox-11	Y	Y	N
	Yafc-1.1.1	N	N	N
	Pidgin-2.10.6	Y	Y	N
	Xpdf-3.03	Y	N	N
	Mplayer-1.1	Y	N	N
	Audacious-2.5.4	Y	N	N

Table 4: Application description

6.1 Effectiveness

Regular User Systems. In the first experiment, we collect audit logs from machines of five different users (with the same system image). Each log is collected from one day's execution. The users have different usage patterns. User1 is a software developer who used `Vim` editor and compilers a lot. He also downloaded and installed several tools during the one-day experiment. User2 ran a web server and a public ftp server. User3 mostly used `Firefox` for web surfing and `Xpdf` to view PDF files. He also used the `Pidgin` chat client to communicate with friends. User4 watched a set of movies using `Mplayer` and also used `Audacious` to listen to music. User5 used the system in the console mode. He used text-based applications such as `W3m` for the Web and `pine` for emails.

We ran LogGC at the end of the one-day execution. The results are presented in Table 5. The third column shows the total number of log entries in each original audit log. Columns 4 and 5 present the reduced log size (and percentage) when we perform the basic GC algorithm (Section 2), except that we do not garbage collect any dead-end event, in order to support forward analysis (Section 3). The results in the first 5 rows show that the basic algorithm is not very effective. It can reduce 35%/40% for the logs of User1/User5 because they ran short-running applications hence false dependences, which would unnecessarily prevent garbage collection, are less likely.

Columns 6 and 7 show log reduction when we use execution partitioning for long running applications on top of the basic algorithm. In the first five rows, we can remove an average of 60.45% of the audit logs. In columns 8 and 9, we additionally garbage-collect temporary file deletions, which also enables collecting the events that they depend on (Section 3). LogGC reduces an average of 76.25% of the original log entries. In columns 10 and 11, we remove the redundant entries captured in application logs (Section 5). The results show that LogGC reduces 91.49% of the original logs. The last two columns show log reduction when we use data unit partitioning (Section 4.1). Only User2 used `mysql` as the backend for his web server. LogGC can reduce half of the remaining entries for User2. Finally, the average size of the remaining logs (of the five users) is only 7.11% of the original size, indicating an order of magnitude of reduction.

Server Systems. In the next experiment, we focus on evaluating LogGC on server system logs. We use RUBiS [5] which is an auction service similar to eBay. We setup the auction site using the default configuration with `apache` as frontend and `mysql` as backend. In our setup, both `apache` and `mySQL` are located in the same machine. After initialization, we have 9 tables in the database which contain 33,721 items for sale in 20 categories. This database is acquired from [5]. Then we use the client emulator that acts as users (buyers or sellers) of the bidding system. The client emulator uses 27 pre-defined transitions such as user registration, item registration, item browsing by category, bidding, buying and leaving comments during execution. We use all 5 different setups for client emulation provided by RUBiS. Each execution emulates 240 users performing 60,000 to 70,000 transitions and lasts 20 to 30 minutes. We acquire the audit log of the bidding server after each execution and apply LogGC .

The "Web server benchmark" rows in Table 5 show the results. We can observe that most GC strategies are not as effective as on user systems. With "execution partitioning", "temp file" and "application log" strategies, we can reduce only 10% of the logs (columns 10 and 11) even though both `apache` and `mysql` have independent execution units, temporary files, and application logs. This is because the executions heavily access the index and data files in the database such that most execution units become reachable through file-level dependences. Columns 12 and 13 show the reduced log size with data unit partitioning. Only 2.65% of the log entries need to be preserved on average, corresponding to a 37-time reduction.

Per Application Results. We also present the results for individual applications. They are aggregated from the aforementioned user system executions and server executions. We observe that the events from some applications can be completely garbage-collected such as `Mplayer` and `Xpdf`. `Mplayer` is a video player and it interacts with the screen and the sound card. But its execution does not affect system execution in the future. Similarly, `Xpdf` only displays PDF files on the screen. On the other hand, we cannot garbage collect any events from `Yafc`, an ftp client, because the user downloaded files using `Yafc` and kept all the files in the system. Similar explanation applies to `W3m` and `Wget`.

6.2 Performance

Table 6 shows the performance of LogGC. The experiments were performed on an Intel Core i7-3770 CPU with 4GB memory running Linux 2.6.35. The execution time of LogGC is mainly divided to two parts: the log parsing time and the GC time. The table shows that LogGC is reasonably efficient. It processes 3GB of logs in about 2 minutes, with parsing time being the dominant factor. Table 7 shows the runtime overhead incurred by data unit instrumentation, using two popular benchmarks: RUBiS and SysBench. The results show that the runtime overhead is very low.

