Kernel-Supported Cost-Effective Audit Logging for Causality Tracking

Shiqing Ma[$], Juan Zhai[§], Yonghwi Kwon[$], Kyu Hyung Lee[*], Xiangyu Zhang[$],
Gabriela Ciocarlie[†], Ashish Gehani[†], Vinod Yegneswaran[†], Dongyan Xu[$], Somesh Jha[‡]
[$]*Purdue University,* [§]*Nanjing University,* [*]*University of Georgia,*
[†]*SRI International,* [‡]*University of Wisconsin-Madison*

Abstract

The Linux Audit system is widely used as a causality tracking system in real-world deployments for problem diagnosis and forensic analysis. However, it has poor performance. We perform a comprehensive analysis on the Linux Audit system and find that it suffers from high runtime and storage overheads due to the large volume of redundant events. To address these shortcomings, we propose an in-kernel cache-based online log-reduction system to enable high-performance audit logging. It features a multi-layer caching scheme distributed in various kernel data structures, and uses the caches to detect and suppress redundant events. Our technique is designed to reduce the runtime overhead caused by transferring, processing, and writing logs, as well as the space overhead caused by storing them on disk. Compared to existing log reduction techniques that first generate the huge raw logs before reduction, our technique avoids generating redundant events at the first place. Our experimental results of the prototype KCAL (Kernel-supported Cost-effective Audit Logging) on one-month real-world workloads show that KCAL can reduce the runtime overhead from 40+% to 15-%, and reduce space consumption by 90% on average. KCAL achieves such a large reduction with 4% CPU consumption on average, whereas a state-of-the-art user space log-reduction technique has to occupy a processor with 95+% CPU consumption all the time.

1 Introduction

Understanding system provenance is an important and challenging task, especially in forensic analysis and problem diagnosis. A common approach is to perform operating system-level audit logging, which is one of the core functionalities required in enterprise-level infrastructures. The Linux Audit system is the most widely used audit system. It resides in the kernel, collects information for predefined kernel events, and records them in log files. Following incidents, investigators use automated tools (e.g., ausearch) to analyze audit logs to search for suspicious system objects (e.g., files, sockets) and subjects (e.g., processes), and identify causal dependencies among them. Such information is critical to locating root causes and assessing damages. Then they use such information to hunt for suspicious activities such as policy violations. In practice, the Linux Audit system has been known to have poor performance, and other researchers have been working on improving the Linux Audit system for a long time. Many works [8, 9, 22, 24, 31, 41, 42] proposed enhancement or alternative designs to provide fast logging infrastructures or highly compressed logs. However, existing solutions do not fundamentally solve the high space and runtime overhead problems. And this motivates us to deeply analyze and understand the overhead problems in the Linux Audit framework.

In this paper, we first describe a comprehensive analysis on the Linux Audit system, and show that the runtime and the storage overheads are essentially caused by transferring and processing huge raw logs that contain substantial redundancies. Previous research failed to solve the problems because methods required first generating the redundant logs. Our key idea is *to remove redundancies inside the kernel so that we can prevent the huge raw logs from being generated at the first place.* Inspired by hardware/software cache system designs, we propose KCAL, a kernel-level, cost-effective, memory-cache-based audit logging system. It caches important dependencies and events, and detects redundancy on the fly using the caches. If redundant events indicated by cache hits are detected, they are immediately discarded. Only events that introduce new system objects/subjects or new dependencies are retained. Dependency caches and event caches are distributed in individual kernel data structures. The caches are carefully designed such that the kernel memory consumption is kept reasonably low, avoiding perturbation of normal kernel functionality. In summary, in this paper, we make the following contributions:

- We describe a comprehensive analysis on the Linux Audit system, which revealed that the root cause of its high runtime and storage overheads is the need to transfer, process, and store the huge raw log, and identify this can be solved by removing the redundant events.
- We propose a kernel-level, cache-based, log-reduction system. The key idea is to prevent the kernel from generating redundant raw logs in the first place. The design features a multi-layered and distributed cache scheme that leverages the autonomous execution sub-structures (i.e., units) in individual processes (e.g., sub-executions serving individual requests in Apache), and indexes largely scattered syscall events belonging to the same object.
- We built a prototype KCAL based on the Linux Audit system. Our experimental results showed that KCAL is capable of reducing the runtime overhead from 40+% to 15-%, log files by 90+%, and it does not introduce significant memory pressure on the existing kernel. The comparison with the state-of-the-art, user-space log-reduction technique ProTracer [24] shows that ProTracer fully occupies an idle processor with 95% constant CPU consumption whereas KCAL only requires 4% CPU consumption on average.

2 Motivation and Related Works

2.1 Audit Logging Systems

There are many existing audit logging systems [2, 5, 7, 24, 28, 31] from commercial companies and research communities. Prior works [13, 14, 16, 35, 37, 43] proposed many different general logging infrastructures. Some of them [11, 27, 28, 34] monitor the whole file system at the `inode` level, while others [31, 36] leverage the Linux Security Module (LSM) to monitor operations on kernel data structures. Many of the techniques [12, 17, 18, 19, 20] use record-and-replay techniques to record system wide events for system replay. They require logging of syscalls including the concrete values such as the content of files or packets. Hence, they tend to be expensive and are mostly used in single application execution. Bates *et al.* provide a general and secure framework for writing a provenance system at the operating system level. Among these provenance systems, the Linux Audit framework [2] is the most practical and widely used. The framework provides a general logging infrastructure that allows the integration of plugins to enhance the system. As such, it is widely used and has been adopted by many other research projects and real-world products [3, 4, 6].

Linux Audit Architecture. Figure 1 shows the architecture of the Linux Audit framework. It contains a few user-space utilities (brown boxes) and a kernel component. The kernel component contains a number of filters (blue circles). Based on the execution order, i.e., before/during/after the syscall processing logic, the filters are named `User/Task/Exit`, respectively. The `Exclude` filter defines exceptions to the filtering rules; namely, any syscall that falls into the `Exclude` category will not be filtered. The `auditctl` program helps administrators manage filters. If these filters determine that a syscall needs to be recorded, the kernel component sends the information to the user-space daemon program `auditd` through the `Netlink` device. `Auditd` collects syscall records and writes them to the log file.

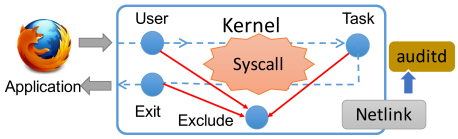

Figure 1: The audit framework architecture

State-of-the-Art Causality Analysis. An important feature of an audit logging system is the dependency analysis support. As demonstrated by previous researchers [21,23], the Linux Audit system suffers from the *dependency explosion* problem because of the large number of fan-outs in process-level analysis. Process execution partitioning techniques [21, 23] were proposed to enable fine-grained dependency analysis in audit logging, and to help remove redundant log information. They partition process executions into *execution units*. Each execution unit is a part of the whole process execution serving a specific task. MPI [23] partitions process execution based on user-defined tasks, e.g., individual tabs in Firefox. BEEP [21] partitions process execution based on event-handling loop, namely, and an execution unit is essentially an iteration of the event handling loop. Execution units are considered largely autonomous. Therefore, *an output syscall event in a unit is considered only dependent on the preceding input events within the same unit unless there are dependencies across units (e.g., through in-memory data structures)*. In contrast, Linux Audit considers that an output event depends on all the preceding events in the same process, causing numerous bogus dependencies [14,18,20,21]. An execution unit is delimited by a special `UnitEnter` event indicating the start of the unit, and a `UnitExit` event denoting the end in these systems. An execution unit may depend on another through variable reads/writes. Such variables/data-structures are treated as Inter-Process Communication (IPC) objects, and exposed to the audit system via the `MemWrite` and `MemRead` syscall events [21, 23].

