*e*Audit: A Fast, Scalable and Deployable Audit Data Collection System*

R. Sekar, Hanke Kimm, and Rohit Aich
Stony Brook University, NY, USA.
{*sekar,hkimm,raich*}*@cs.stonybrook.edu*

***Abstract*—Today's advanced cyber attack campaigns can often bypass all existing protections. The primary defense against them is after-the-fact detection, followed by a forensic analysis to understand their impact. Such an analysis requires *audit logs* (also called provenance logs) that faithfully capture all activities and data flows on each host. While the Linux auditing daemon (auditd) and sysdig are the most popular tools for audit data collection, a number of other systems, authored by researchers and practitioners, are also available. Through a motivating experimental study, we show that these systems impose high overheads, slowing workloads by $2\times$ to $8\times$; lose a majority of events under sustained workloads; and are vulnerable to log tampering that erases log entries before they are committed to persistent storage. We present a new approach that overcomes these challenges. By relying on the extended Berkeley Packet Filter (eBPF) framework built into recent Linux versions, we avoid changes to the kernel code, and hence our data collector works out of the box on most Linux distributions. We present new design, tuning and optimization techniques that enables our system to sustain workloads that are an order of magnitude more intense than those causing major data loss with existing systems. Moreover, our system incurs only a fraction of the overhead of previous systems, while considerably reducing data volumes, and shrinking the log tampering window by $\sim 100\times$.**

1. Introduction

We are in an era of long-running cyber attack campaigns involving "Advanced and Persistent Threats" (APTs) [65], [9]. Carried out by sophisticated actors that prioritize stealth over other goals, these campaigns often remain undetected for many months [75], [22], [76], [7], [71]. During this period, attackers remain hidden in a victim's network, while moving across hosts, installing malware, and gathering data.

Because of the stealthy nature of APTs, the primary recourse against them is after-the-fact detection, followed by forensic analysis to understand their full impact. Such an analysis requires logs that faithfully capture all important system activity across the hosts in an enterprise. While application logs (e.g., web server logs) are useful, they are incomplete because they don't cover the activities of all applications on the system. For instance, there are no application logs that cover malware activity, or the activities carried out in a remote login session by an attacker. Hence it is necessary to collect system-wide logs that cover all applications. Most recent research works on APT investigation [40], [38], [31], [70], [41], [91], [100], [98], [23], [101], [36], [8], [60], [13], [79] rely on *system audit logs* that operate roughly at the level of system calls. Coarse-grained *provenance tracking* enabled by these logs ensures the completeness of forensic analysis, i.e., the analysis won't miss any of the effects of an attack.

For server systems that tend to be based on Linux, the Linux auditing daemon `auditd` is an obvious choice for audit data collection [32], [55], [44]. Unfortunately, *auditd* incurs very high overheads for system-call granularity data collection, slowing down programs by $5\times$ or more. This prompted research on higher performance audit data collection systems [80], [64], [78], [62], [59]. However, these systems require OS kernel modifications, making them a challenge from a deployment perspective. Operators of production systems are reluctant to make changes to kernel code because it can impact system stability and/or introduce compatibility problems. Moreover, implementations of these log collection systems are typically tied to specific kernel versions, adding concerns about update and upgrade options.

In addition to these portability and deployability concerns, we show that existing audit data collection systems suffer from serious data loss and performance problems:

- *Data loss:* Under moderate to heavy loads, most existing systems drop a significant fraction of the events. This happens even on single-core workloads. Data loss increases proportionately with multi-core loads, leading to a situation where most of the data is dropped. This defeats the primary purpose of audit data collection, namely, ensuring that every link between attack activity and subsequent effects is captured.
- *High overhead:* Even when operating at loads they can cope with, today's log collection systems incur high overheads, slowing down workloads by $2\times$ to $8\times$.
- *Vulnerability to log data tampering [77]:* In APT attacks, attackers can gain sufficient privilege to tamper with the auditing system. The typical defense is to log the data to a remote host beyond the attacker's reach. If data is immediately sent to the remote host, evidence of attack will be preserved in the log even if the attacker is able to control all records logged after attack completion. Unfortunately, we show that existing systems can buffer hundreds of thousands of log records in memory before the data is output. A successful attack can wipe these buffers, thus removing all attack evidence from the logs.
- *Large data volume:* Existing systems produce logs ranging from several GBs to hundreds of GBs *per* host *per* day. The costs of storing this much data can be prohibitive since many APT campaigns are uncovered months later [75], [22], [76].

In this paper, we present a new audit log collection approach

*This work was supported by NSF grants 1918667 and 2153056.

that overcomes these challenges. It has been implemented into a light-weight and easily deployable system called *e*Audit. Below, we summarize our approach and contributions.

1.1. Approach Overview, Results and Contributions

*e*Audit is based on the *Extended Berkeley Packet Filter (eBPF)* framework built into recent Linux versions. This framework enables the deployment of *safe probes* at various hooks defined in the Linux kernel, such as the Linux *tracepoints* and LSM (Linux Security Module) hooks. Probes use a restricted interface and are statically verified for safety properties such as termination and memory safety. Consequently, they cannot crash the operating system, or use excessive amounts of CPU or other resources. eBPF probes can be dynamically loaded into stock Linux kernels such as those packaged with most Linux distributions. Consequently, *e*Audit can be readily deployed on these kernels without loading kernel modules, or changing the kernel code. Consequently, *e*Audit works "as is" on most recent Linux distributions. Our main contributions are:

- *Performance study of existing tools:* Our motivating study in Sec. 2 includes two widely available software tools `auditd` [35] and *sysdig* [50]; two research systems for which working software is available, namely, *CamFlow* [78] and PROVBPF [59]; and two more eBPF-based tools *Tracee* [83], [84] and *Tetragon* [18] studied in Sec. 4. We show that all these systems:
  - incur high performance overheads, slowing down systems by 2x to 8x,
  - can drop a large fraction of events, and
  - store events in memory for substantial periods, making it easier for attackers to wipe out suspicious activities before they are logged permanently.
- *Design of an efficient and resilient audit collection system:* In Sec. 3, we describe *e*Audit design, focusing specifically on features for avoiding data loss, reducing runtime overhead, and minimizing opportunities for log tampering. Specifically, we present:
  - a compact data encoding scheme that results in log files that are $10\times$ smaller than those of other systems;
  - a two-level buffer design that reduces contention and avoids data loss;
  - a simple analytical model that underpins an optimal balancing of system throughput and latency; and
  - a granular and tunable event prioritization scheme that further reduces log tampering opportunities.

  Our techniques can be applied to other eBPF-based systems as well, and improve their performance.
- *Experimental evaluation:* This evaluation establishes several key benefits of our design:
  - *e*Audit avoids data loss even on peak loads that cause the best previous systems to lose over 90% of the data.
  - Our two-level buffer design and parameter tuning optimizations are very effective, reducing overheads by an average of $18.4\times$ across our benchmarks (Fig. 17).
  - With the benchmarks and metrics used in our motivational study, *e*Audit's overhead is just 4.5% (Fig. 20).
  - Our design and optimizations reduce the log tamper window by about $100\times$ over previous systems. Our event prioritization scheme shrinks this window by another $100\times$ for the most important system calls.

The source code for *e*Audit can be found at https://eprov.org and our lab website http://seclab.cs.stonybrook.edu/download.

2. Motivating Experimental Study

We motivate our research with an experimental study that shows the drawbacks of existing systems: lost events (Sec. 2.1), high overhead (Sec. 2.2), large tamper window (Sec. 2.3) and high data volume (Sec. 2.4).

Our study includes (a) two major software systems that are in wide use, namely, the Linux audit daemon (`auditd`) and sysdig [50]; and (b) two research prototypes for which we could obtain working systems, namely, PROVBPF and CamFlow. We omitted older research systems such as PASS [72], HiFi [80] and LPM [11] because they are based on Linux kernels from 10+ years ago, making it nontrivial to get them to work with today's Linux distributions, or to draw meaningful performance comparisons with other systems that run on today's Linux kernels. PROTRACER [64] and KCAL [62] were also omitted because of the unavailability of their code. Finally, we did not consider fine-grained provenance collection systems such as RAIN [45] since they prioritize precision even if it decreases performance. However, note that many coarse- and fine-grained provenance collection systems, including TRACE [44], Spade [32], Winnower [91], MCI [54], MPI [63], BEEP [55] and ALchemist [99], rely on `auditd`, and hence inherit all of `auditd`'s performance challenges shown below.

To the extent feasible, we configured all four systems to collect roughly the same provenance information. We followed the documentation that came with the respective systems, and followed the best practices suggested in online resources. *Every available performance-relevant configuration parameter was tried out, and we used the settings that produced the best performance.* The only thing we insisted on is full provenance collection for all processes.

PROVBPF and CamFlow are both intended to capture whole system provenance, so they don't need further configuration beyond turning them on. In contrast, `auditd` and sysdig both need to be configured to log specific system calls of interest to us. We configured them to log all system calls relevant for coarse-grained provenance, including all operations for reading/writing (or sending/receiving) data. Since these system calls use file descriptors, it is also necessary to log operations that create or modify file descriptors. Operations for creating, modifying or changing the privileges of processes, and those for modifying file names and permissions are also recorded. The full list of logged calls is shown in Table 1.

Some of these systems (e.g., sysdig) can produce outputs in a binary format while others support only a printable for-

accept accept4 bind chdir chmod clone clone3 close connect creat dup dup2 dup3 execve execveat exit exit_group fchdir fchmod fchmodat finit_module fork ftruncate getpeername init_module kill link linkat mkdir mkdirat mknod mknodat mmap mprotect open openat pipe pipe2 pread pread64 preadv ptrace pwrite pwrite64 pwritev read readv recvfrom recvmmsg recvmsg rename renameat renameat2 rmdir sendmmsg sendmsg sendto setfsgid setfsuid setgid setregid setresgid setresuid setreuid setuid socket socketpair splice symlink symlinkat tee tgkill tkill truncate unlink unlinkat vfork vmsplice write writev

Table 1: List of system calls logged. This list of 80 calls is a superset of that of TRACE [44] (66 calls), the most mature and comprehensive provenance-collection system for Linux. All syscall arguments are logged, except for the data buffer argument to `read`, `write`, etc.

mat. For consistency, we configured them all to use a printable output format, logging to a normal file in `/var/log`. Experimental platform was matched as closely as possible.

Our experimental platform is a Ubuntu 22.04 system with i7-6500U processor (2 cores/4 threads), 8GB memory and 1TB SSD. `Auditd` and sysdig were run natively on this hardware in order to avoid virtualization overheads and to minimize factors that lead to variability in time measurements. However, ProvBPF and CamFlow require Fedora and are tied to specific kernel versions; so we run them within a VirtualBox VM running on the same hardware.

We used two benchmarks: *postmark* [47], a widely used file system benchmark that simulates the behavior of a mail server; and *shbm*, a script that repeatedly forks another binary (/bin/echo). The latter is designed to model the most common behavior of shell scripts and exercise common process-related syscalls. We chose these relatively simple benchmarks because we need to know in advance the numbers and types of system calls made by them. This is necessary to determine the fraction of calls captured by a provenance system. All benchmarks are single threaded to avoid overloading these logging systems. For the same reason, we used a 2-core (4 threads) platform for this evaluation.

2.1. Lost Events

CamFlow and ProvBPF**.** These systems capture provenance in terms of LSM events, which are somewhat lower level than system calls. As a result, not all system calls can be directly matched with their log records. So we focused on two categories of system calls most frequently used in these benchmarks that have identifiable records in the logs produced by CamFlow and ProvBPF. Specifically, we counted the fraction of `execve` system calls captured for the shbm benchmark, and file creation/deletion calls for postmark.

The left-most point in Fig. 2 shows the fraction of data lost by these systems on unmodified postmark. Both CamFlow and ProvBPF drop most events (about 66% and 98% respectively) at this point. To determine if the loss occurs due to the inability of these systems to keep up with the benchmark, we inserted delays (`nanosleep`'s) in postmark's main loop. The X-axis shows the factor of slowdown in the benchmark's speed due to these delays. Slowdown factor is the ratio of the runtime (wall-clock time) of the slowed down benchmark to that of the unmodified benchmark. Data loss decreases gradually when the benchmark is slowed down, confirming that the main factor is the rate of system calls.