Log	Parsing time (s)	GC time (s)
User1	35.97	3.76
User2	22.09	2.65
User3	26.71	2.82
User4	47.47	4.52
User5	23.65	3.17
Rubis1	104.88	10.62
Rubis2	101.21	9.45
Rubis3	101.09	9.48
Rubis4	102.89	10.13
Rubis5	98.21	9.16

Table 6: Garbage collection performance.

Benchmarks	Response time (ms)		Overhead (%)
	Without Instrumentation	With Instrumentation	
RUBiS	4,769.6	4,825.6	1.17%
SysBench	44.68	45.58	2.04%

Table 7: Runtime overhead for data unit instrumentation.

6.3 Attack Investigation

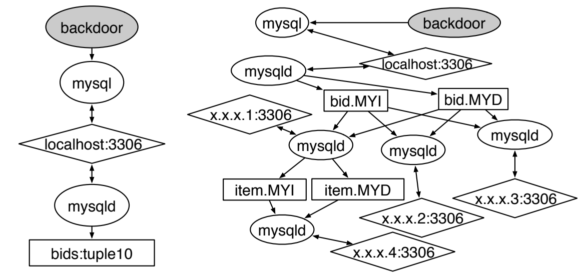
In this section, we show that the reduced audit logs are equally informative in forensic analysis through a number of case studies. We adopt eight attack scenarios previously used to evaluate related approaches [12, 20]. For each attack scenario, we generate the causal graphs from both the original log and the reduced logs, starting from an attack symptom event (for backward analysis) or the root attack event (for forward analysis). Then we compare the two graphs to verify if they contain all causal relations pertinent to the attack and further, if they carry any unrelated ones. To avoid having over-sized graphs, the original logs are generated with execution partitioning (i.e., BEEP) [20]. In other words, we are comparing graphs from LogGC with graphs by BEEP (only).

Scenarios	# of audit log entries		Back-ward	For-ward
	Total	After GC		
1. Trojan attack [20]	356,798	9,614 (2.69%)	Match	Match
2. Attack ramification [20]	690,231	50,271 (7.30%)	Match	Match
3. Information theft [20]	572,712	178,213 (31.12%)	-	Match
4. Illegal storage [12]	212,321	59,236 (27.90%)	Match	Match
5. Content destruction [12]	328,297	37,282 (11.36%)	Match	-
6. Unhappy student [12]	572,385	45,821 (8.01%)	Match	Match
7. Compromised database [12]	102,415	4,657 (4.55%)	Match	Better
8. Weak password [12]	182,346	43,214 (23.70%)	Better	Better

Table 8: Attack scenarios ("Match" means identical causal graphs with and without LogGC; "Better" means a smaller and precise graph with LogGC.)

Table 8 summarizes the results. The second and third columns show the number of audit log entries in the original and reduced logs, respectively. The last two columns show the results of causal graph comparison in both backward and forward analysis. The results show that all causal graphs by LogGC capture the minimal and precise attack paths and the right set of attack ramifications; whereas the graphs from the original logs either are identical to their counterparts or contain extra (and unrelated) causal relations.

In the **first** scenario ("trojan attack"), the victim received a phishing email that contained a malicious URL. The user clicked it and `firefox` visited the malicious page. The user downloaded a backdoor trojan and executed it. The administrator later detected the backdoor program and started forensic analysis. In the **second** scenario, the attacker exploited a vulnerability of `Proftpd` to acquire a root shell and installed a backdoor. Then he modified *.bash_history* to remove the footprint. The user later noticed the backdoor process. The causal graphs (from both the original and the reduced logs) precisely capture that the attacker modified `.bash_history`. The **third** scenario involves information theft. An insider used `vim` to open three classified files and two other html files. He copied some classified information from *secret_1* to *secret.html* and also modified *index.html*. Then an external attacker connected to the web server and read *secret.html*. LogGC does not garbage collect the `httpd` units that sent the modified html files and thus the reduced log contains the event that sent *secret.html* to the attacker. The forward casual graphs from the secret files clearly tracks down the attack, including the attacker's IP through which he retrieved the secret information. In the **fourth** scenario, the attacker launched a `pwck` local escalation exploit to get a root shell and then modified */etc/passwd* and */etc/shadow* to create an account. Then the attacker created directories and downloaded illegal files, including a trojaned `ls` to hide the illegal files. A victim user used the trojaned `ls` and created two files in his home directory. Later, the attacker logged into the system using the created account and downloaded more illegal files. The administrator later detected the trojaned `ls` and started forensic analysis. There are two forward causal graphs in this case. The first one starting from the trojaned `ls` identifies the victim user and the files generated, which may be compromised. The second one from */etc/passwd* is larger because it includes all `ssh` login activities. The attacker's login and download activities are captured but events from normal users are