Figure 2 shows an example of using Firefox to open webpages and download a file (File-N). It also shows the simplified log events. In each line, we show the events

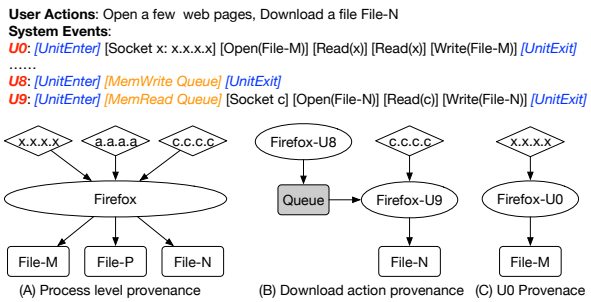

Figure 2: Comparison of different dependency granularity

that belong to a unit marked with the unit ID. Without unit information, we will get a graph shown in (A). The file object File-M (transitively) depends on all the socket read by Firefox before the write to the file, which introduces many bogus dependencies. With partitioning, the events are properly grouped. For example, the first unit, *U0* represents a unit for opening a web page. It creates a socket, fetches a page, stores it, and then renders it on screen. The dependency relationship is shown in (C). The multiple page loading tasks are separated to units, and the resulting provenance graph is accurate. Downloading in Firefox causes explicit dependencies between units (*U8*, *U9*). *U8* first inserts the download request to a queue, and then *U9* fetches it from the queue and downloads the file. These units are connected through `MemWrite/MemRead` events as shown in (B).

2.2 Linux Audit System Performance

To motivate our technique, we perform a few experiments to measure the overhead of Linux Audit and explain the limitations of existing overhead reduction techniques. We run 20 virtual machines with Ubuntu 14.04 as the guest OS on the Kernel-based virtual Machine (KVM) platform, and each virtual machine has two cores and 4 GB main memory. The machines are classified into two categories: servers running server programs (e.g., HTTP server Apache, FTP server ProFTPd), and clients running client programs (e.g., Firefox and Vim) for daily use. Each group has 10 virtual machines.

Figure 3 shows the log size growth along time. We configure the audit system to only record 60 provenance-

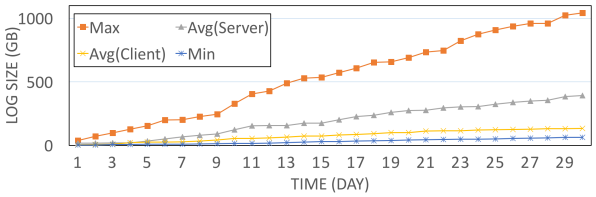

Figure 3: The audit framework log sizes growth in 30 days

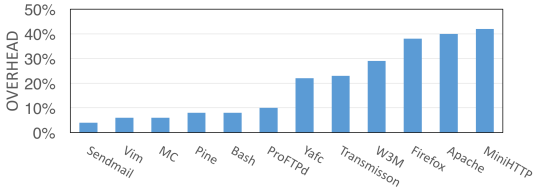

Figure 4: The audit framework runtime overhead

related syscalls [24, 31]. These system calls are related to process creation/termination, file/socket creation/read-/write/deletion and IPCs and so on. Observe that in the worst case, a machine generates 1100+GB log in 30 days. Even in the best case, 60 GB log is generated within 30 days. On average, a server machine can generate about 130GB data per day, whereas a client machine generates about 5GB data per day. The data is also consistent with previous research [22, 41]. Such a large volume of data causes many problems. First, it is expensive to store or transmit log files. By default, the audit log is stored on the local disk and consumes substantial storage space. It can be sent to external servers for storage and inspection, but this incurs runtime overhead, network traffic, and maintenance efforts on servers. Second, processing such large files can be extremely challenging. It may take hours to days to answer a provenance query as it requires searching through log files in the size of GBs to TBs. The situation becomes worse in the enterprise environment, where there are hundreds to thousands of inter-connected machines, which increases the problems associated with storing, correlating, and processing audit logs. Compressing the log is one way to reduce the storage overhead, but causes more runtime overhead for compressing/decompressing the logs to investigate an attack. Zhang *et al.* [41] also demonstrated that in an enterprise environment, using databases to store the logs is also very challenging in such scenarios.

Figure 4 shows the runtime overhead (caused by Linux Audit) for a few programs including both server programs like Apache and client programs like Vim. We leverage existing workloads to test the performance if possible. For programs that support batch mode (e.g., Vim), we write scripts to test the performance. Some of the programs generate frequent system calls (e.g., Apache), and naturally cause higher runtime overhead. As we can see, the overhead for some programs like Vim is tolerable. But for I/O intensive programs such as browsers and server programs, the overhead can be rather high.

Understanding the Overheads. The Linux Audit framework has three parts: the kernel part (filtering rules etc.), the `Netlink` data transmission channel, and the user space logger (i.e., `auditd`). Figure 5 shows the runtime overhead of each component. Similar observations are made on both Hard Disk Drives (HDDs) and Solid State

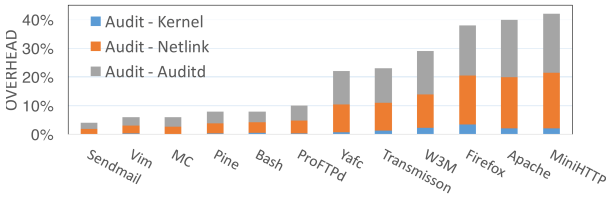

Figure 5: Audit framework runtime overhead

Drives (SDDs). The graph tells us the kernel filters are relatively lightweight, and the other two parts, `Netlink` and `auditd`, are the dominant factors of the overhead. `Netlink` provides a socket-like channel for transmitting data, and `auditd` is responsible for writing the log data to disk. The major factor that affects the time spent on these two components is the size of log data that needs to be processed (transferred/written). Considering the amount of data we need to handle (Figure 3), it is understandable these two parts dominate the overhead. As such, we can say *the root cause of both runtime and storage overheads is the large amount of data generated by the Linux Audit framework.*

2.3 Log Redundancy

Previous works addressed the storage overhead problem by shrinking the log size. Most of them [10, 15, 25, 29, 30, 32, 33, 38, 39, 40] generated the dependency graph first, and then used various graph visualization or compression algorithms to help causality analysis. These techniques ignore the importance of reducing the redundancy of audit logs, and cannot solve the runtime overhead problem caused by such redundancy.

Existing Linux Audit generates highly redundant logs. Based on our analysis (see §4), over 89% log entries are redundant. Previous research [22, 24, 41] has also presented similar observations. Thus pruning the log could improve the performance of the Linux Audit system. Some existing works [8, 41] suggest removing redundancy by various analysis techniques, e.g., rule-based filtering. However, this requires human effort to create and maintain the rules. ProTracer [24] leverages execution partitioning for log reduction. It has a kernel module, which simply receives syscall events, filters them, and then sends the remaining event records including unit-related events to the user-space daemon, which consists of multiple processes. These processes run in parallel to remove redundant events. The ProTracer views system objects as taints and monitor their propagation during execution by performing syscall level taint analysis while processing the log. Each unit/object is associated with a taint set denoting the set of data sources that it depends on. The causalities denoted by the taint sets (instead of individual events) are emitted to the log. Therefore events leading to the same taint set are essentially reduced.