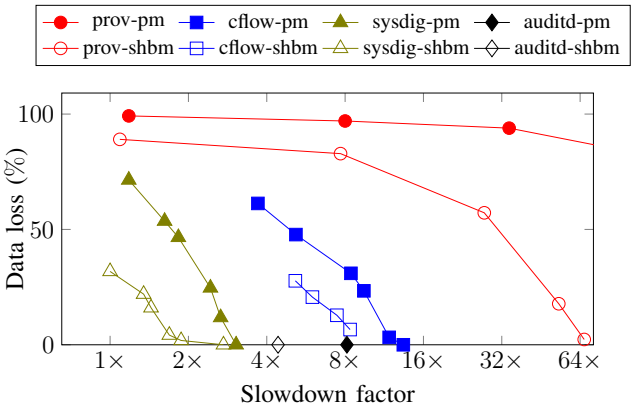

Fig. 2: Data Loss: Fraction of events dropped by existing provenance collection systems. The base for calculating the slowdown factor is the benchmark runtime in the absence of provenance collection. The left-most point on each line shows the data loss on unmodified benchmarks. In the legend, "pm" stands for postmark, "prov" for ProvBPF, and "cflow" for CamFlow.

CamFlow requires an order of magnitude slowdown before it captures all the events. A tell-tale indicator in this context is the CPU usage of `camflowd`, the user-level process used for data logging in CamFlow. When it hits 100%, records begin to get dropped. This mark was hit until we slowed down benchmarks by 8× (shbm) to 16× (postmark).

ProvBPF needs a larger slowdown (by about two orders of magnitude) before it captures a significant fraction of data. At that point, it becomes difficult to separate the overhead due to the benchmark from that of background system activity. For this reason, we omitted ProvBPF from the remaining experiments in this section.

Sysdig and `auditd`**.** These systems track system calls directly, so it is easy to check the number of events against the expected number. For sysdig, we inserted delays in the benchmark as described before, and plotted data loss against slowdown in Fig. 2. It requires a slow down by about 3× before it can capture all data. Auditd does not require delays, as it seems capable of slowing down benchmarks until it can keep up with them. Thus, its performance is represented by a single point on the X-axis that indicates zero data loss with about 5× slowdown for shbm and 8× for postmark.

2.2. CPU utilization and overhead

Existing provenance collection techniques also suffer from high overheads and CPU usage. Note that performance measurements are meaningless when the system under study isn't operating correctly, i.e., a significant fraction of events are being dropped. So, we once again slow down benchmarks by inserting delays as before. These delays were increased until data loss fell below 10%. We also ensured that at least 95% of these events were due to the benchmark, with background activities accounting for less than 5%.

We measure overhead as the ratio of the CPU time used by the provenance collection system to that of the benchmark. We use CPU time (user plus system time reported by the operating system) instead of wall-clock time because it is unaffected by the delays introduced in the benchmark, or the times when processes are idle.

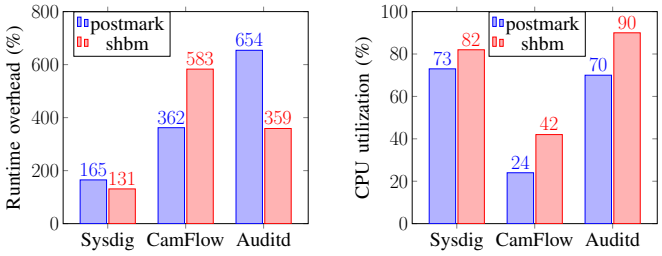

Fig. 3: Runtime overhead and CPU utilization: Benchmarks were first slowed down by inserting sleeps to ensure a data loss rate below 10%. Overhead is defined as t_a/t_b, where t_a is CPU time of the provenance collection system ("agent time") — specifically, the CPU time of `camflowd` in the case of CamFlow, `sysdig` in the case of sysdig, and `auditd` and `kauditd` for Linux auditd; and t_b is the CPU time of the benchmarks (with embedded sleep's) in the absence of provenance collection.

Our results are shown in Fig. 3. Note that the overheads are high, slowing down programs by $4\times$ to $16\times$ *even on single-threaded benchmarks* studied in this section.[1]

2.3. Log tampering window

Log contents in persistent storage can be protected from an attacker by logging the data to a (protected) remote server. However, log records that are still in memory buffers at the time of a successful exploit are subject to tampering by an attacker. In the simplest case, the attacker can immediately kill the logger, causing all those entries to be lost. Exploits that achieve the privilege needed for tampering may constitute a minority in general, but they are commonplace in APT campaigns. *Moreover, post-attack detection is aimed precisely at this class of adversaries — those powerful enough to break through every deployed defense and hence can only be detected after the fact.*

To measure the log tampering window, the benchmark was killed at a randomly chosen time. The length of the log was immediately recorded. These systems continued to write records into the log file — these must be the records buffered in memory, since the benchmark had already terminated. The end of this phase was determined by a sharp drop in the CPU usage of the data collection system, further confirmed by a sudden drop in the growth of the log. The length of the log file was recorded again. We count the number of records that fall between the two length measurements, and average this number across 10 repetitions.

As shown in Fig. 4, the log tampering windows for these systems are of the order of several tens to hundreds of thousands of records. In terms of time, this approaches the range of a second or so, making it possible for automated exploits to erase critical activity surrounding the break-in.

[1] If we ran postmark without delays and measured the overhead in terms of benchmark wall-clock time, then CamFlow's overhead roughly matched that reported in [78]. However, this number does not truly reflect CamFlow's overhead as it is dropping the vast majority of events at this point.

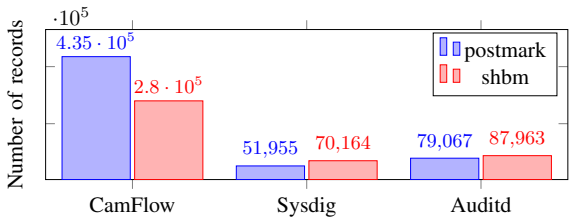

Fig. 4: Log tampering windows of CamFlow, sysdig and auditd.

2.4. Data Volume

Another major drawback of existing provenance collection frameworks is data volume. A 3-minute run of the postmark benchmark produces 9GB for sysdig, 26 GB for Linux auditd, and 41 GB for CamFlow. If we take the smallest of these numbers, and further assume that average event rates will only be a tenth of this rate, the number still adds up to over 400GB per day per host.

3. *e*Audit System Design

The main goals of our design are:

- *Reduce data volume* so as to speed up every component involved in the data pipeline;
- *Eliminate data loss* even at peak system loads;
- *Reduce overhead* so as to minimize degradation of peak workloads that can be sustained; and
- *Reduce latency* of data capture so that records are logged to safe storage quickly, minimizing the opportunities for an adversary to tamper with these records.

We begin by introducing our threat model and how it guides our choice of benchmarks in Sec. 3.1. Following this in Sec. 3.2 is a short overview of eBPF that is necessary for understanding *e*Audit design. Next, we describe *e*Audit architecture in Sec. 3.3, followed by techniques for achieving the above goals. Specifically, Sec. 3.4 describes a compact data encoding scheme to achieve the first goal. We then address the next three goals, which are conflicting in nature: techniques for reducing data loss, such as the use of large buffers, tend to increase the tamper window size. Similarly, throughput increase techniques tend to increase latencies as well. We develop a two-level buffer design in Sec. 3.5 to mitigate these conflicts. We formulate an optimization problem to achieve an ideal combination of throughput and latency in Sec. 3.6. Finally, we describe event prioritization to further reduce latencies in Sec. 3.7.

3.1. Goals, Threat Model and Benchmark Selection

The primary goal of this paper is to develop techniques for efficient provenance collection — techniques that do not degrade the workloads that can be sustained on a target system. As we showed in the previous section, existing approaches increase CPU workload by more than 100% for provenance collection. If logging imposes significant overheads, operators may respond by turning it off during sustained or peak system activity. This, in turn, may be exploited by attackers to hide their activities. First, attackers may time their attacks to coincide with periods of high loads on their victim systems. Secondly, attackers may themselves generate large workloads that resemble benign activity. By removing performance overhead as a significant concern, we can take away this avenue for attacks.

The second goal is to thwart attacks that exploit the tendency of existing provenance collection systems to lose

Name	Description
postmark	file system benchmark [47] simulating a mail server
shbm	a shell script that repeatedly executes `/bin/echo`
find	uses `find` command to print all file names in `/usr`.
tar	uses `tar` to archive `/usr/lib`.
rdwr	a C-program that calls `read`, `write` in a tight loop.
httperf	a benchmark for web servers.
kernel	compile the Linux kernel.

Table 5: Benchmarks used in this paper.

records during intense loads. As a result, even if provenance collection is permanently on, attackers may still be able to hide their activities during peak loads: since a significant fraction of records are being dropped, much of the attacker activity may not be present in the logs. This provides a second avenue for evasion attacks that rely on high loads.

To address these avenues of evasion, we aim to build a provenance collection system that can operate without lost records or significant performance overheads *even at the peak system loads that can be sustained by the underlying OS and hardware.* Our choice of benchmarks (Fig. 5) reflects this goal. For instance, multiple instances of `shbm` can easily max out the rate at which the OS is able to execute programs. Similarly, `rdwr` can achieve the peak rate of reads and writes, while `find`, `tar` and `postmark` can stress file-system throughput of the OS. Finally, we added `httperf` and `kernel`, two well known benchmarks for network and CPU-related loads.

Our last goal is to degrade the ability of attackers to hide their activities through log tampering. Adversaries may attempt to maximize their chances by actively creating a huge backlog in the queues holding provenance records that haven't yet been written to (secure) storage. Alternatively, they may time their attacks to take place during periods when the backlog is expected to be large. A large backlog gives the attacker sufficient time to break into the victim, escalate their privilege, and then delete the records in the queue before attack related activity is written out. Our goal is to shrink this backlog to the point where successful log tampering becomes extremely difficult.

Except for the kernel benchmark that can already use all available cores, the rest of the benchmarks are parallelized by adding a top-level loop that creates a specified number of processes that each run a copy of the benchmark. This allows us to create configurable multi-core loads. We ran these benchmarks on the hardware platform used for experiments in Sec. 2: an i7-6500U processor with 2 cores (4 threads), 8GB memory and 1TB SSD. We also added a second platform with more recent hardware: an i7-12700 processor with 12 cores (hyperthreading disabled), 16GB memory and a 512GB SSD. Henceforth, we use both these platforms as a way to verify that our results are generalizable.

3.2. Background: Overview of eBPF

This section provides a short overview of the eBPF framework, focusing on the features core to our system design. The eBPF framework enables small code snippets, called *ebpf probes,* to be *safely* deployed at well-defined hooks within the Linux kernel. Currently supported hooks include the Linux *tracepoints*, *kprobes*, LSM hooks, and so on. When the kernel control flow reaches a hook, the callback function registered by a probe for that hook is invoked. While some of these hooking points (e.g., tracepoints) are enabled out-of-the-box on most Linux distributions, others (e.g., LSM hooks on Ubuntu) require rebuilding the kernel with non-default options. In operational settings, such non-default kernels may not be an option, so *e*Audit uses only the tracepoint hooks for system call entry and exit.

The eBPF runtime defines a *virtual machine* supporting a virtual instruction set for writing probes. Programmers typically write their probe code in a C-subset, which is then compiled into eBPF instructions. Before the probes are loaded, the Linux kernel verifies several properties such as memory safety and absence of loops.[2] These checks ensure that probes cannot crash the kernel or unduly slow it down. Finally, the virtual instructions are JIT-compiled into native code and loaded into the kernel.

Probes can use a (very) small set of helper functions that have been carefully designed to minimize risk. One set of helper functions is for safely reading memory (user or kernel). A function for writing memory is also provided, but its use is discouraged by printing a warning message on the console on each use. As a result, most probes (including all tracepoint probes) have a "read only" behavior.[3]

A second set of helper functions provide a key-value store called *eBPF maps*. Since static variables are not permitted by the verifier, eBPF maps provide the sole mechanism for maintaining any state across callbacks into probes. Maps can be per-cpu or shared across all CPUs. There is one instance of each per-cpu map on each core, which means that the same probe sees distinct instances of these maps on different cores. Operations on shared maps incorporate concurrency control, so they are not as efficient as the per-cpu maps that can be safely accessed without such control.

A third set of helper functions support communication with user-level applications. Indeed, the purpose of an eBPF probe is to intercept selected operations and send the observed data to a user-level "consumer." *Perf buffers* are the older mechanism for communication with the user-level, while the more recently introduced *ring buffer* is recommended for better performance [73]. We discuss them in more detail in Sec. 3.5.