(a) Graph by LogGC **(b) Graph without data unit support**

Figure 8: Causal graph comparison for attack scenario 7.

also included. In the **fifth** scenario ("content destruction"), the attacker exploited a `sendmail` vulnerability to get a root shell and he deleted files from other users' directories. The victim detected some of his files were missing and restored them from backup storage. In the **sixth** scenario, the attacker launched a remote attack on the ftp server and modified some file permissions to globally writable. Two other malicious users modified a victim's files and copied them into their own directories. The victim later detected that his files were globally writable.

In the next two scenarios, *mysql* played an important role on the attack paths. LogGC hence produces better causal graphs than those derived from the original logs. In the **seventh** scenario, the attacker launched a remote attack on the Samba daemon to get a root shell and created a backdoor. The attacker logged in later through the backdoor and issued SQL queries to remove some transaction from the local database. Later the user accessed the database and detected problems. We performed backward analysis from the backdoor. The causal graphs from both the reduced log and the original log are identical. However, data units prove to be very effective for forward analysis such that we can precisely pinpoint tuples affected by the attacker. In contrast, the graph from the original log indicates that the entire table may be affected by the attacker. In Fig. 8, we compare the forward causal graphs for this case with and without data unit support. The graph by LogGC (with data unit support) precisely detects the tuple modified by the attacker, whereas the graph without data unit support indicates that the entire *bid* table was affected and it also shows three other users who accessed table *bid* even though they did not access the modified tuple. The user from `x.x.x.1` modified a tuple in table *item* after he read *bid* hence table `item` is considered affected. After that, another user from `x.x.x.4` accessed table *item* and thus is also included in the graph. As a result, the graph by LogGC contains 5 nodes which precisely capture the attack ramifications, whereas the graph without data unit support has 16 nodes including 10 unrelated objects and users.

In the last scenario, the administrator used `photo-gallery` to upload digital pictures and created an account with a weak password for the user. Before the user changed the password, the attacker grabbed the password using dictionary attack. The attacker logged into the gallery program, uploaded some pictures, and viewed the user's album. The user later detected the attacker's pictures. The graphs by LogGC are precise in revealing the attack: The backward graph includes 47 nodes and the forward graph contains 61 nodes. Both are verified to carry the precise set of forensic information items related to this attack. In contrast, the backward graph from the original log contains 326 nodes and the forward graph has 517 nodes. Most of them are introduced by false database dependences.

7. RELATED WORK

Classic Garbage Collection. There is a large body of work on garbage collecting in-memory objects [4, 9, 10, 14]. The nature of our problem has some similarity to classic garbage collection. However we cannot simply use classic GC for provenance log reduction for the following reasons. First, we have to operate on audit logs with a flat structure instead of on memory reference graphs in classic GC. Second, classic GC only needs to identify object reachability in one direction; but we have to consider both forward and backward directions to cater for attack forensic needs. Third, classic GC can make use of very precise byte level reference information to determine reachability; whereas we only have the coarse-grain system level dependences in audit logs. As shown in our experiments, a basic reachability-based GC algorithm can hardly work on audit logs.

System-level Provenance. In recent years, significant progress has been made on tracking system-level dependences for attack forensics using audit logs [3, 8, 11, 13, 16, 17, 18, 20, 22, 23, 24]. These techniques use audit logs to identify root cause of an attack and perform forward tracking to reveal the attack's impacts. LogGC complements these techniques by garbage collecting audit logs to substantially reduce their size without affecting forensic analysis accuracy. In particular, while we leverage the execution partitioning technique in BEEP [20], LogGC and BEEP differ in that: (1) LogGC focuses on garbage-collecting audit logs whereas BEEP does not; (2) BEEP cannot handle dependences with database engines, which are critical to reducing server audit logs and generating precise causal graphs.

System-level replay techniques have been proposed to roll back a victim system after an attack [6, 12, 15]. They record system-wide execution events so that the whole system can be replayed from a checkpoint. LogGC may potentially complement these techniques by garbage-collecting unnecessary events from execution logs without affecting replay fidelity.

Database Provenance. There exists a line of research in providing fine-grain data lineage for database systems. Trio [2] and SubZero [25] introduce new features to manage fine-grain lineage along with data. They track provenance by transforming/reversing queries. As such, they need to know the queries before hand, without instrumenting the database engine.