All these techniques first generate the full-fledged log and then reduce it. It is the huge raw log that causes the substantial overhead. These techniques cannot be applied in the kernel space because it has rather limited resources that prevents loading and processing huge raw logs. For instance, the parallel (tainting-based) processing required by ProTracer cannot be ported to the kernel space due to its high CPU consumption (See data in §4). An ideal solution is to *prevent redundant log entries from being generated by the kernel in the first place*. This is the motivation behind KCAL, a kernel-supported log cache and reduction system.

3 Design

3.1 Overview

We propose a cache-based, cost-effective audit logging system inside the kernel called KCAL. It leverages execution partitioning and is orthogonal to the underlying partitioning scheme. Any partitioning scheme [21, 23] that generates unit boundary syscalls and cross-unit memory dependency events can be seamlessly integrated with KCAL, and we use BEEP. Upon a new syscall event, KCAL determines if there is a cache hit, which means the new event reveals the same causal information as some event(s) that have been recorded before and hence can be safely discarded. Since the cache is positioned at the kernel, redundant log events are prevented from being generated in the first place, leading to highly succinct raw logs without any information loss. KCAL is not a monolithic caching system like traditional memory caches because different subjects/objects have diverse life times and various numbers of associated syscall events distributed in their life spans. Due to the nature of audit logging, we cannot be certain if events belonging to a subject/object are redundant before it is closed or terminated. A monolithic cache design would require complex data structure support for indexing and removing sparse and highly distributed log events. Therefore, we propose a distributed cache design so each process/object (e.g., a file) has its own cache storing associated events, and these caches are encapsulated as part of the kernel data structures. Figure 6 shows the overall architecture of KCAL.

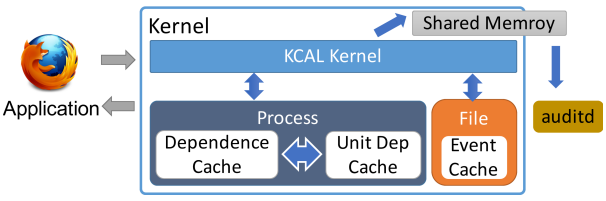

Figure 6: Overview of KCAL architecture

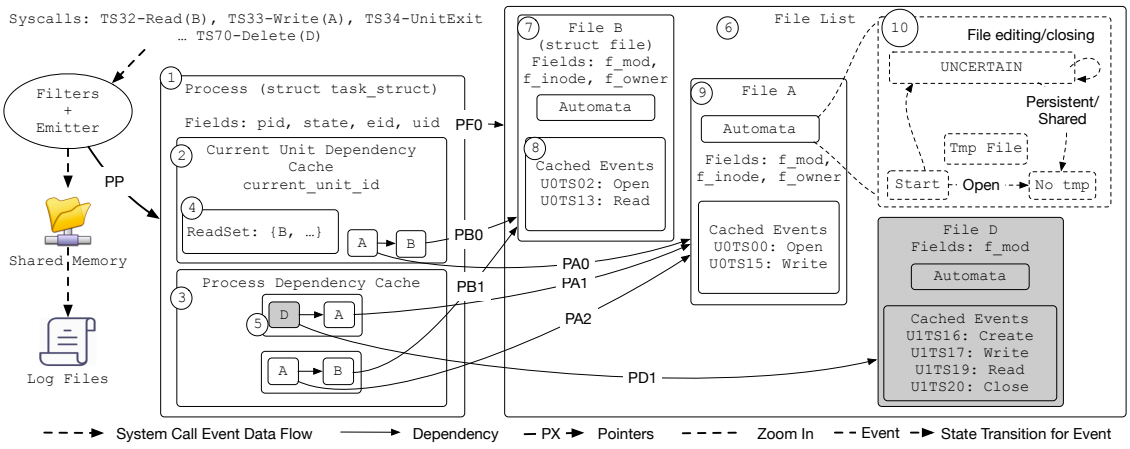

Figure 7: Overview of KCAL

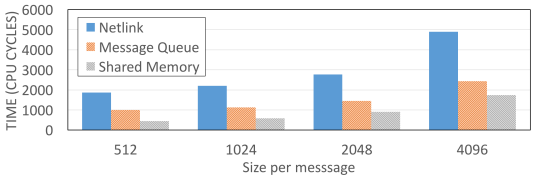

Figure 8: Performance of 3 data channels (kernel to user-space)

First, we enhance the Linux Audit module with an online cache-based log-reduction algorithm, and modify the kernel data structures for processes and objects (e.g., files and sockets) to insert caches. Second, we use shared memory instead of `Netlink` as the transfer channel.

In-kernel Architecture and Workflow. Figure 7 shows a simplified view of the kernel part of KCAL. The first modification is in the `task_struct` data structure (①), which stores process specific information such as the *pid*. We add more pointer fields. The first one is a pointer to a unit dependency cache (box ②). The cache uses a *ReadSet* to store the objects that have been read by the current unit (box ④), and also maintains the detected dependencies in the current unit (e.g., $A \rightarrow B$ shown inside ②). Each object in the cache also has a pointer (e.g., PA0 and PB0) to the corresponding kernel data structure instance such as a `File` structure. The second pointer points to a process-level dependency cache (box ③), which stores the dependencies detected in this process (box ⑤) by aggregating the unique dependencies from individual units. The unit cache is needed for in-unit redundancy and the process cache is for cross-unit redundancy.

We also enhance the kernel data structures representing objects (resources). For example, we enhance the `File` data structure that contains file-specific information, such as its inode, by adding two pointers. The first one points to a cache that stores the syscall events operating on the object with timestamps (box ⑧). Redundant events are removed at the unit/process level before being added to the object cache. We do not directly send these events to the user space but rather cache them because all the events in the object cache may be deemed redundant if the resource is determined as temporary. More details will be discussed in §3.3. The second pointer points to an automaton used to detect if the resource is temporary (box ⑨). Box ⑩ shows the states and the transitions. Details will be discussed in §3.3. KCAL does not cause any compatibility issues as it does not change the meanings of existing fields in these data structures. More importantly, our method is general, and one could easily use stand-alone hash tables that map a process/object to its auxiliary data structures and avoid touching any kernel data structures. As we will show in §4, although KCAL is mainly kernel based, its perturbation to the normal kernel functionalities is negligible due to its small memory footprint and limited instrumentation inside the kernel.

The Linux Audit module is enhanced with an on-the-fly reduction algorithm that interacts with the caches to determine if an event is redundant. When a syscall event occurs, it first goes through the filters. Non-provenance related syscalls like time-related system calls are filtered out. The remaining syscalls (i.e., reads/writes) are passed to the reduction component. This component checks if there is a cache hit for the dependency represented by the event. Note the caches are accessible through the `current` variable, which points to the `task_struct` of the current process that contains direct or transitive pointers to multiple layers of caches. If hit, the event is safely discarded. Otherwise the dependence is inserted to the dependence cache, and the event is inserted to the event cache of the object that is being operated on. Eventually, non-redundant events will be emitted to the shared memory and saved to the disk by the user-space component.