While the Linux kernel implements all of the core features of eBPF, the interface provided by it is rather low level, requiring applications to work with eBPF byte code. To ease development, an LLVM-based compiler is available for translating probes written in a higher level language (a restricted subset of C) into eBPF bytecode. Then there are many toolkits and libraries such as bcc [12], bpftrace [2] and libbpf [58] that simplify other low-level aspects such as the parsing of ELF binaries, invoking the eBPF loader,

[2]Loops with a statically known bound can be unrolled manually, or using compiler pragmas. However, this "syntactic sugar" doesn't increase the expressive power of eBPF code, as the kernel loads only loop-free code.

[3]LSM probes do have the ability to deny operations, so errors in LSM probes have a much greater capacity to break the system — a likely reason why LSM eBPF hooks are disabled on Linux distributions such as Ubuntu.

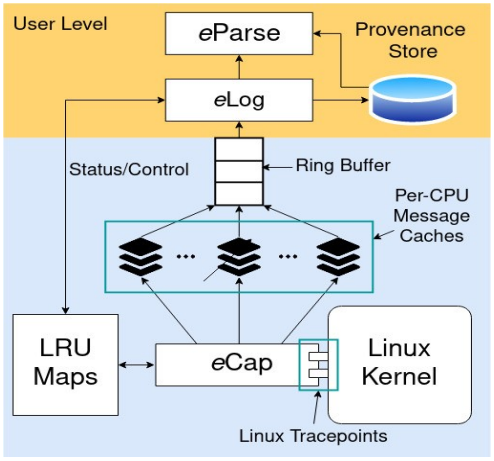

Fig. 6: *e*Audit Architecture.

setting up and accessing eBPF maps from the user level, and so on. BCC [12] is (arguably) the most mature among these, so *e*Audit implementation currently uses this toolkit. For more details on eBPF, see [21], [81], [34], [15], [29].

3.3. *e*Audit Architecture

Fig. 6 depicts the architecture of *e*Audit. It consists of a component *e*Cap that operates in the kernel, and two user-level components *e*Log and *e*Parse for logging and parsing/printing respectively. Each of these components is described in more detail below.

*e*Cap is concerned with data capture in the kernel. It attaches probes at the Linux Kernel Tracepoints interface [48], specifically those associated provenance-related system calls listed in Table 1. Each probe is a function with the signature specified in the sysfs pseudo file system at `/sys/kernel/debug/tracing/events/syscalls/⟨scevent⟩/format`, where ⟨scevent⟩ stands for the entry or exit of a specific system call, e.g., `sys_enter_execve`. System call arguments can be accessed in the `enter` events, while return values can be accessed at the `exit` events. System call information, including argument and return values, is first serialized and stored in per-CPU buffers that we call as *message caches*. When one of these caches is filled, it is written to the (shared) ring buffer.

*e*Cap uses eBPF maps to maintain state as a set of key-value pairs. Resource leaks, in the form of allocated state that is no longer used, pose a serious problem for long-running software such as *e*Audit. Unfortunately, leak detection algorithms generally require loops, which are not permitted in eBPF. Least Recently Used (LRU) maps provide an elegant mechanism in this context — when space is needed for new items, oldest entries are automatically purged.

*e*Log is the user-level component that reads the data sent by *e*Cap and immediately logs it to provenance store. The provenance store can be located on a remote machine that is locked down, but for simplicity and consistency with the other tools we have compared, our current implementation uses a local file. Because our implementation relies on BCC [12], *e*Log uses a Python program for loading eBPF probes into the kernel, and to access a subset of the maps that are used for querying the status of the probes or configuring them. However, in order to maximize performance, the ring buffer handler and the rest of *e*Log are written in C. This means that the critical path in the user-level code avoids possible performance bottlenecks in Python code.

*e*Parse is a user-level component for parsing and printing the (binary) data from *e*Cap in a readable format, or in an architecture-neutral format that is suitable for intrusion detection and forensic analysis. *e*Parse can be chained on top of *e*Log to parse the data in real-time, or operate offline using data from the provenance store.

3.4. Compact Data Encoding

An obvious strategy for reducing data volumes is to develop compact data representations. Previous research tends to downplay the importance of this step by emphasizing that compaction can be performed later on, after initial data collection. Yet, as our results show, the volume of provenance data overwhelms the system, causing data to be lost even before it reaches post-processing. This point was driven home by our performance studies on recent Fedora distributions: since these distributions turn on file compression by default, file system performance is reduced, which translates to a much higher data loss rate for existing provenance collection systems. For this reason, *e*Cap starts with a compact encoding.

Some of the basic elements of our compact encoding scheme are: (a) encoding each event using a single byte, (b) suppressing thread id information for single-threaded processes, and (c) including only the least significant 3-bytes of the timestamp (in nanoseconds) with every event. A separate timestamp record is emitted when the leading 5 bytes of the timestamp change, i.e., every 16 milliseconds. Argument number and type information is not included in event records since it can be inferred from the event name.

Many integer arguments have small values, e.g., file descriptors and system call return values (which are often zeroes). We use a variable length encoding for integer arguments, using 1, 2, 4 or 8 bytes, depending on the actual value of the argument. This length information is encoded into a single byte, which is sufficient to represent up to 3 arguments plus a return value. System calls with more arguments use 8 bytes for the remaining arguments.

File name and socket addresses use a variable-size representation, with a prefix byte encoding their length. For `execve`, which is unusual in taking a variable number of arguments, the argument number is encoded in the record. Our implementation currently limits the number of arguments and environment variables to a maximum of 32 — attempts to record more arguments leads to a permission denial by the eBPF verifier. We can use eBPF tails calls to overcome this limit, but this is left for future work. (Note that this limit is never reached in our benchmarks, so our performance numbers and comparisons are unaffected.)

A side benefit of compact encoding is that we can record the entry and exit events separately, without being overly concerned about log size. Producing a single combined

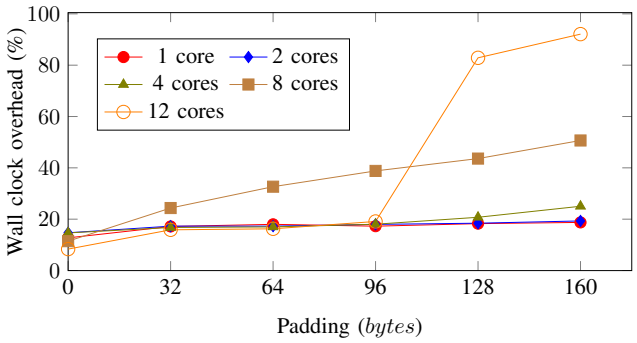

Fig. 7: Impact of compact encoding on *e*Audit overhead.

record can seem more efficient, but by recording them separately, we can reduce latency, i.e., the time before the information is stored securely. This can be particularly important for system calls that have significant security implications such as `setuid`, `kill` and `execve`. For system calls that occur frequently and/or have less security implications, we capture a combined record at system call exits.

Due to this encoding, our system call records are 17 bytes on average (postmark benchmark), as compared with 175 bytes for `sysdig` and 850 bytes for `auditd`. Note that *e*Audit as well as `sysdig` and `auditd` provide the same information about syscalls — all argument values are available, except for the data buffer argument to syscalls that read or write data.

Effectiveness. To evaluate the impact of our compact encoding, we measured the performance overhead of *e*Audit as a function of the syscall record size. We used the `postmark` benchmark. Record size was increased by adding a specified number of padding bytes. The number of padding bytes was varied from 0 to 160. This causes the total record size to vary from the average size produced by *e*Audit to the average size produced by `sysdig`. Note that the overhead is also affected by the the parameters p and w discussed in the next section. We set them to the same (optimal) values used in Sec. 4.

Fig. 7 shows that the wall clock overhead, defined as the percentage increase in the completion time of a benchmark, goes up roughly linearly with syscall record size. This increase is less than 50% for 1- and 2-core loads, but increases rapidly to $4\times$ and $11\times$ for 8-core and 12-core loads. At padding sizes of 128 and 160 bytes, the 12-core load overwhelms the system, leading to an abrupt jump in the overhead, as well as some dropped data.

3.5. Two-level buffer design to avoid data loss

As noted earlier, eBPF provides two mechanisms for communication between the in-kernel data probes and the user-level consumer process: *perf* and *ring* buffers. Ring buffer is the more recent mechanism, introduced to address some of the drawbacks of perf buffers [73]. In particular, the ring buffer uses a single large buffer across all CPU cores, whereas perf uses N buffers for N cores. A single large buffer tolerates data volume spikes much better than N buffers that are each $1/N$th of this size. Moreover, because of its focus on performance, the ring buffer API includes features to reduce unnecessary data copying, and explicit control over how often the user-level consumer is signaled (using the poll/epoll mechanisms) about data availability.

Despite its performance focused design, we found that a straight-forward use of the ring buffer API is insufficient for lossless provenance collection. We therefore developed a two level buffering scheme to reduce the number of accesses to the ring buffer. In this scheme, system call records are first assembled in a per-CPU "message cache." When the cache becomes full, its content ("message") is queued on the ring buffer.[4][5] Although this approach introduces an extra data copy (from the message cache to the ring buffer), this is unavoidable: the reserve/commit API, which is used to construct ring buffer messages in place, requires message sizes to be *compile-time constants* [73]. Events that produce nontrivial data are all associated with file names or other variable size data. Hence we must rely on the `ringbuf_output` that requires the data to be assembled in a temporary buffer and then copied over into the ring buffer.

The following table outlines the key parameters that affect the performance of this two-level buffer design:

p	the size of the per-CPU message cache, measured in terms of number of system call records.
r	the size of the ring buffer.
w	wake-up interval — one in every w operations on the ring buffer will signal data availability to the user-level.
N	the number of CPU cores.

Among these four parameters, we found that p has a much larger impact on data loss than w. Increasing r reduces data loss for short-running benchmarks but not longer-running ones. In particular, entries in the ring buffer begin to shoot up when the system is unable to process events at the rate at which they are produced. This imbalance causes even a large ring buffer to fill up eventually, at which point data is dropped.

This leaves us with two main parameters that affect data loss, namely, p and N (the number of cores used by the benchmark). Fig. 8 focuses on a single benchmark *find* and shows the effect of p and N on data loss. When the workload uses one or two CPU cores, data loss can be avoided for all values of p. For this reason, data points for 1-core and 2-core workloads line up along the X-axis. But as the number of cores used by the workload is increased, larger message cache sizes are needed. For the maximum configuration studied, 12 native cores on an i7-12700 processor, $p = 10$ was needed to eliminate data loss.

[4]Note that message caches are small — a few KBs — as compared to the ring buffer that is several MBs in size. Thus, almost all the kernel memory used by *e*Audit is in the ring buffer.

[5]Although their per-CPU nature may suggest a similarity between message caches and *perf* buffers, that is not the case. Per-CPU caches require no synchronization whatsoever. In contrast, perf-buffers are a mechanism for communication with the user level, and have the same signaling and wake-up overhead associated with the ring buffer. Worse, perf API does not have an option to avoid signaling the user level on each message, and hence it incurs high overhead (specifically, t_s from Table 11) on every syscall if it were used in place of the message cache. In contrast, as we show later, our design incurs t_s once per few hundred syscalls.

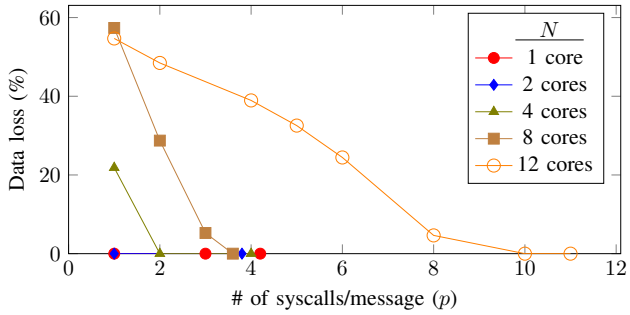

Fig. 8: Data loss Vs Message cache size (p). All curves correspond to the *find* benchmark, but differ in the number of cores used (N).

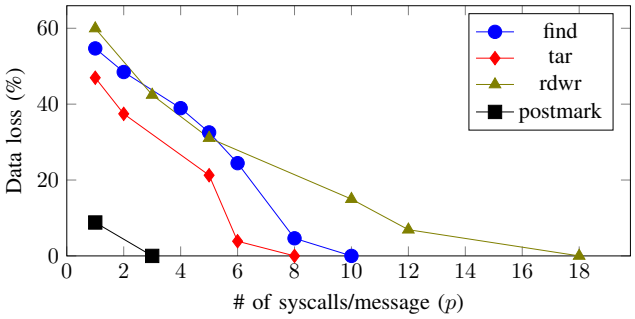

Fig. 9: Data loss Vs Message cache size (p). Curves correspond to different benchmarks, each parallelized to use all cores on a 12-core system.