Log Compression. Some existing techniques involve compressing provenance logs via a web graph compression technique [26] or detecting common sub-trees and then compressing them [7]. As log compression techniques, they are orthogonal to LogGC . We envision future integration of LogGC and these techniques.

8. CONCLUSION

We present LogGC, a GC-enabled audit logging system towards practical computer attack forensics. Audit log garbage collection poses new challenges beyond traditional memory GC techniques. It should support both forward and backward forensic analysis whereas traditional memory GC only needs to support one direction of correlation. Moreover, the granularity of existing audit logging approach is insufficient, especially for long running programs and database servers. We propose a technique that partitions a database file into data units so that dependences can be captured at tuple level. Together with our earlier solution of partitioning a process into execution units, LogGC greatly reduces false dependences that prevent effective GC. LogGC entails instrumenting user programs only at a few locations, incurring low overhead ($< 2.04\%$). Without any compression, LogGC can reduce audit log size by a factor of 14 for user systems and 37 for server systems. The reduced audit logs preserve all necessary information for full forensic analysis.

9. ACKNOWLEDGMENT

We would like to thank the anonymous reviewers and our shepherd, Wenliang Du, for their insightful comments. This research has been supported in part by DARPA under Contract 12011593 and by NSF under awards 0917007 and 0845870. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of DARPA and NSF.

10. REFERENCES

[1] http://sysbench.sourceforge.net.

[2] P. Agrawal, O. Benjelloun, A. D. Sarma, C. Hayworth, S. Nabar, T. Sugihara, and J. Widom. Trio: a system for data, uncertainty, and lineage. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB 2006.

[3] P. Ammann, S. Jajodia, and P. Liu. Recovery from malicious transactions. *IEEE Transaction on Knowledge and Data Engineering*, September, 2002.

[4] A. W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 1989.

[5] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of ejb applications. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA 2002.

[6] R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich. Intrusion recovery for database-backed web applications. In *Proceedings of the 23th ACM Symposium on Operating Systems Principles*, SOSP 2011.

[7] A. P. Chapman, H. V. Jagadish, and P. Ramanan. Efficient provenance storage. In *Proceedings of the ACM SIGMOD international conference on Management of data*, SIGMOD 2008.

[8] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th conference on USENIX Security Symposium*, SSYM 2004.

[9] L. P. Deutsch and D. G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 1976.

[10] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM*, 1978.

[11] A. Goel, W.-c. Feng, W.-c. Feng, and D. Maier. Automatic high-performance reconstruction and recovery. *Computer Networks*, April, 2007.

[12] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The taser intrusion recovery system. In *Proceedings of the 22th ACM symposium on Operating systems principles*, SOSP 2005.

[13] X. Jiang, A. Walters, D. Xu, E. H. Spafford, F. Buchholz, and Y.-M. Wang. Provenance-aware tracing of worm break-in and contaminations: A process coloring approach. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, ICDCS 2006.

[14] R. Jones and R. Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., 1996.

[15] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. Intrusion recovery using selective re-execution. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI 2010.

[16] S. T. King and P. M. Chen. Backtracking intrusions. In *Proceedings of the 19th ACM symposium on Operating systems principles*, SOSP 2003.

[17] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen. Enriching intrusion alerts through multi-host causality. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium*, NDSS 2005.

[18] S. Krishnan, K. Z. Snow, and F. Monrose. Trail of bytes: efficient support for forensic analysis. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS 2010.

[19] M. Laurenzano, M. Tikir, L. Carrington, and A. Snavely. Pebil: Efficient static binary instrumentation for linux. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS 2010.

[20] K. H. Lee, X. Zhang, and D. Xu. High accuracy attack provenance via binary-based execution partition. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium*, NDSS 2013.

[21] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, PLDI 2005.

[22] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium*, NDSS 2005.

[23] S. Sitaraman and S. Venkatesan. Forensic analysis of file system intrusions using improved backtracking. In *Proceedings of the 3rd IEEE International Workshop on Information Assurance*, IWIA 2005.

[24] D. Tariq, M. Ali, and A. Gehani. Towards automated collection of application-level data provenance. In *Proceedings of the 4th USENIX conference on Theory and Practice of Provenance*, TaPP 2012.

[25] E. Wu, S. Madden, and M. Stonebraker. Subzero: a fine-grained lineage system for scientific databases. In *Proceedings of the 29th IEEE international conference on Data Engineering*, ICDE 2013.

[26] Y. Xie, D. Feng, Z. Tan, L. Chen, K.-K. Muniswamy-Reddy, Y. Li, and D. D. Long. A hybrid approach for efficient provenance storage. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, CIKM 2012.