Transfer Channels. `Netlink` provides a socket-like communication method between the kernel space and

the user space and was widely adopted by SELinux as it provides a simpler interface and better performance as compared with its competitors (printk, ioctl etc.). We compare three general ways of transferring bulk data from the kernel space to the user space: `Netlink`, message queues, and shared memory. Figure 8 shows the performance comparison of these channels. The X-axis represents the size of each message. We use four configurations: 512, 1024, 2048, and 4096 bytes. For each message size, we generate 10,000 random messages and perform the experiments 10 times. The Y-axis is the performance measured by the average time (CPU cycles) used to transfer one message. Shared memory has the best performance. In the past, due to the memory size limits made it practical to use shared memory as the transfer channel as it requires reserving a memory pool, but this is no longer a problem in modern computers.

3.2 Redundancy in the Linux Audit Log

Our definition of *redundancy* is with respect to the attack investigation, which is based on a causal graph according to the latest *Open Provenance Model* (OPM) [26]. OPM standardizes the forensic analysis procedure and is the most widely adopted provenance model. A causal graph is generated by first starting from a given subject or object (e.g., a suspicious file) and then performing *forward/backward* traversal along dependencies to find all the reachable subjects and objects. Backward traversal is used when the inspector wants to trace back the root cause of an attack starting from some observed symptom. In contrast, forward traversal is used when the inspector has already identified the root cause and now wants to understand the damage caused by the attack. It starts with the root cause and finds all the affected subjects/objects. In this context, *we consider an event redundant if the derived causal graph contains the same dependency information with and without the event*. That is, we can reach the same set of objects and subjects with and without the event. As such, an event is redundant if it leads to some dependency that was induced in a previous unit. This is because the previously recorded events and the entailed dependence render the same reachability. Events denoting the same dependency may be in the same or different execution units, and they are referred as in-unit redundancy and cross-unit redundancy, respectively.

Another type of redundancy is what we call *temporary files*. We define the term *temporary file* from the provenance analysis perspective. A *temporary file* is a file that is created, edited, and deleted by the same owner and during its whole life cycle; the file is not "shared" with any other process. These files only temporarily exist in the system and are used internally by applications. For example, editors with auto-saving features often use a temporary file to keep the newly edited contents to support recovery. These files are deleted when the user saves the file explicitly. Another example is that browsers like Firefox use a large number of temporary files to store downloaded web elements. The browser regularly removes such files to save space. Temporary files do not lead to useful forensic information as they do not have interactions with other system objects or subjects and can be removed. Many programs use temporary files because they need to save a large amount of contents locally, and memory is not sufficient for them to do so. From the provenance point of view, the temporary files are a part of the program execution just like the runtime variables in memory. As all the information source and sink points are the same, it can be viewed part of the program execution.

Our definition of *temporal files* is different from that of traditional temporary files that usually refers to the files in the `tmp` file system. Many of these files can interact with other processes and generate new provenance information, and thus the corresponding events are not redundant according to our design. For example, Firefox has one `open with` option for many types of files such as torrent files in its download dialog. It will first download the selected file (e.g., one torrent file), and then open the file using the system default program for this file type. In this case, the downloaded file is stored in the `tmp` file system, but it will be kept in our design as it is read by another process, which is new provenance information.

Therefore, KCAL guarantees the information completeness from the provenance graph point of view. Namely, the log files before and after reduction will output the same provenance graph for the same provenance query. As a provenance tracking system, it does not guarantee information completeness for other audit purposes such as the total number of syscalls in a time range.

3.3 Redundancy Reduction

In this section, we explain the details of the KCAL design. There are three important design choices: *a two-layer dependency cache (the unit layer and the process layer) for a process*, *a distributed event cache for objects*, and methods for *handling cross-unit memory dependencies*.

Two-layer Dependency Cache For Process. As shown in Figure 7, we cache two-layer dependencies for a process, the dependencies detected in the current unit (box ②) and those in the whole process (box ③). The former is to remove in-unit redundancy, and the latter is to reduce cross-unit redundancy.

• *In-unit Redundancy.* Read/write syscalls use buffers with limited sizes to transfer data. To load/edit a file larger than the buffers, an application has to issue a sequence of read/write syscalls. For example, Vim reads a file piece by piece and adds the pieces to the in-memory content

tree. This produces a sequence of events in the audit log (without reduction) containing tens to hundreds of read syscalls on the target file in the same execution unit. These syscalls denote the same dependency and are redundant. Such redundancy is detected by the unit cache in KCAL. KCAL only keeps one instance of them. At the first read event, the object is added to the `ReadSet`. If it is already in the `ReadSet`, the event is simply discarded. When a write event happens, it is considered dependent on all the objects in the `ReadSet` as the information from these objects can affect the content it writes. KCAL checks if these dependencies are present in the unit-dependency cache. If not, it adds the write event and the read events of objects in the unit cache to the event caches of the corresponding objects. Otherwise, the event is discarded.

• *Cross-unit Redundancy.* Processes usually perform repeated actions on the same system objects. Some of them are because of repeated user actions. For example, editors usually work on a few files for a long time with repeated editing operations. And some of them are due to built-in application functionalities. For example, Firefox writes to the `recovery.js` file every 15 seconds (through a unit) to support purpose. As a result, the same dependency can appear in the log file across different units repeatedly, leading to *cross-unit redundancy*.

An example in Figure 9 on the left-hand-side, (A) shows the simplified log entries. There are three units. Unit 0 (`U0`) reads File-A, File-B, and writes File-S; `U1` reads File-B and writes File-T; and `U3` reads File-A and writes File-S. The blue entries are the `UnitEnter`/`UnitExit` events. The yellow entries are the in-unit redundant events. In particular, `U0` keeps loading contents from File-A. Events `U0TS03` to `U0TS05` all represent the same action, and are redundant. The red entries are the cross-unit redundant events. In this case, the causal dependency between File-A and File-S in `U3` is already detected in `U0`, and hence is redundant. The graphs in (B) show the generated backward analysis graphs starting from File-S and File-T (in gray), and the graphs in (C) represent the generated forward analysis graphs starting from File-A and File-B (in gray). Events `U3TS01` and `U3TS02` do not induce any new dependencies and removing them does not affect the reachable objects and subjects in both forward and backward analyses. Without the execution partitioning, File-T would depend on File-A because the process loads File-A before writing to File-T. As shown later in §4, our reduced logs generate the same casual graphs as using the original BEEP logs (with redundancy reduction).

Distributed-event Cache. In KCAL, dependencies are cached in processes and syscall events are cached in objects. KCAL features a distributed-event cache mechanism in which each object caches the syscall events that operate on the object. They are not transferred to the user

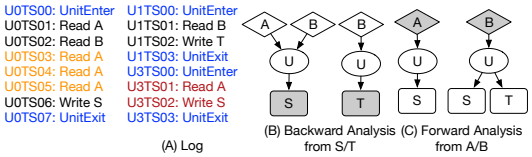

Figure 9: In-unit and cross-unit redundancy removal example

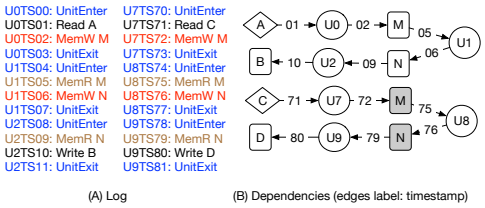

Figure 10: Cross-unit memory causality example

space for storage until the object is no longer needed or the process terminates. This is to handle the substantial redundancy caused by temporary files (defined in §3.2).