Fig. 9 examines data loss from a different perspective: the benchmarks are varied, while keeping N fixed at 12. Note that values of $p \geq 18$ avoids data loss in all cases. *e*Audit uses a default value of $p = 100$, which gives a comfortable margin above this minimum value. (For these two charts, we used $w = 8$ and $r = 16MB$, values that are large enough that further increases don't affect data loss.)

Effectiveness of per-CPU message cache. In our implementation, setting $p = 1$ bypasses the per-CPU cache. Figs. 8 and 9 show that *e*Audit experiences substantial data loss (up to 60%) in the absence of CPU-message cache. As the message cache size is increased, data loss decreases gradually, falling to 0% at $p = 18$ for all benchmarks and all values of N used in our experiments.

Although data loss is avoided at $p = 18$, larger values of p provide additional benefits in terms of performance.

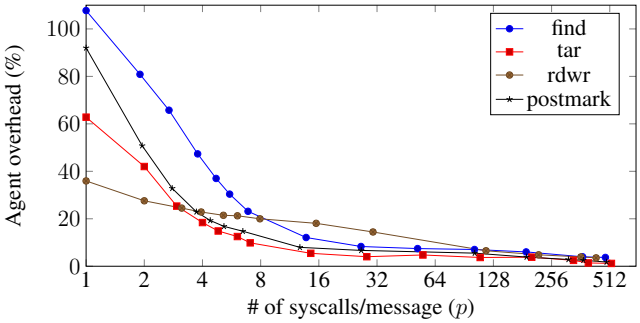

Fig. 10: *e*Audit's Agent Overhead Vs Message Cache Size (p). For $p < 18$, overhead numbers are an underestimate in cases where there is data loss.

Fig. 10 shows that the agent overhead continues to drop as p goes from 16 to 100. On average, a $2\times$ reduction in overhead is observed across the benchmarks in Fig. 10. Together, these results show that per-CPU caches are very effective in improving the performance of provenance collection. As a result, *e*Audit is able to avoid data loss on benchmarks that are more than $10\times$ as intense (in terms of syscall rate) as those sustained by existing provenance collection systems.

3.6. Optimizing Overhead Vs Latency Trade-off

In the last section, we showed that data loss can be avoided by increasing p, but larger values of p have a negative effect as well: they increase *latency,* i.e., the period during which events are buffered in memory and are hence susceptible to tampering (Sec. 2.3). A similar comment applies to the w-parameter as well: larger w values decrease overheads at the cost of increasing latency. However, the two parameters affect latency and overhead in different ways, enabling us to formulate an interesting optimization problem for deciding the right trade-off. Table 11 summarizes our formulation and the analytical solution derived from it. We describe this formulation below and present an experimental validation.

On a per-event basis, the agent's execution time comes from:

(a) the time t_b to gather and store event information in the per-CPU message cache, and to copy it around until the data is finally recorded into provenance store;

(b) the time t_q to enqueue/dequeue a message on the ring buffer, incurred once every p events; and

(c) the time t_s to wake up the user-level consumer, incurred once every w messages, i.e., every wp events.

Of this, (a) is unaffected by the buffering scheme, and represents the "base" cost for event logging. So we focus in Table 11 on the *buffering overhead* O that includes (b) and (c).

Parameters	N	number of CPUs
	p	message cache size (number of events)
	w	wake-up interval
	t_q	time to queue messages on the ring buffer
	t_s	time to context-switch (to wake up user-level agent)
Metrics	T	Agent CPU time per event: $t_b + t_q/p + t_s/w \cdot p$
	O	Buffering overhead per event: $t_q/p + t_s/w \cdot p = (t_q + t_s/w)/p$
	L	Maximum length of buffered data: $N \cdot p + p \cdot w = (N + w) \cdot p$
Goal	*Minimize* $O \cdot L = (t_q + t_s/w)(N + w)$. *Solution:* $w = \sqrt{N \cdot t_s / t_q}$ $\left(\text{by setting } \frac{d(OL)}{dw} = 0\right)$. *Note:* Objective emphasizes O and L equally. At the optimal w value, $d(O \cdot L)/dw = 0$, which is equivalent to: $\frac{1}{L}\frac{dL}{dw} = -\frac{1}{O}\frac{dO}{dw}$ In other words, the relative change in L with w will be equal and opposite of the corresponding change in O. At all other w-values, it can be shown that any change to w will improve one metric more than it degrades the other.	

Table 11: A simple performance model for tuning w to find an optimal trade-off between overhead and latency.

We measure latency L in terms of the maximum number of events that are stored in memory at any time. After reading a message from the ring buffer, *e*Log immediately writes it out to provenance store using a `write` system call.[6] For this reason, no messages are queued by *e*Log, so we need only consider the messages buffered in the kernel. This includes (i) N message caches, each of which contain at most p events, and (ii) up to w messages that may be on the ring buffer but *e*Log has not been woken up to process them. Thus $L = N \cdot p + w \cdot p = (N + w) \cdot p$.

A key benefit of our analytical performance model is that it can yield optimal solutions for a variety of conditions, e.g., if latency reduction is considered twice as important as overhead reduction. For the rest of this section, we make another natural choice, which is to weigh latency and overhead reductions equally. As shown in Table 11, this corresponds to minimizing the product of O and L. Differentiating the expression for $O \cdot L$ with respect to w, equating it to zero and solving, we get the optimal value of w to be $\sqrt{N \cdot t_s / t_q}$. Next, we proceeded to experimentally measure t_q, t_s and t_b.

To measure t_b, we first used very large values for p and w to make the impact of t_q and t_s negligible, so that the entire runtime of the agent can be attributed to t_b. Specifically, we set $p = 1000$ and $w = 256$ for this measurement. For smaller p and w values that introduce nontrivial buffering overhead, we can obtain O from the observed agent time T using the relationship $O = T - t_b$. By measuring O in this manner for different p and w values, and performing a regression analysis to fit the equation for O shown in Table 11, we obtain t_s and t_q.

Fig. 12 illustrates how well our model matches the actual measurements on our experimental platforms. For each data point, we plot the experimentally measured overhead on the X-axis and the calculated overhead on the Y-axis. Calculated overhead for a data point is obtained from the p and w values for that point, and the values of t_s and t_q from the regression analysis. The diagonal represents the ideal case, where there is a perfect match between predicted and actual overheads. Actual data points all fall around this line. Note that the fit is more accurate on the 2/4 core platform ($N = 4$) as compared to the 12-core platform ($N = 12$). This is because the 12-core platform experiences much more contention and saturation effects, making the measurements more noisy. These measurements pertain to the *find* benchmark. (Other benchmarks produce similar results.)

[6]More precisely, *e*Log performs a single `write` each time it is woken up. On each wake-up, it processes the messages already on the ring buffer (w messages on average). This happens without the possibility of blocking or waiting, and is followed by a write operation.

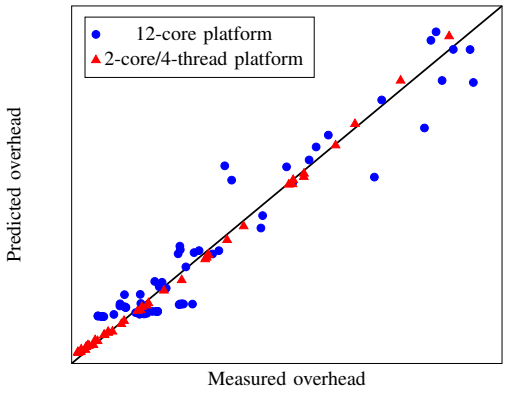

Fig. 12: Experimental Vs Predicted overhead for *e*Audit. Predicted values come from the formula for O in Table 11 and the values of t_q and t_s obtained by regression analysis. These values are $t_q = 2.26 \cdot 10^{-6}$ and $t_s = 2.06 \cdot 10^{-5}$ on the 2-core platform, and $t_q = 7.72 \cdot 10^{-7}$ and $t_s = 2.85 \cdot 10^{-6}$ on the 12-core platform.

Fig. 13 is an alternate visualization of how well the performance model matches the measurements. Specifically, the smooth curves depict the equation from the model:

$$O \cdot L = (t_q + t_s/w)(N + w)$$

As in the previous figure, t_q and t_s are obtained from regression analysis, and the red and blue curves correspond to the 2-core and 12-core platforms respectively. The dots correspond to experimentally measured points. Note that once again, the analytical model closely matches the measurements for the 2-core platform but the 12-core platform is more noisy. Nevertheless, the points follow the general shape of the smooth curves.

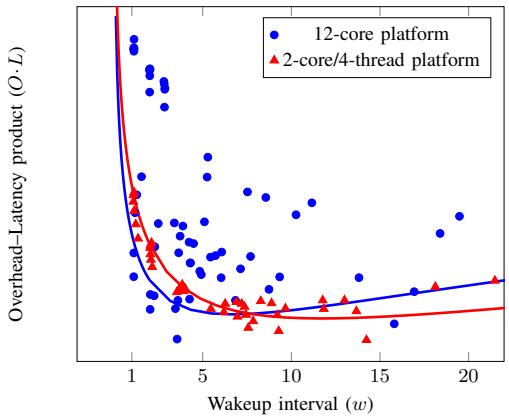

Fig. 13: Experimental Vs Predicted overhead–latency product.

Another key point illustrated by Fig. 13 is that $O \cdot L$ has a minima between $w = 5$ and $w = 10$. Fig. 14 is a more direct illustration of such an optimal value. *Note how, for a given latency (on the X-axis), the use of larger w leads to a lower benchmark overhead,* as w is increased from 1 to 7. This chart also shows that for small values of w, overhead reduction is much more significant as compared to larger w values. This is explained by our analytical model: when w goes from 1 to 2, it cuts down context-switching overhead by $2\times$. Since the context switching overhead is much larger than queuing overhead — across our experimental platform, t_s ranges between $6t_q$ and $12t_q$ — dividing t_s by w effectively halves the total overhead as w goes from 1 to 2. However, for larger w values, contribution of t_q to the total overhead becomes comparable to that t_s/w, so w is no longer very effective in reducing the total overhead. This is

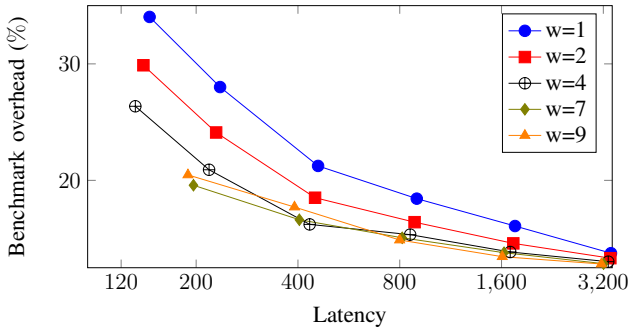

Fig. 14: Latency Vs. Benchmark overhead for different w values. Latencies below 200 are difficult to achieve for $w \geq 7$, so those curves stop at 200.

apparent in Fig. 14 as well: $w = 7$ offers little benefit over $w = 4$. In fact, the $w = 9$ line is higher than the $w = 7$ line, showing that 9 is larger than the optimal w-value.

Tuning w**.** Since the experimental results closely match the analytical model, we can use the formula for optimal w from Table 11. This requires obtaining t_s and t_q through timing measurements. However, it would be preferable if parameters can be tuned without requiring such measurements before each deployment. Below, we discuss how to do this.

The optimal value of w depends on N and the ratio of t_s and t_q. We found that processors with a larger number of cores (N) tend to have a larger t_q (queuing overhead), but t_s does not seem to be affected much by N. As a result, the increase in N and t_q tend to cancel each other out. This factor, combined with the square root operator in the formula for optimal w, causes the optimal value of w to be fairly stable across platforms and benchmarks. Specifically, our calculated optimal values for three experimental platforms were between 5 and 8. More importantly, the $O \cdot L$ curve becomes very flat between $w = 5$ and $w = 10$ across all these platforms. So, choosing any w value between 5 and 10 yields highly similar performance and latency results. This factor stands out in Fig. 14, where the lines for $w = 7$ and $w = 9$ are very close to each other. Hence, we chose $w = 8$ as the default value in *e*Audit.

Explaining the performance of 2-level buffer design. Based on our performance model (Table 11) and the measurements shown in Fig. 13 and Fig. 12, there are two main sources of overhead, namely t_q for queuing the message on the ring buffer and t_s to signal/wakeup the user-level agent waiting for the data. Focusing on the 12-core platform used for the bulk of our experimental evaluation, $t_q \approx 0.8\mu s$ and $t_s \approx 2.9\mu s$ (blue circular marks in Fig.12). Without the optimizations described in this and the previous sections, every system call will incur the overhead of t_q to queue the syscall record on the ring buffer, and t_s to wake up the user-level agent, for a total overhead of $3.7\mu s$ per syscall. This limits the maximum number of syscalls that can be handled per second to hundreds of thousands. However, multicore loads (such as those presented by our benchmarks) can issue millions of system calls per second, with the result that most of them would be dropped.

Our optimized two-level buffer design incurs t_q once every p system calls, and t_s every $p \cdot w$ system calls. With parameters tuned as described above, we can expect the overhead per system calls to drop to $0.8/100 + 2.9/800 \approx 0.01\mu s$, which means *e*Audit can potentially keep up with syscall rates of tens of millions per second. In contrast, systems that do not incorporate such optimizations can be expected to face data loss on moderate to high intensity loads, an expectation that is confirmed by our experimental evaluation of existing provenance collection systems.

3.7. System call prioritization

The techniques described in the previous two sections helps to reduce the log tampering window significantly from several tens of thousands of records for previous systems (Fig. 4) to the range of hundreds. Previous work [77] has shown that hundreds of system calls may be sufficient to carry out a log-tampering attack, so we present techniques in this section to further reduce this window.

Our approach is based on the observation that not all system calls contribute towards privilege escalation and/or log tampering. Indeed, not all system calls are equally important for attack investigation, e.g., `execve`'s are far more significant than `read`'s. Hence we develop an event prioritization scheme that further mitigates log tampering. Our classification into *critical* and *non-critical* system calls is similar to that of HARDLOG [6], and is driven by the same observation about the key system calls used in real-world exploits [67], [85], [82], APT attacks [19], as well as an analysis of privileges needed to carry out a log tampering attack. Our prioritization is also based on our research experience in detecting APT campaigns from system-call level audit data [40], [70], [41]. At the same time, it is important to note that we are not presenting a single pre-defined prioritization scheme, but instead, a framework that enables the users of our system to define these priorities in a manner that suits their needs.

Although our prioritization is driven by the same reasoning as HARDLOG, our design improves over theirs in two important ways. First, *e*Audit supports a *user-configurable* and *granular prioritization scheme* where users can assign one of 256 possible weights to each system call based on their specific needs and requirements. In contrast, HARDLOG has only two pre-defined priority levels. This means that there is no way to prioritize system calls with an intermediate level of importance, such as those for removing files, over less important ones such as reading files.

Secondly, HARDLOG requires a dedicated hardware device, and moreover, makes significant changes to the kernel code in order to avoid sending events to the user level. In contrast, *e*Audit is a software-only solution that requires no kernel changes. While it does not achieve the zero tamper window of HARDLOG on critical system calls, our evaluation shows that it comes close, achieving a window of just a few system calls even on very intensive workloads. The specifics of our design are as follows.

We classify system calls into several categories, and associate a *weight* with each category. When the added

weights of the events in the per-CPU message cache reaches a specified threshold W_{th}, the buffer is immediately written to the ring buffer, and the consumer *e*Log is awakened immediately. In other words, we ignore p and w when the weight criteria is met. Our system call categorization is as follows, listed in decreasing weight order:

- *Privilege escalation and tampering:* This group consists of system calls typically involved in privilege escalation, tampering with other processes in the system, initiating malware execution, etc. Key examples in this category include `execve`, `setuid`, `kill`, `ptrace`, and variants.
- *Process provenance:* This group consists of a small number of system calls that affect process provenance and code loading, including `fork`, `clone`, `exit` and `mmap`.
- *File name and attribute change:* This group includes system calls that alter persistent attributes of files such as their names, permissions, etc. It includes system calls such as `rename`, `link`, `unlink`, and `chmod`.
- *Data endpoint creation:* At the next level of importance are system calls such as `open`, `connect` and `accept` that create new endpoints for reading or writing data.
- *Datagram network operations:* This group includes system calls such as `sendto` and `recvfrom`.
- *File descriptor operations:* System calls in this group include `dup`, `dup2`, `pipe`, `sockpair`, `fcntl`, etc.
- *Reads and writes:* This group includes syscalls such as `read`, `write`, `send`, `readv`, `pwrite`, and `recvmsg`.
- *Others:* This includes less important calls such as `close`.

Weight assignments for each category, as well as the global weight threshold W_{th}, are all configurable. Nevertheless, it is helpful to describe the defaults, as they help one understand how the priorities work. In the default setting, W_{th} equals the weight assigned to the privilege escalation and tampering category. This means that every critical event will be immediately propagated to the user level and logged, thus minimizing the tampering window. Weights for the next two categories are set at one-eighth of W_{th}, meaning that up to 8 of them can accumulate in the per-CPU buffer before they are logged. For each successive category, the weight is halved. This effectively means that the parameters p and w discussed earlier will primarily control the latency involving events in the lowest categories, such as reads and writes.

Finally, we incorporate a maximum time for which any event can be buffered in the per-CPU cache. Past this time limit, the cache contents are pushed to the ring buffer even if the cache contains just a single event. By default, this time limit is set to be 2^{24} nanoseconds $\approx$ 16 milliseconds.

4. Experimental Evaluation

*e*Audit is compatible with recent versions of Linux on 64-bit x86 processors. It has been tested with several recent versions of Ubuntu (20.04, 21.04 and 22.04) and Fedora. It uses the bcc toolchain [12]. We find that installing bcc from source [3] is preferable because it works across different Linux distributions and kernel versions. All of *e*Audit's dependencies are satisfied once bcc is installed this way. The goal of our experiments is to evaluate *e*Audit for its:

- ability to capture data without dropping events,
- performance improvements achieved by our design,
- runtime overhead,
- log tampering window, and
- data volume.

On many of these criteria, we compare our results with that of *sysdig*, the most performant among existing systems for full provenance collection. In Sec. 2, sysdig was configured to use a printable log file format for consistency with other logging tools. However, for the comparison results in this section, we configured it to use its binary ("capture file") format since it is faster and more compact.

We use the benchmarks described in Sec. 3.1. Unless otherwise stated, all results in this section were obtained on an i7-12700 system with 16GB of memory and a 500GB solid-state drive with Ubuntu 22.04 and Linux kernel 5.19 or 6.2. This processor has 8 performance cores and 4 efficiency cores for a total of 12 cores, and was configured to disable hyperthreading. All benchmarks use a configurable number of N cores, where N ranges between 1 and 12. For single-threaded benchmarks such as *postmark*, this meant running N copies of the benchmark, each with its own copy of data files and working directories. We used the default values of $p = 100$ and $w = 8$ mentioned earlier. Ring buffer was set to 16 MB, and all the syscalls from Table 1 were recorded.

4.1. Data Loss Comparison

We ran all of the benchmarks, while varying the number of cores used by the benchmark from 1 to 12. *None of the benchmarks resulted in any data loss in the case of eAudit*. Even `rdwr`, a benchmark designed to maximize the rate of provenance-related system calls, does not lead to data loss. (It generated $\sim$17M syscalls per second across 12 cores.)

Sysdig data loss. As shown in Fig. 15, sysdig is able to keep up with the kernel benchmark for the most part because the benchmark is CPU-intensive, and makes relatively few system calls. The remaining benchmarks are more I/O-intensive, and sysdig is unable to cope with them. Especially on multi-core workloads, it ends up dropping most events, with the worst case corresponding to over 95% data loss.

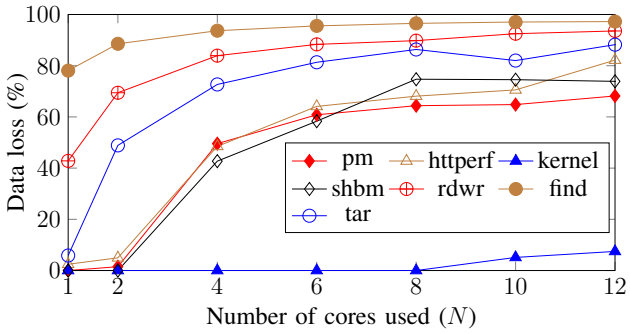

Fig. 15: Data loss experienced by Sysdig. *e*Audit does *not* lose any data on any of these benchmarks.

Cilium Tetragon and Tracee. Tetragon [18] and Tracee [83], [84] are two eBPF-based tools that can log system calls. It is unclear if the designers of these tools envisioned use cases involving whole system provenance logging. Nevertheless, we have include them here for completeness.

Tetragon provides a language for users to specify system calls to be logged. The specification is somewhat low level, requiring the identification of specific Linux trace points and low level type information for the arguments of interest. Support is provided for logging the most common argument types such as integers and file names, but we did not find a way to log more complex arguments such as the arrays of strings passed into `execve`. We used this specification language to develop a logger for most system calls used in our benchmarks. In addition, Tetragon comes with an example specification that logs raw system calls, recording the system call number and arguments as they appear in processor registers.

Tracee provides a full-featured system call logger that is simpler to use, as it only requires a specification of system calls that need to be logged. Similar to sysdig and other tools considered so far, Tracee does not require system call argument specification.

As shown in Fig. 16, both Tetragon and Tracee experience much more data loss than `sysdig`. For this reason, we evaluated the four lower intensity benchmarks used on `sysdig`. Whereas sysdig experiences almost no loss on the kernel benchmark, these two systems experience significant loss. On `postmark`, these systems drop about 90% of the records, in comparison with sysdig's 60%.

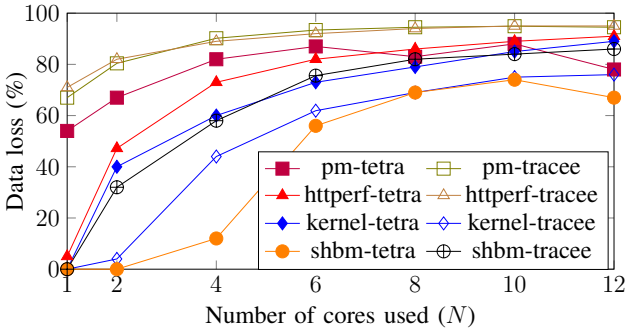

Fig. 16: Data loss experienced by Tracee and Tetragon.

For the postmark benchmark, we performed a second data loss measurement to confirm the results shown in Fig. 16. For this measurement, we used the raw system call logger that comes with the Tetragon system. Although this logger will log all system calls as opposed to the subset related to provenance, this is not a factor in the case of postmark since 98% of the system calls it makes are provenance related. This second measurement resulted in essentially the same curve as the one shown in Fig. 16.

4.2. Effectiveness of 2-level buffer and parameter tuning

Our optimized design achieves major reductions in data loss and runtime overhead. The data loss chart in Fig. 17 shows that *unoptimized* *e*Audit will experience data loss

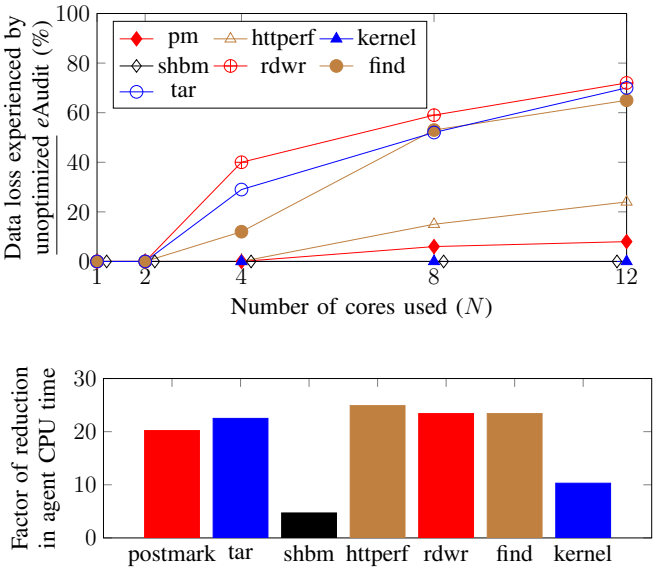

Fig. 17: Performance gains of 2-level buffer design and parameter tuning.

exceeding 50% on many multicore workloads. This data loss is completely eliminated in our optimized design.