We detect temporary files using the automaton shown in box (10) in Figure 7. Each `File` data structure maintains its own state. At first, when a file is opened by the owner process, KCAL checks the creator of the file. If the file exists and was created by another process, it is marked as a non-temporary file. Otherwise, it can potentially be a temporary file (i.e., the *UNCERTAIN* state). Normal file editing operations by the owner such as read/write/close do not change the state of the file. Any operation from a different process indicates that information propagates beyond the current process through the file, and hence the file must not be temporary. If the file is deleted by its owner without being read/edited by other processes, it is temporary. If the file is not deleted by its owner process (and hence is persistent), it is not temporary.

As KCAL cannot be certain if a file is temporary or not until the file is deleted, edited by other processes, or the owner process terminates, it buffers all the events for a file it created (after redundancy reduction) in the cache associated with the file until either condition is satisfied. Then, KCAL discards all the events in the cache or emits them to the user space. The emission order of events may be different from the temporal order due to the distributed caching. It is not problematic, however, as all events have global time stamps.

Cross-unit Memory Causality. As mentioned earlier, there may be dependencies across units caused by variables or data structures. For example, in Vim's built-in clipboard, a piece of memory (known as *registers*) is used to support copy/cut-and-paste operation across units.

Existing execution partitioning schemes generate special syscalls `MemWrite`/`MemRead` to denote the write/read operations on cross-unit, dependency-inducing variables, respectively. The nature of these memory operations is

very similar to file reads and writes. Hence, KCAL models these events in a similar way. Particularly, each unique memory location is considered as a separate object. The difference is we do not remove events that cause the same memory dependencies across units. Instead, KCAL treats the memory object as a new object each time it is redefined. This is because each memory write to a location is considered as a complete redefinition of the memory object, which is different from a file write. For such an object, each read only depends on the latest write.

For example, in the syscall sequence in Figure 10, unit `U1` receives a request through the memory queue from `U0` at location `M` and then forwards another request to `U2` through a different memory location `N`. KCAL detects a dependency from `N` to `M`. Later, the same procedure happens again and hence the same dependence is detected inside unit `U8`. The same memory locations are observed due to memory reuse, and we cannot unify the multiple instances of the memory locations and throw away the memory events in `U8`. Otherwise, bogus dependencies would be introduced. In the shown example, `D` only depends on `C`. If the dependency introduced by `U8` is removed and the two `M` nodes are unified, `D` would depend on {`C`,`A`}. In KCAL, the variables `M, N` associated with `U8` and the ones associated with `U1` are treated as a new set of system objects even though they are using the same memory addresses. KCAL leverages existing execution partitioning techniques and existing execution partitioning techniques only instrument a very small number of memory operations through sophisticated analysis [21, 23]; and hence, the number of memory events generated at run time is small.

3.4 Implementation and Discussion

KCAL is implemented on the long-supported Linux kernel 3.19 and the Linux Audit framework 2.3. By default, each system object cache size is 32 events. The number of dependencies a process can cache is capped at 256, and the number of dependencies a unit can cache is 8. These values are configurable in KCAL. If the cache is full, and we use the Least Recently Used (LRU) cache replacement policy to evict entries. It is important to note *the consequence of cache eviction is merely that some redundancy cannot be removed. It does not affect information completeness*. The study of the effect of various cache sizes can be found in §4.

KCAL is a provenance tracking system built on top of the Linux Audit framework. The audit log message format is still the same. This makes it compatible with existing audit log processing tools such as `aureport` and `ausearch`. On the other side, the generated messages are for provenance tracking only, and the number of such messages is significantly reduced. This will affect audit tools that calculate the sysall frequencies or concretely analyze individual syscalls such as `aureport`. Also, KCAL modifies the Linux kernel source code including many data structures. As a result, porting it to other kernel versions requires extra human effort. We also port our prototype from Linux kernel 3.19 to kernel 3.2, and most of the patches can be directly applied. The new modification is less than 10 lines. We expect that the porting efforts will be limited as long as the kernel data structure change is not significant. KCAL also depends on existing execution partitioning techniques such as BEEP [21] or MPI [23]. Without the execution partitioning support, cross-unit redundancies cannot be removed, which affects the reduction effectiveness (see §4).

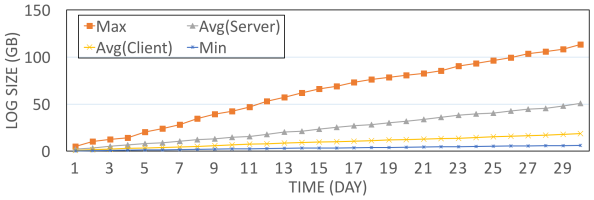

Figure 11: The space overhead of KCAL in a month

4 Evaluation

We evaluate our prototype to answer the following research questions (RQ):
RQ1: How efficient is KCAL? (§4.1)
RQ2: Can KCAL remove the redundancy while keeping the accuracy of the forensic analysis? (§4.2, §4.3)
RQ3: What are the rationales of our design choices, and what are the benefits? (§4.4)

4.1 KCAL Performance

Space Overhead. The space overhead is shown in Figure 11. The experimental environments and workloads are the same with the ones in §2. The orange shows the growth of log size for the machine that has the maximum size. In our case, the log file is less than 120GB after 30 days, while the old log size was almost 1TB without our technique (Figure 3). The gray line represents the average log size for the server machines, and the yellow line shows the average log size for the client machines. Compared with the original audit log (Figure 4), the log size is less than 10%. The workloads also include many applications that do not have the execution partitioning instrumentation, and thus do not benefit from log reduction. The blue lines show the log size of the machine that has the minimal log size. The log now is only about 6GB for 30 days. This shows that KCAL generates much smaller log files than the Linux Audit system.

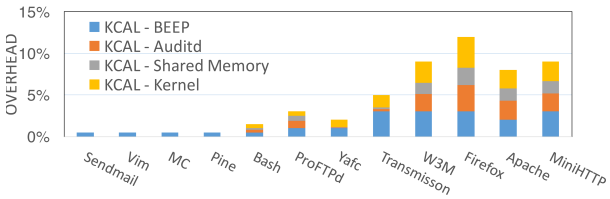

Figure 12: The KCAL runtime overhead analysis

Runtime Overhead. We used the same configuration and the same set of applications with the experiments used in §2 to measure the runtime overhead. Figure 12 shows the results. The bottom bars show the runtime overhead caused by the instrumentation, and the upper bars show the overhead caused by KCAL. For most client programs, the overhead caused by KCAL is less than 1%. The overhead for server programs is about 5% to 10%. This is because a server program needs to serve many clients at the same time, causing a large number of dependencies. Firefox has the most significant overhead, about 15%. This is because Firefox dynamically creates and terminates hundreds of threads, and uses many sockets and files for DNS queries, browsing history, cache, page preferences, and so on. It generates more events within the same duration compared with other applications, leading to higher overhead.