The bottom half of Fig. 17 evaluates the overhead reduction achieved by our optimized design. It plots the ratio of agent CPU times between the unoptimized ($p = 1, w = 1$) and optimized ($p = 100, w = 8$, prioritized buffering) designs. When data is being lost, agent CPU time underestimates the overhead, so this chart only considers data points where there is no data loss. *Across these benchmarks, our optimized design decreases the agent overhead by* 18.4×.

4.3. Agent Overhead

Agent overhead, defined as the ratio of the agent and the benchmark CPU times, is the primary performance measure we have used so far. Fig. 18 plots this overhead for different benchmarks as a function of the number of cores used. The average overhead across all these data points is 3.1%.

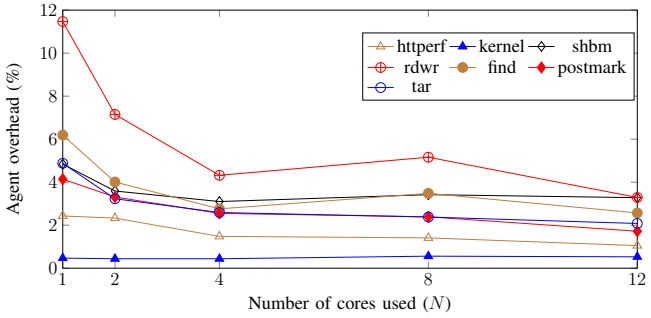

Fig. 18: *e*Audit Agent Overhead, defined as the ratio of the CPU time of the agent to the base CPU time of the benchmark.

Fig. 18 shows that agent overheads decrease slightly as the number of cores is increased. This does not mean that the agent takes less time for processing a multicore workload as compared to a single core. Instead, the reduction occurs

because the scalability of the workloads is generally worse than that of the agent. To show this effect, we have plotted the average per-system call overhead in Fig. 19. The median among these averages is 48 nanoseconds.

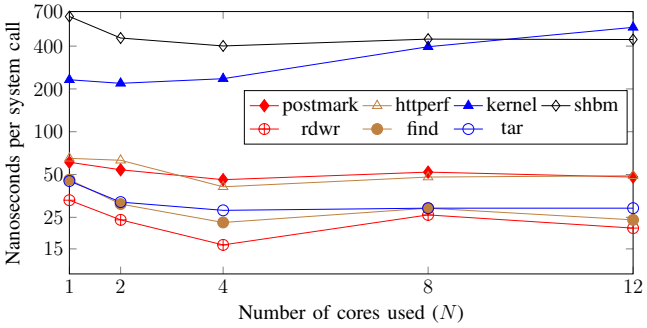

Fig. 19: *e*Audit's per-system-call Agent Overhead.

The flatness of these curves shows that the agent doesn't face much contention and scales well to multicore loads. Since `shbm` and `kernel` use many syscalls with large argument values, they have a higher per-system call overhead. Moreover, the CPU-intensive kernel build scales exceptionally well, surpassing the scalability of *e*Audit. This leads to an upward trend in the kernel benchmark overhead. (Since *e*Audit's overhead on CPU-intensive loads is very small, this increase isn't a source of concern.)

4.3.1. Comparison With Existing Systems

Since existing systems lose data on most benchmarks, we limit our comparison to the same two benchmarks `postmark` and `shbm` used in Sec. 2. To simplify our measurement and make it easier to reproduce our results, we did not slow down the benchmarks as in Sec. 2.2 to avoid data loss. As a result, Fig. 20 underestimates the overhead for systems such as CamFlow that experience significant data loss.

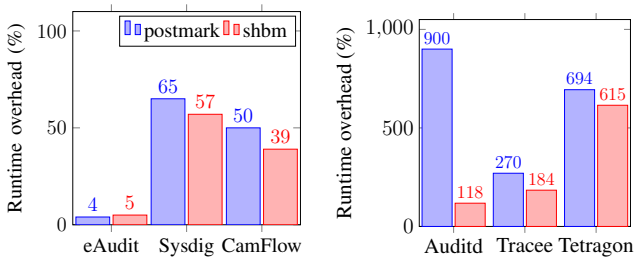

Fig. 20: Runtime Overhead of *e*Audit Vs Existing Systems.

Note that *e*Audit incurs between 4% and 5% overhead on these benchmarks, while all other systems have overheads that are much higher. Tetragon's particularly high overhead suggests that its authors may not have intended it to be used for continuous system call logging. Perhaps because of this, Tetragon tends to produce large syscall records that include much repeating information, e.g., parent process info, command line and arguments, all of the userids, etc.

4.4. Benchmark Overhead

Benchmark overhead refers to the increase in the wall clock time of the benchmarks due to *e*Audit. Fig. 21 shows that this overhead stays under 30% for all but `rdwr`.

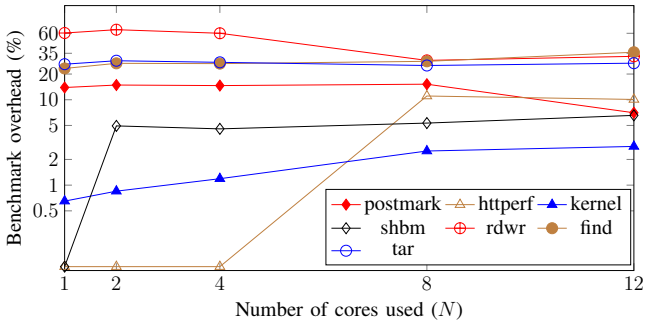

Fig. 21: *e*Audit's Benchmark Overhead: Percentage increase in benchmark wall clock time as a result of provenance data collection.

Similar to agent overhead curves, many wall clock overhead curves are also flat. Benchmarks that start off with very low overheads, e.g., `shbm` and `httperf`, show an upward trend. Other benchmarks such as `rdwr` and `postmark` show a downward trend because they have limited scalability. The average overhead across all the points in Fig. 21 is about 18%. If we leave out `find`, `tar` and `rdwr` that were purpose designed to max out the system call rate that can be sustained by the hardware and the OS, the overhead on the remaining benchmarks is just 5.8%.

4.5. Log Tampering Window

In this section, we analyze *e*Audit's tamper window and compare it with that of sysdig. For sysdig, we reused the measurement method from Sec. 2.3. In that method, there is a period of time between the benchmark termination and the measurement of log file length. Records written to the log file during this period are not included in the tamper window, thus leading to an under-approximation. The relative error introduced by this delay is insignificant when the window is large, as was the case for the tools discussed in Sec. 2, but we found that it significantly underestimates *e*Audit's log tamper window. So, we developed a second method based on directly instrumenting *e*Audit.

Specifically, instead of sending a kill signal to the benchmark, we send a `SIGUSR1` signal to *e*Log. On receiving this signal, *e*Log queries the current time t. As it continues to process events, *e*Log counts the number of events that have a timestamp $\leq t$. These are the events that took place before t but have not yet been written into disk at t, and hence represent the log tampering window. In addition to measuring the total number of records that are vulnerable to tampering, this method can count the number of *important* and *critical* events in the tamper window. Recall from Sec. 3.7 that *critical* ones include `execve` and other events related to privilege escalation and tampering. We define the important category to include the next two event groups, which include process provenance related events and those relating to file name and attribute changes.

As shown in Table 22, *e*Audit's tamper window is several hundreds, as opposed to 10K to 100K records for sysdig.

Implications for a successful log tampering attack. While a small reduction in tamper window size may not have

Benchmark	*e*Audit			Sysdig	Ratio
	Crit.	Imp.	All		
postmark	2	5	355	185904	*555*
httperf	2	1	355	26243	*74*
kernel	5	6	412	4473	*11*
shbm	2	5	355	3479	*10*
find	11	6	940	21144	*23*
rdwr	5	6	412	702277	*1704*
tar	2	5	355	56275	*168*
Geo. mean (N=1)	**3.6**	**5**	**426**	**35,000**	***82***
Geo. mean (N=12)	**2.5**	**3.5**	**1007**	**188,000**	***187***

Table 22: Log tampering windows for *e*Audit and Sysdig. Detailed data is shown for single-core workloads and then averages are included for both single-core (N=1) and multi-core (N=12) workloads.

a sufficient impact, *e*Audit achieves a 100-fold reduction, which significantly raises the bar for log tampering attacks. Moreover, due to our syscall prioritization, the window for critical and important events is in the single digits. Our study of privilege escalation attacks in the DARPA TC data [19] indicates that the combined number of important and critical syscalls — between 6–9 in Table 22 — is insufficient to carry out those attacks. This is not to say that such an attack is impossible, but it is certainly difficult.

Another way to assess the window is to measure it in terms of time. This is a challenge because we need to measure the time that a particular record was actually written into a file. While one can easily monitor file timestamps, it is difficult to accurately measure the time a particular record was written. But it is possible to obtain a lower bound on this time based on the rate at which these logging systems are processing the records from the memory buffers. Our measurements indicate that sysdig is processing in the range of 1M records per second, which means that it will take between 35ms and 188ms to clear the backlog. This is considerably longer than the 15ms suggested by the authors of HardLog [6]. The corresponding bound for *e*Audit will be just 1ms, based on its backlog of 426–1007 records.

4.6. Data Volume

Since whole system provenance data tends to be voluminous, it is important to minimize the storage requirements. In this regard, we compare *e*Audit with sysdig. We omitted `auditd` and CamFlow from our comparison — as noted in Sec. 2.4, their data volume is far larger than that of sysdig.

Both *e*Audit and sysdig support a binary format that is more compact than the printed version of provenance data, so we use this format for measuring uncompressed data size. We limited our comparison to a subset of benchmarks on which sysdig does not experience data loss. The results are shown in Table 23. Data volumes generated by sysdig are about $11\times$ more than that of *e*Audit on the average.

Benchmark	Syscalls	*e*Audit	Sysdig	Ratio
postmark	32M	430MB	5.72GB	*13.3*
httperf	0.8M	25.3MB	290MB	*11.5*
kernel	27M	605MB	5.03GB	*8.3*
Geo. mean	**9M**	**187MB**	**2.03GB**	***10.8***

Table 23: Uncompressed Log Size Comparison.

Benchmark	*e*Audit	Sysdig	Ratio
postmark	141MB	653MB	*4.6*
httperf	9.6MB	116MB	*12*
kernel	178MB	619MB	*3.5*
Geo. mean	**62.3MB**	**361MB**	***5.78***

Table 24: Log sizes after gzip compression.

Table 24 shows log sizes after compression using *gzip*. Sysdig logs are more compressible, so the difference in log sizes shrinks. But a substantial gap remains, with *e*Audit logs having about one-sixth the size of sysdig logs.

4.7. Discussion

Applicability to APT Detection. We have used *e*Audit data to construct provenance graphs. We have taken simple scenarios, such as those observed during a run of our benchmarks, constructed the corresponding graph, run queries on it, and navigated it. Although we have not presented an experimental evaluation of its effectiveness in detecting attacks, this is because there is already a rich body of research that has shown that system call data, together with arguments, is sufficient for detecting these attacks [40], [38], [31], [70], [41], [91], [100], [98], [23], [101], [4], [36], [8], [60], [13], [79]. As discussed in Table 1, the list of syscalls and arguments we collect is a superset of that collected by TRACE [44], which has been used extensively by researchers for APT data collection and analysis. Both *e*Audit and TRACE provide all syscall arguments, except for the data buffer argument to `read`, `write`, etc. Other important information such as timestamps, process and thread identifiers, and sequence numbers are also included.

Limitations and Future Work. We have shown that *e*Audit scales to processors with a dozen cores, and can sustain peak workloads on such machines. It is possible that scaling to processors with a larger number of cores will require additional design measures, e.g., parallelizing the user level logging process, using multiple ring buffers, and/or increasing the size of message caches.

Some previous provenance collection systems (e.g., LPM [11], HiFi [80], CamFlow[78]) go beyond the system call interface and can collect some kernel events unrelated to syscalls. The practical significance of this ability for APT detection has not been established. To the extent it is useful, we note that there is scope for extending *e*Audit in this direction because eBPF can hook into many interfaces in the Linux kernel.