4.2 KCAL Log Reduction Effects

Table 1 summarizes the effects of using our log-reduction algorithm. The first two columns show the experimental environment. The third column shows the number of raw audit logs. We also present the number of log events and the corresponding percentage of in-unit redundancy (columns 4-5), cross-unit redundancy (columns 6-7), and temporary files (columns 8-9). The last two columns show the number of log entries and the percentages after reduction. We first ran the system on five machines for one month, collected the numbers (rows 3-7), and calculated the average (row 8, in gray). For different settings and runs, the reduction effects are different. Some of them have substantial in-unit redundancy (machines 1,2) while others have more cross-unit dependency (machines 4,5). Overall, KCAL keeps only 8% to 14% of the original logs, and on average the number is 11%. We also collected the reduction effects for some representative applications. The results are shown in rows 9-20 in Table 1. For different programs, the effects of the algorithm are significantly different. For example, most server programs do not have temporary files. On the contrary, browsers like Firefox use temporary files a lot. Server programs, especially FTP servers, need to read large files, and generate a huge number of in-unit redundant events, while this is not true for most client programs. For many programs like Vim, the dependency relationships are simple because they work on a limited number of system objects, and we can reduce the events to a very small number. For other programs like Bash, most of their events are related to process manipulations, which cannot be reduced. Thus, most of the logged events are kept. These process-related events will not be cached as they cannot be reduced, and they will not flood the cache. KCAL directly generates such reduced logs from the kernel, leading to substantial savings in raw log transfer from the kernel to the user space and in log processing compared to existing user-space reduction techniques [22, 24, 41].

4.3 Support for forensic analysis

We also performed a few experiments to verify the correctness of our log-reduction algorithm and the benefit for forensic analysis. We reproduced the attack cases used by previous research works [21, 23] and compared the generated graphs and the analysis time. In another set of experiments, we randomly selected 100 objects, and performed backward analysis to identify all of its dependencies. The results are summarized in Table 2. We show the number of nodes and edges in the graphs, the size of the log file, and the analysis time spent using the Linux Audit log, BEEP log, and KCAL log measured by log size. Note that BEEP logs are usually 10-30% larger than the Linux Audit logs due to the additional unit-related events. The last row shows the average number of nodes and edges, the average size of the log files, and the average analysis time for the randomly selected objects. We manually inspected and compared the graphs. The results show that all generated graphs contain the needed and complete information. The graphs generated by the Linux Audit framework usually contain redundant nodes/edges (representing wrong dependencies), whereas graphs generated by the other two methods generated the same graphs. This shows our reduction algorithm is lossless. Because of the complex dependency relationships, it takes far longer time to perform the analysis on the Audit log. BEEP benefits from simpler dependency relationships, but it spends more time inspecting the large number of log entries and checking and updating the dependency sets for each event including many redundant operations. The KCAL log provides simplified and accurate provenance information, enabling faster forensic analysis.

4.4 Understanding KCAL

Cache Behavior. Table 3 shows summarized data for cache behaviors. It shows the name of applications (column-1), the average/maximum number of dependencies in unit cache (column-2), the average/maximum num-

Table 1: KCAL log reduction effects

	Scenario	Audit (#logs)	In-Unit Redundancy		Cross-Unit Redundancy		Temporary Files		KCAL	
			#logs	(%)	#logs	(%)	#logs	(%)	#logs	(%)
Monthly Execution	Machine 1	62,384,284	42,887,843	69%	4,594,385	7%	9,827,394	16%	5,074,662	8%
	Machine 2	137,121,400	97,384,284	71%	13,428,384	10%	12,398,283	9%	13,910,449	10%
	Machine 3	152,385,284	85,727,385	56%	15,228,384	10%	32,299,384	21%	19,130,131	13%
	Machine 4	87,837,384	18,395,394	21%	40,293,293	46%	20,923,283	24%	8,225,414	9%
	Machine 5	93,284,284	27,485,743	29%	40,293,842	43%	12,238,482	13%	13,266,217	14%
	Average	106,602,527	54,376,130	51%	22,767,658	21%	17,537,365	16%	11,921,375	11%
Apps	Firefox	6,284,385	1,128,384	18%	3,238,478	52%	1,248,284	20%	669,239	11%
	Apache	8,942,845	4,829,423	54%	2,684,284	30%	0	0%	1,429,138	16%
	Sendmail	63,284	32,493	51%	12,284	19%	16,293	26%	2,214	3%
	Vim	123,485	36,827	30%	52,284	42%	33,235	27%	1,139	1%
	MC	83,495	16,283	20%	21,384	26%	2,942	4%	42,886	51%
	Bash	20,495	2,342	11%	0	0%	0	0%	18,153	89%
	Pine	10,294	1,023	10%	8,348	81%	494	5%	429	4%
	ProFTPd	3,485,924	3,128,385	90%	100,242	3%	0	0%	257,297	7%
	Yafc	924,395	801,384	87%	39,274	4%	0	0%	83,737	9%
	Transmission	88,384	5,394	6%	80,283	91%	1,482	2%	1,225	1%
	W3M	2,485,395	423,242	17%	1,024,385	41%	743,284	30%	294,484	12%
	MiniHTTP	98,285	78,283	80%	12,384	13%	0	0%	7,618	8%

Table 2: Forensic analysis cases

Cases	Audit				BEEP				KCAL			
	#Node	#Edge	Size(MB)	Time(s)	#Node	#Edge	Size(MB)	Time(s)	#Node	#Edge	Size(MB)	Time(s)
Phishing	317	354	1905	2234	18	23	2096	142	18	23	168	16
Intrusion	860	2135	1626	30864	5	4	1888	162	5	4	226	18
InfoTheft	51	51	1148	823	7	6	1286	92	7	6	154	10
Random	412	683	2345	3349	14	32	1532	122	14	32	169	14

Table 3: KCAL cache summary (avg/max)

Application	#Deps/Unit	#Deps/Pr	#Events/Obj
Firefox	0.8/4	123/256	7.4/18
Apache	1.8/4	52/69	8.6/12
Sendmail	0.6/3	7/12	8.2/16
Vim	0.2/2	5/13	6.9/14
MC	0.2/2	6/11	7.2/11
Bash	1/1	4/7	3/6
Pine	0.3/3	8/16	9.3/16
ProFTPd	0.9/2	21/63	7.8/18
Yafc	0.8/2	42/66	8.2/14
Transmission	1.2/5	64/172	12.4/18
W3M	0.7/4	134/199	8.7/15
MiniHTTP	1.4/2	46/88	9.2/14

ber of dependencies in process cache (column-3), and the average/maximum number of events cached in a system object (column-4). From the table, it is clear that the number of dependencies in the unit cache is quite small thanks to execution partitioning. The number of dependencies in the process caches varies for different applications. In most cases, the number of dependencies is less than 200. Firefox is the only one that reaches the size limit (256) and triggers the cache replacement. For the cache in each object, the average number is less than 10 events for most programs. Even for the maximum values, the average number is still less than 32 (the cache size).