5. Related Work

System-call data has underpinned most research on intrusion detection, prevention and recovery in the past 25+ years [28], [93], [86], [25], [30], [51], [33], [88], [89], [52], [57]. Prompted in part by this, *coarse-grained provenance* collection approaches have often been based on audit logs, e.g., Spade [32] and Trace [44]. Others rely on directly instrumenting the OS kernel, e.g., LPM [11], HiFi [80], CamFlow [78], and KCAL [62]. An advantage of OS instrumentation is that coverage is extended to include some kernel events that

don't relate to system calls, but the practical significance of these events for APT analysis is unknown. On the downside, approaches involving kernel changes are hard to maintain: just one of these systems (CamFlow) is available for OS kernels released in the past 10 years [94].

PROVBPF [59] is an implementation of CamFlow-style provenance capture for container environments using the Linux eBPF framework. However, their approach is based on *extension* of eBPF called SABPF. As such, they still need kernel modifications, leading to the same maintenance/deployability concerns.

Provenance collection. W3C [92] defines provenance as "information about entities, activities, and people involved in producing a piece of data or thing, which can be used to form assessments about its quality, reliability or trustworthiness." Early provenance research was motivated by concerns of scientific reproducibility, which requires recording the full details of every process execution on an OS, including the contents of all code and data files used. Provenance-aware storage [72] was proposed as a natural place to record such information. Reproducibility essentially requires historical file contents to be preserved, leading to high storage costs.

Security applications such as attack detection and forensics don't require full reproducibility, but they benefit from recording a broader set of security-relevant events, a factor that led to many recent provenance efforts to be based on system calls, e.g., Spade [32], Trace [44] and CamFlow[78].

OS Auditing and Logging Tools. Linux auditing system `auditd` was originally developed by Red Hat for Common Criteria [27] certification. It supports a range of security events, including system calls. This feature made `auditd` the go-to source for numerous provenance collection systems [32], [44], [91], [54], [63], [55], [99] as well as APT analysis [40], [38], [31], [70], [41], [91], [100], [98], [23], [101], [79]. However, the authors of `auditd` likely didn't intend their system for continuous system call logging. As a result, it has faced performance challenges when used for coarse-grained provenance collection. The go-audit project [87] is aimed at faster user-level processing than `auditd`, but since it continues to rely on `kauditd` and netlink sockets that account for 50% of the Linux auditing system's overhead (Fig. 5 in [62]), it cannot achieve our performance goals.

Tracee [83], [84] is an "eBPF-based threat detection engine," specializing in rules for detecting suspicious and/or evasive behaviors. There is also a stand-alone tool for event collection (Tracee-eBPF) that was already discussed in our evaluation. Sysdig is a "simple tool for deep system visibility" that captures system calls and other OS events, and provides the features of "strace + tcpdump + htop + iftop + lsof ..." For system call monitoring, Sysdig improves significantly over the performance of `auditd`, prompting recent research efforts [98], [23] to switch to Sysdig. Sysdig's default distribution uses a kernel module, but an eBPF version is also available [14]. As acknowledged by the authors of the eBPF version [14], we found its performance to be worse than sysdig's native version, and hence our experiments use the (more performant) native version.

Sysmon for Linux [26] is an eBPF-based implementation of the main functionality of Windows Sysmon [66]. It focuses on a small subset of system calls, including process creation/termination and network connections. The events tracked are insufficient for tracking *data provenance,* e.g., determining the dependencies of files on the system. Moreover, previous work shows that data provenance related calls (e.g., read, write, open and close) account for an overwhelming majority of system calls [40], [42], which means that data provenance tracking is inherently more demanding. So it is not very meaningful to compare Sysmon's performance with that of the other tools discussed in this paper.

Datadog [5] provides a log management framework and rule-based threat detection using eBPF, but does not have a general capability for syscall logging. In earlier work [1], we studied the feasibility of building an audit collection based on eBPF, but the performance concerns central to this paper were not examined there.

Fine-grained provenance. Coarse-grained provenance can lead to a *dependence explosion* that can degrade attack forensics. Some recent works have developed techniques to mitigate dependence explosion at the time of analysis [70], [41], [37]. A more common alternative is to rely on fine-grained information flow (aka taint) tracking [74], [96], [10], [49], [53], [45], [46], but unfortunately, it slows down systems by 2x to 10x, while greatly increasing log sizes. To address performance challenges, BEEP [55], PROTRACER [64] and MPI [63] developed a new fine-grained tracking technique called *execution-partitioning.* MCI [54] and PROPATROL [69] perform fine-grained tracking using model-based inference. ALchemist [99] combines application logs with system audit logs to derive finer-granularity provenance. Note that all of these techniques still require system call audit data, and hence they can directly benefit from the performance and scalability gains of our approach.

Log Tampering. Paccagnella et al [77] show that `auditd` is vulnerable to tampering attacks on in-memory records. We show that other data collection systems are vulnerable as well, and that their windows can be very large. Hoang et al [39] present an algorithm that improves in terms of performance and security over Paccagnella et al. Neither of these works prevent the compromise of audit records but ensure that the tampering effort will be detected. In contrast, our work is aimed at minimizing the tamper window and hence maximizing attack evidence that is preserved. In this aspect, the goals of HARDLOG [6] are similar to ours. They ensure that critical syscalls (e.g., execve) are logged synchronously, while others are logged with a bounded delay. However, in order to achieve this, they require specialized hardware, and moreover, make significant kernel changes that allow them to avoid sending the log data to the user level. In contrast, our work prioritizes deployability on today's hardware and out-of-box compatibility with existing Linux distributions. Despite these restrictions, we show that we can achieve very small log tampering windows, especially for critical and important system calls.

Data reduction techniques. Many researchers have focused on reducing the massive size of audit logs. One line of investigation is (lossless) compression [16], [95], [17], [24], [20]. Since compressed data is not amenable to general-purpose search or analysis algorithms, these techniques are primarily useful for reducing storage costs rather than analysis costs. In contrast, (lossy) data reduction that prunes away "unimportant" events can reduce analysis costs as well.

One class of data reduction techniques deem benign events unimportant, and store only the events likely to be part of attacks. Winnower [91] uses a DFA learning technique to prune away benign events. They report impressive reductions if the same application is replicated on numerous hosts. However, the technique does not seem very effective when applied to individual processes [43]. Rapsheet [37] suggests pruning away benign events unless they have a causal relationship to a suspicious event.

A key drawback of these approaches is that malicious events may be misclassified as benign, causing attack steps or their effects to be missed during a forensic analysis. LogApprox [68] avoids such misses, but allows generalizations that can introduce spurious dependencies. In contrast, a number of research efforts avoid both false positives *and* false negatives in the dependency relationships. LogGC [56], [61] is a garbage-collection-inspired approach to discard operations on temporary files used exclusively by a single process. Clearly, removal of these operations cannot sever any causal chains. While this technique is effective when used together with their fine-grained unit instrumentation [55], it has only a modest effect in coarse-grained settings [42], [43]. NodeMerge [90] identifies templates for repeated activities such as library loading and replaces them with a single node. On some workloads, it can achieve significant reductions, but on others, its effect can be modest [43].

While LogGC and NodeMerge identify two important instances of event redundancy, Xu et al [97] develop a more general characterization called *full-trackability equivalence* that achieves $\sim 2\times$ reduction in data size [42], [43]. By provably preserving reachability relationships between nodes at all times, their technique ensures accuracy of forensic traceback or trace-forward results. In subsequent work [42], we showed that for faithful forensic analysis results, it is sufficient to preserve forward reachability from a node at times when that node's state can possibly change. This relaxation, combined with global optimizations enabled by our versioned graph formulation of the problem, enabled our *full dependence preservation (FD)* technique to achieve a further $\sim 4\times$ data reduction [42], [43] over Xu et al. Zhu et al [102] explore dependency preserving as well as the more aggressive benign event pruning techniques, but their main focus is on simpler algorithms that yield faster runtimes.

The above efforts view data reduction as a post-processing phase operating on initial data generated by the tools studied in this paper. In contrast, *e*Audit's goal is to reduce the size of this initial data. A reduction here has the potential to translate into corresponding reduction in the output size of data reduction algorithms. We have already shown this for data compression techniques (Table 24). Moreover, these data size reductions can lead to a proportionate improvement in the runtimes performance of the entire data reduction pipeline.

6. Conclusion

The research presented in this paper identifies and analyzes critical bottlenecks in existing audit collection systems, including high performance overheads, the dropping of a large fraction of events under sustained workloads, and large windows for log tampering. We presented several new techniques to overcome these challenges, including a compact data encoding technique that significantly cuts down data volumes; a two-level buffering scheme that minimizes contention and avoids data loss even on intense multi-core workloads; an analytical model for optimally tuning latency and throughput; and an event prioritization scheme that reduces opportunities for log tampering. Through targeted experiments, we showed that our techniques achieve their objectives. The techniques developed in the paper can be directly applied to improve the performance of other eBPF-based systems that gather nontrivial amounts of data. They also have applicability to kernel extensions that involve gathering and sending significant amounts of data to the user level.

Acknowledgment

We wish to acknowledge the contribution of Richard Yao, a former graduate student of our department. Richard was examining the performance of Linux audit daemon and persuaded us to investigate an eBPF based approach for an efficient and deployable system call logger.

References

[1] Efficient audit data collection for linux. http://www.seclab.cs.sunysb.edu/seclab/pubs/rohitth.pdf.

[2] High-level tracing language for linux eBPF. https://github.com/iovisor/bpftrace. Accessed: 2022-07-21.

[3] Installing BCC. https://github.com/iovisor/bcc/blob/master/INSTALL.md. Accessed: 2022-02-28.

[4] A new tag-based approach for real-time detection of advanced cyber attacks. http://seclab.cs.sunysb.edu/seclab/pubs/nahidth.pdf.

[5] Runtime security monitoring with eBPF. https://www.sstic.org/media/SSTIC2021/SSTIC-actes/runtime_security_with_ebpf/SSTIC2021-Article-runtime_security_with_ebpf-fournier_afchain_baubeau.pdf.

[6] Adil Ahmad, Sangho Lee, and Marcus Peinado. Hardlog: Practical tamper-proof system auditing using a novel audit device. In *IEEE S&P*, May 2022.

[7] Chloe Albanesius. Target ignored data breach warning signs. http://www.pcmag.com/article2/0,2817,2454977,00.asp, 2014.

[8] Abdulellah Alsaheel, Yuhong Nan, Shiqing Ma, Le Yu, Gregory Walkup, Z Berkay Celik, Xiangyu Zhang, and Dongyan Xu. ATLAS: A sequence-based learning approach for attack investigation. In *USENIX Security*, 2021.

[9] APT notes. https://github.com/kbandla/APTnotes. Accessed: 2023-08-06.

[10] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *PLDI*, 2014.

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

[12] bcc reference guide. https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md. Accessed: 2022-02-28.

[13] Bibek Bhattarai and Howie Huang. Steinerlog: prize collecting the audit logs for threat hunting on enterprise network. In *ACM CCS*, 2022.

[14] Gianluca Borello. Sysdig and Falco now powered by eBPF. https://sysdig.com/blog/sysdig-and-falco-now-powered-by-ebpf/, 2019. Accessed: September 2023.

[15] BPF documentation. https://www.kernel.org/doc/html/latest/bpf/index.html. Accessed: 2022-07-15.

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

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

[18] Cilium. Cilium/tetragon: Ebpf-based security observability and runtime enforcement. https://github.com/cilium/tetragon.

[19] DARPA transparent computing Engagement 3 data release (also includes Engagement 5 data). https://github.com/darpa-i2o/Transparent-Computing/. Accessed: 2023-6-8.

[20] Hailun Ding, Shenao Yan, Juan Zhai, and Shiqing Ma. Elise: A storage efficient logging system powered by redundancy reduction and representation learning. In *USENIX Security*, 2021.

[21] eBPF — introduction, tutorials & community resources. https://ebpf.io. Accessed: 2022-07-13.

[22] Actions taken by equifax and federal agencies in response to the 2017 breach. https://www.gao.gov/assets/700/694158.pdf.

[23] Pengcheng Fang, Peng Gao, Changlin Liu, Erman Ayday, Kangkook Jee, Ting Wang, Yanfang Fanny Ye, Zhuotao Liu, and Xusheng Xiao. Back-propagating system dependency impact for attack investigation. In *USENIX Security*, 2022.

[24] Peng Fei, Zhou Li, Zhiying Wang, Xiao Yu, Ding Li, and Kangkook Jee. SEAL: Storage-efficient causality analysis on enterprise logs with query-friendly compression. In *USENIX Security*, 2021.

[25] Henry Hanping Feng, Oleg M Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *IEEE S&P*, 2003.