To understand the behaviors of the caches, we ran Apache and Firefox, and counted the number of dependencies they cached over time. We set the process cache bound to a large number to observe the cache pressure. Figure 13 shows the results. For Apache, we used the ab benchmark [1] with 10 concurrent clients to generate the workload. For Apache, the number of dependencies varies in a small range and remains < 70. This is because each request has just a few read/write operations on the requested file and the socket (with the client), and cached dependencies are discarded when the corresponding files and sockets are closed. For Firefox, we performed two different experiments. The first one was a normal browsing. The blue line shows the result of this experiment. Firefox uses many system objects when it loads pages. After loading the page, many dependencies can be discarded as sockets are closed and temporary files are deleted. In our test scenario, the number of dependencies (in the process dependency cache) is around 150. The other experiment used a script to open a new web page in a new tab every second. The gray line shows the results. At the beginning, each new page caused a peak, and the script opened pages more frequently than the normal user. The number of dependencies is hence larger. After 10 minutes with 500+ pages opened, Firefox reached its capacity. The number of dependencies in the cache becomes flat. Even in this extreme situation, the number of dependencies is still reasonable due to the elimination of redundant and bogus dependencies.

Kernel Memory Consumption. Figure 14 shows the maximum kernel memory footprint caused by KCAL for each application. Since our cache sizes are fixed,

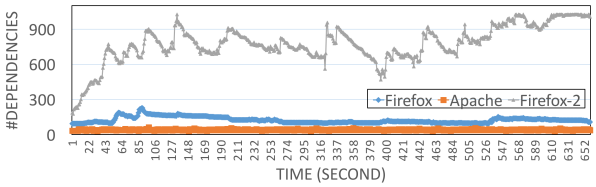

Figure 13: Number of dependencies in process dep. cache

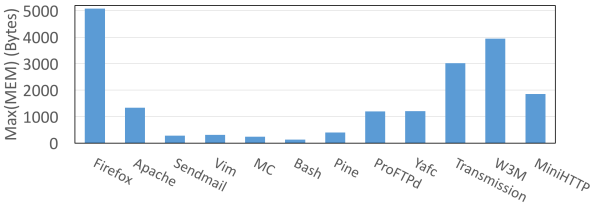

Figure 14: Max memory usage for individual applications

the memory overhead for each process including all its opened system resources (e.g., files/sockets) is fixed at 4224 bytes. In comparison, the original `task_struct` itself is 3520 bytes and it has a lot of pointers for opened system resources. One of its pointer fields `mm` pointing to a `mm_struct` is 952 bytes. Depending on the total number of system objects accessed to a process, the total memory footprint may vary a lot. However, since the number of events cached in an object tends to be small (Table 3), the kernel memory consumption is reasonably small, which ensures that KCAL does not perturb normal kernel functionalities.

Cache Size vs. Reduction Rate. The dependency cache sizes affect the reduction rate because evicting caches can result in keeping some redundant events. Previous experimental results indicate a small cache size is sufficient for many programs. In this experiment, we chose Firefox, whose dependency caches, especially the process cache, vary a lot over time, and we tested the effects of using different cache sizes. The results are presented in Figure 15. Even when the cache size is small, KCAL can still reduce many redundant events such as in-unit redundancies. With larger cache sizes, KCAL is able to detect and remove more cross-unit dependencies. If the cache is large enough (e.g., 1200 entries), all redundant dependencies are detected and the reduction rate is flat.

Comparison with State-of-the-Art ProTracer ProTracer can achieve a high reduction rate with a low runtime overhead (7% according to [24]). However, since ProTracer demands first generating the raw log before reduction, it requires parallel user-space processes to load and reduce the raw log. As a result, although its runtime overhead is low, the CPU consumption is substantial, because tainting on the large raw log files. Here we use the `ab` benchmark as an example to compare the CPU consumption of the two systems. Figure 16 shows the results.

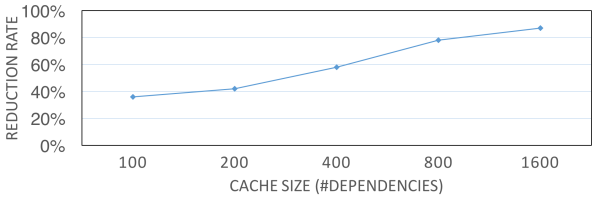

Figure 15: KCAL reduction results with different cache sizes

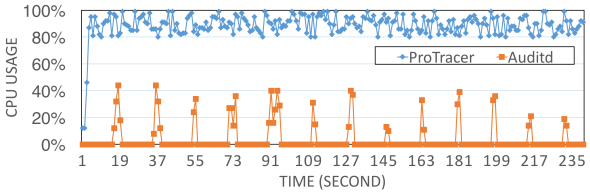

Figure 16: CPU consumption of ProTracer and KCAL

As seen in the graph, ProTracer uses the CPU consistently, and consumes almost all the cycles. In contrast, KCAL uses the CPU periodically. The average consumption is 4%. Even for the peaks, the CPU usage is about 40%, much less than ProTracer. This is because KCAL avoids generating the huge raw log in the first place and hence examines far fewer events for the same workload. In fact, ProTracer has to be pinned to a CPU to achieve the low runtime overhead. In contrast, KCAL's user-space processes just run as normal processes.

5 Conclusion

We analyzed the Linux Audit system and found the root cause of its high runtime and space overheads is its redundancy events. To solve this problem, we propose KCAL, a kernel-supported, cost-effective audit logging system for causality tracking. It caches dependencies and system events in the kernel and performs online log redundancy reduction. KCAL removes the overhead caused by transferring, processing, writing, and storing the redundant events. Our evaluation shows that KCAL can significantly reduce the log sizes and speed up the system.

6 Acknowledgements

We thank the anonymous reviewers for the valuable comments. In particular, we thank our shepherd, Howie Huang, for the constructive suggestions. This research was supported, in part, bythe United States Air Force and DARPA under contract FA8650-15-C-7562, NSF under awards1748764,1409668, and1320444, ONR under contracts N000141410468 and N000141712947, and Sandia National Lab under award1701331. Any opinions, findings and conclusionsor recommendations expressed in this materialare those of theauthors and do not necessarily reflect the views oftheUnited States Air Force and DARPA or othersponsors.

References

[1] Apache benchmark. https://httpd.apache.org/docs/2.2/programs/ab.html.

[2] Linux audit. https://people.redhat.com/sgrubb/audit/.

[3] Mozilla audit-go. https://github.com/mozilla/audit-go.

[4] Orchids. http://projects.lsv.ens-cachan.fr/orchidsdoc/.

[5] osquery. https://osquery.io/.

[6] Prelude siem. https://www.prelude-siem.org/.

[7] Sisdig. https://www.sysdig.org/.

[8] BATES, A., BUTLER, K. R., AND MOYER, T. Take only what you need: Leveraging mandatory access control policy to reduce provenance storage costs. In *7th USENIX Workshop on the Theory and Practice of Provenance (TaPP 15)* (Edinburgh, Scotland, 2015), USENIX Association.

[9] BATES, A. M., TIAN, D., BUTLER, K. R. B., AND MOYER, T. Trustworthy whole-system provenance for the Linux kernel. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.* (2015), J. Jung and T. Holz, Eds., USENIX Association, pp. 319–334.

[10] BORKIN, M. A., YEH, C. S., BOYD, M., MACKO, P., GAJOS, K. Z., SELTZER, M., AND PFISTER, H. Evaluation of filesystem provenance visualization tools. *IEEE Transactions on Visualization and Computer Graphics 19*, 12 (Dec. 2013), 2476–2485.

[11] BRAUN, U., GARFINKEL, S. L., HOLLAND, D. A., MUNISWAMY-REDDY, K., AND SELTZER, M. I. Issues in automatic provenance collection. In *Provenance and Annotation of Data, International Provenance and Annotation Workshop, IPAW 2006, Chicago, IL, USA, May 3-5, 2006, Revised Selected Papers*, L. Moreau and I. T. Foster, Eds., vol. 4145 of *Lecture Notes in Computer Science*. Springer, 2006, pp. 171–183.

[12] DEVECSERY, D., CHOW, M., DOU, X., FLINN, J., AND CHEN, P. M. Eidetic systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 525–540.

[13] GEHANI, A., AND TARIQ, D. Spade: support for provenance auditing in distributed environments. In *Proceedings of the 13th International Middleware Conference* (2012), Springer-Verlag New York, Inc., pp. 101–120.

[14] GOEL, A., PO, K., FARHADI, K., LI, Z., AND DE LARA, E. The taser intrusion recovery system. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2005), SOSP '05, ACM, pp. 163–176.

[15] GUO, Z., ZHOU, D., LIN, H., YANG, M., LONG, F., DENG, C., LIU, C., AND ZHOU, L. G2: A graph processing system for diagnosing distributed systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2011), USENIXATC'11, USENIX Association, pp. 27–27.

[16] JAKOBSSON, M., AND JUELS, A. Server-side detection of malware infection. In *Proceedings of the 2009 Workshop on New Security Paradigms Workshop* (New York, NY, USA, 2009), NSPW '09, ACM, pp. 11–22.

[17] JI, Y., LEE, S., DOWNING, E., WANG, W., FAZZINI, M., KIM, T., ORSO, A., AND LEE, W. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2017), CCS '17, ACM, pp. 377–390.

[18] KIM, T., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Intrusion recovery using selective re-execution. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 89–104.

[19] KING, S. T., AND CHEN, P. M. Backtracking intrusions. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 223–236.

[20] KING, S. T., MAO, Z. M., LUCCHETTI, D. G., AND CHEN, P. M. Enriching intrusion alerts through multi-host causality. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA* (2005), The Internet Society.

[21] LEE, K. H., ZHANG, X., AND XU, D. High accuracy attack provenance via binary-based execution partition. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013* (2013), The Internet Society.

[22] LEE, K. H., ZHANG, X., AND XU, D. Loggc: garbage collecting audit log. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013* (2013), A. Sadeghi, V. D. Gligor, and M. Yung, Eds., ACM, pp. 1005–1016.

[23] MA, S., ZHAI, J., WANG, F., LEE, K. H., ZHANG, X., AND XU, D. MPI: Multiple perspective attack investigation with semantic aware execution partitioning. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC, 2017), USENIX Association.

[24] MA, S., ZHANG, X., AND XU, D. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016* (2016), The Internet Society.

[25] MEHTA, V., BARTZIS, C., ZHU, H., CLARKE, E., AND WING, J. Ranking Attack Graphs. *9th International Symposium on Recent Advances in Intrusion Detection (RAID'06) 4219* (2006), 127–144.

[26] MOREAU, L., CLIFFORD, B., FREIRE, J., FUTRELLE, J., GIL, Y., GROTH, P., KWASNIKOWSKA, N., MILES, S., MISSIER, P., MYERS, J., PLALE, B., SIMMHAN, Y., STEPHAN, E., AND DEN BUSSCHE, J. V. The open provenance model core specification (v1.1). *Future Gener. Comput. Syst. 27*, 6 (June 2011), 743–756.

[27] MUNISWAMY-REDDY, K.-K., BRAUN, U., HOLLAND, D. A., MACKO, P., MACLEAN, D., MARGO, D., SELTZER, M., AND SMOGOR, R. Layering in provenance systems. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2009), USENIX'09, USENIX Association, pp. 10–10.

[28] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-aware storage systems. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference* (Berkeley, CA, USA, 2006), ATEC '06, USENIX Association, pp. 4–4.

[29] OU, X., BOYER, W. F., AND MCQUEEN, M. A. A scalable approach to attack graph generation. In *Proceedings of the 13th ACM conference on Computer and communications security - CCS '06* (2006), p. 336.

[30] OU, X., GOVINDAVAJHALA, S., AND APPEL, A. W. Mulval: A logic-based network security analyzer. 8–8.

[31] POHLY, D. J., MCLAUGHLIN, S., MCDANIEL, P., AND BUTLER, K. Hi-fi: Collecting high-fidelity whole-system provenance. In *Proceedings of the 28th Annual Computer Security Applications Conference* (New York, NY, USA, 2012), ACSAC '12, ACM, pp. 259–268.

[32] SAWILLA, R. E., AND OU, X. Identifying critical attack assets in dependency attack graphs. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2008), vol. 5283 LNCS, pp. 18–34.

[33] SHEYNER, O., HAINES, J., JHA, S., LIPPMANN, R., AND WING, J. M. Automated generation and analysis of attack graphs. In *Proceedings - IEEE Symposium on Security and Privacy* (2002), vol. 2002-January, pp. 273–284.

[34] SITARAMAN, S., AND VENKATESAN, S. Forensic analysis of file system intrusions using improved backtracking. In *Third IEEE International Workshop on Information Assurance (IWIA'05)* (March 2005), pp. 154–163.

[35] SUNDARARAMAN, S., SIVATHANU, G., AND ZADOK, E. Selective versioning in a secure disk system. In *Proceedings of the 17th Conference on Security Symposium* (Berkeley, CA, USA, 2008), SS'08, USENIX Association, pp. 259–274.

[36] TIAN, D. J., BATES, A., BUTLER, K. R., AND RANGASWAMI, R. Provusb: Block-level provenance-based data protection for usb storage devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 242–253.

[37] VASUDEVAN, A., QU, N., AND PERRIG, A. Xtrec: Secure real-time execution trace recording on commodity platforms. In *Proceedings of the 2011 44th Hawaii International Conference on System Sciences* (Washington, DC, USA, 2011), HICSS '11, IEEE Computer Society, pp. 1–10.

[38] XIE, Y., FENG, D., TAN, Z., CHEN, L., MUNISWAMY-REDDY, K.-K., LI, Y., AND LONG, D. D. A hybrid approach for efficient provenance storage. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management* (New York, NY, USA, 2012), CIKM '12, ACM, pp. 1752–1756.

[39] XIE, Y., MUNISWAMY-REDDY, K., LONG, D. D. E., AMER, A., FENG, D., AND TAN, Z. Compressing provenance graphs. In *3rd Workshop on the Theory and Practice of Provenance, TaPP'11, Heraklion, Crete, Greece, June 20-21, 2011* (2011), P. Buneman and J. Freire, Eds., USENIX Association.

[40] XIE, Y., MUNISWAMY-REDDY, K.-K., FENG, D., LI, Y., AND LONG, D. D. E. Evaluation of a hybrid approach for efficient provenance storage. *Trans. Storage 9*, 4 (Nov. 2013), 14:1–14:29.

[41] XU, Z., WU, Z., LI, Z., JEE, K., RHEE, J., XIAO, X., XU, F., WANG, H., AND JIANG, G. High fidelity data reduction for big data security dependency analyses. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 504–516.

[42] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in histar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 19–19.

[43] ZHU, N., AND CHIUEH, T.-C. Design, implementation, and evaluation of repairable file service. In *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.* (June 2003), pp. 217–226.