[26] Sysmon for Linux. Sysmon for Linux. https://github.com/Sysinternals/SysmonForLinux. Accessed: 2023-08-09.

[27] International Organization for Standardization. ISO/IEC 15408-2:2008. information technology —- security techniques — evaluation criteria for it security —- part 2: Security functional components, 2009.

[28] Stephanie Forrest, Steven Hofmeyr, Anil Somayaji, and Thomas Longstaff. A sense of self for unix processes. In *IEEE S&P*, 1996.

[29] FridayOrtiz https://ortiz.sh/contact/. Every boring problem found in eBPF. https://tmpout.sh/2/4.html. Accessed: 2022-05-21.

[30] Debin Gao, Michael K Reiter, and Dawn Song. Gray-box extraction of execution graphs for anomaly detection. In *ACM CCS*, 2004.

[31] Peng Gao, Xusheng Xiao, Ding Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, Chung Hwan Kim, Sanjeev R Kulkarni, and Prateek Mittal. SAQL: A stream-based query system for real-time abnormal system behavior detection. In *USENIX Security*, 2018.

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

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

[34] Brendan Gregg. Linux extended BPF (eBPF) tracing tools. https://www.brendangregg.com/ebpf.html. Accessed: 2022-07-19.

[35] Steve Grubb. Linux audit. https://people.redhat.com/sgrubb/audit/. Accessed: 2022-07-19.

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

[37] Wajih Ul Hassan, Adam Bates, and Daniel Marino. Tactical provenance analysis for endpoint detection and response systems. In *IEEE S&P*, 2020.

[38] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. NODOZE: Combating threat alert fatigue with automated provenance triage. In *NDSS*, 2019.

[39] Viet Tung Hoang, Cong Wu, and Xin Yuan. Faster yet safer: Logging system via fixed-key blockcipher. In *USENIX Security*, 2022.

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

[41] Md Nahid Hossain, Sanaz Sheikhi, and R. Sekar. Combating dependence explosion in forensic analysis using alternative tag propagation semantics. In *IEEE S&P*, 2020.

[42] Md Nahid Hossain, Junao Wang, R Sekar, and Scott D Stoller. Dependence preserving data compaction for scalable forensic analysis. In *USENIX Security*, 2018.

[43] Muhammad Adil Inam, Yinfang Chen, Akul Goyal, Jason Liu, Jaron Mink, Noor Michael, Sneha Gaur, Adam Bates, and Wajih Ul Hassan. Sok: History is a vast early warning system: Auditing the provenance of system intrusions. In *IEEE S&P*. IEEE, 2023.

[44] Hassaan Irshad, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Kyu Hyung Lee, Jignesh Patel, Somesh Jha, Yonghwi Kwon, Dongyan Xu, and Xiangyu Zhang. TRACE: Enterprise-wide provenance tracking for real-time APT detection. *IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY*, 2021.

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

[46] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking. In *USENIX Security*, 2018.

[47] Jeffrey Katcher. Postmark: A new file system benchmark. Technical report, Network Appliance, 1997.

[48] Matt Keenan. Taming tracepoints in the Linux Kernel. https://blogs.oracle.com/linux/post/taming-tracepoints-in-the-linux-kernel. Accessed: 2022-07-21.

[49] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. *ACM CCS*, 2012.

[50] Aaron Kili. Sysdig – a powerful system monitoring and troubleshooting tool for linux. https://www.tecmint.com/sysdig-system-monitoring-and-troubleshooting-tool-for-linux/.

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

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

[53] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. Ldx: Causality inference by lightweight dual execution. *ASPLOS*, 2016.

[54] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela Ciocarlie, Ashish Gehani, and Vinod Yegneswaran. MCI: Modeling-based causality inference in audit logging for attack investigation. In *NDSS*, 2018.

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

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

[57] Zhenkai Liang, Weiqing Sun, V. N. Venkatakrishnan, and R. Sekar. Alcatraz: An isolated environment for experimenting with untrusted software. In *ACM TISSEC*, 2009.

[58] libbpf. https://www.kernel.org/doc/html/latest/bpf/libbpf/index.html. Accessed: 2022-07-11.

[59] Soo Yee Lim, Bogdan Stelea, Xueyuan Han, and Thomas Pasquier. Secure namespaced kernel audit for containers. In *SoCC*, 2021.

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

[61] Shiqing Ma, Kyu Hyung Lee, Chung Hwan Kim, Junghwan Rhee, Xiangyu Zhang, and Dongyan Xu. Accurate, low cost and instrumentation-free security audit logging for Windows. In *ACSAC*, 2015.

[62] Shiqing Ma, Juan Zhai, Yonghwi Kown, Kyu Hyung Lee, Xiangyu Zhang, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. Kernel-supported cost-effective audit logging for causality tracking. In *USENIX ATC*, 2018.

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

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

[65] MANDIANT: Exposing one of China's cyber espionage units. https://www.fireeye.com/content/dam/fireeye-www/services/pdfs/mandiant-apt1-report.pdf. Accessed: 2023-08-06.

[66] Markruss. Troubleshooting with the windows sysinternals tools - sysinternals. https://learn.microsoft.com/en-us/sysinternals/resources/troubleshooting-book. Accessed: 2023-08-09.

[67] Metasploit. Metasploit — penetration testing software, pen testing security. https://www.metasploit.com/.

[68] Noor Michael, Jaron Mink, Jason Liu, Sneha Gaur, Wajih Ul Hassan, and Adam Bates. On the forensic validity of approximated audit logs. In *ACSAC*, 2020.

[69] Sadegh M Milajerdi, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. Propatrol: Attack investigation via extracted high-level tasks. In *Intl. Conf. on Information Systems Security*, 2018.

[70] Sadegh M. Milajerdi, Rigel Gjomemo, Birhanu Eshete, R. Sekar, and V.N. Venkatakrishnan. HOLMES: Real-time APT detection through correlation of suspicious information flows. In *IEEE S&P*, 2019.

[71] Stephanie Mlot. Neiman Marcus hackers set off nearly 60k alarms. http://www.pcmag.com/article2/0,2817,2453873,00.asp, 2014. [Online; accessed 06-August-2023].

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

[73] Andrii Nakryiko. BPF ring buffer. https://nakryiko.com/posts/bpf-ringbuf/. Accessed: 2022-07-18.

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

[75] GAO (U.S. Government Accountability Office). Solarwinds cyberattack demands significant federal and private-sector response. https://www.gao.gov/blog/solarwinds-cyberattack-demands-significant-federal-and-private-sector-response-infographic. Accessed: 2022-09-19.

[76] Committee on Oversight and Government Reform. The OPM data breach: How the government jeopardized our national se curity for more than a generation. https://oversight.house.gov/report/opm-data-breach-government-jeopardized-national-security-generation/.

[77] Riccardo Paccagnella, Kevin Liao, Dave Tian, and Adam Bates. Logging to the danger zone: Race condition attacks and defenses on system audit frameworks. In *ACM CCS*, 2020.

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

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

[80] Devin J Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. Hi-Fi: Collecting high-fidelity whole-system provenance. In *ACSAC*, 2012.

[81] Cilium Project. BPF and XDP reference guide. https://docs.cilium.io/en/latest/bpf/. Accessed: 2022-06-20.

[82] Jonathan Salwan. https://shell-storm.org/shellcode/index.html.

[83] Aqua Security. Aqua Tracee: Runtime eBPF threat detection engine. https://www.aquasec.com/products/tracee/. Accessed: 2022-12-02.

[84] Aqua Security. Tracee: Runtime security and forensics using eBPF. https://github.com/aquasecurity/tracee. Accessed: 2022-12-02.

[85] Offensive Security. Exploit database - exploits for penetration testers, researchers, and ethical hackers. https://www.exploit-db.com/.

[86] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. A fast automaton-based approach for detecting anomalous program behaviors. In *IEEE S&P*, 2001.

[87] Slackhq. Slackhq/go-audit: Go-audit is an alternative to the AUDITD daemon that ships with many distros. https://github.com/slackhq/go-audit.

[88] Weiqing Sun, R. Sekar, Gaurav Poothia, and Tejas Karandikar. Practical proactive integrity preservation: A basis for malware defense. In *IEEE S&P*, 2008.

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

[90] Yutao Tang, Ding Li, Zhichun Li, Mu Zhang, Kangkook Jee, Xusheng Xiao, Zhenyu Wu, Junghwan Rhee, Fengyuan Xu, and Qun Li. Nodemerge: Template based efficient data reduction for big-data causality analysis. In *ACM CCS*, 2018.

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

[92] W3C. PROV-overview: An overview of the PROV family of documents. https://www.w3.org/TR/prov-overview/Overview.html. Accessed: 2022-09-17.

[93] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE S&P*, 1999.

[94] Wikipedia. Provenance (computer science). https://en.wikipedia.org/wiki/Provenance#Computer_science, last accessed: 2022-09-17.

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

[96] Wei Xu, Sandeep Bhatkar, and R Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security*, 2006.

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

[98] Zhiqiang Xu, Pengcheng Fang, Changlin Liu, Xusheng Xia, Yu Wen, and Dan Meng. DEPCOMM: Graph summarization on system audit logs for attack investigation. In *IEEE S&P*, 2022.

[99] Le Yu, Shiqing Ma, Zhuo Zhang, Guanhong Tao, Xiangyu Zhang, Dongyan Xu, Vincent E Urias, Han Wei Lin, Gabriela F Ciocarlie, Vinod Yegneswaran, and Ashish Gehani. ALchemist: Fusing application and audit logs for precise attack provenance without instrumentation. In *NDSS*, 2021.

[100] Jun Zeng, Zheng Leong Chua, Yinfang Chen, Kaihang Ji, Zhenkai Liang, and Jian Mao. WATSON: Abstracting behaviors from audit logs via aggregation of contextual semantics. In *NDSS*, 2021.

[101] Jun Zeng, Xiang Wang, Jiahao Liu, Yinfang Chen, Zhenkai Liang, Tat-Seng Chua, and Zheng Leong Chua. Shadewatcher: Recommendation-guided cyber threat analysis using system audit records. In *IEEE S&P*, 2022.

[102] Tiantian Zhu, Jiayu Wang, Linqi Ruan, Chunlin Xiong, Jinkai Yu, Yaosheng Li, Yan Chen, Mingqi Lv, and Tieming Chen. General, efficient, and real-time data compaction strategy for APT forensic analysis. *IEEE TIFS*, 2021.

Appendix A.
Meta-Review

A.1. Summary

The study presents *e*Audit, an innovative audit data collection system that addresses limitations existing systems, including high overheads, data loss, log tampering vulnerability, and large data volumes. The extended Berkeley Packet Filter (eBPF) framework allows *e*Audit to offer a scalable, easy-to-deploy solution compatible with most Linux distributions.

A.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field

A.3. Reasons for Acceptance

1) The paper targets a long-known issue and has a clear presentation. The authors take a data-driven approach to uncover limitations with prior solutions, and introduces new techniques that lay a compelling foundation for future research in the domain of intrusion detection.
2) The paper provides a valuable step forward by introducing novel techniques that allow *e*Audit to simultaneously and efficiently deal with high overhead, log loss, log tampering, and large data volumes.
3) The paper creates a new tool *e*Audit (to be open-sourced upon publication) that is based on eBPF and can be easily deployed to most Linux systems. This capability also draws a bridge to the large body of recent research with eBPF (e.g., for other system observability tasks) that opens many questions for future work in intrusion detection.

A.4. Noteworthy Concerns

1) Sysmon for Linux and Cilium Tetragon are closely related systems aimed at efficient audit log collection. The paper provides a performance comparison against Cilium Tetragon, demonstrating improvements w.r.t. data loss. The authors could enhance the depth of the paper with a comprehensive discussion on architectural differences between *e*Audit and Tetragon/SysMon that make *e*Audit better. This comparison could illuminate, for instance, the utilization of ring buffers and per-CPU caches, which could have substantial implications for system performance.
2) Several reviewers have expressed concerns regarding the validity and generality of the system call prioritization technique in real-world scenarios. The paper could be improved with a more comprehensive discussion/analysis of system call prioritization in practice. Moreover, such an examination could enable a more nuanced understanding of this method and its broader implications for threat detection and forensic analysis.
3) Regarding the reduction of log tampering window, the paper could be improved with a more comprehensive analysis on how real attacks are prevented or made more difficult to carry out.

Appendix B.
Response to the Meta-Review

Regarding the concerns:

1) As noted in our related work, Sysmon does not track data provenance. Since data provenance tracking is inherently much more performance intensive, including Sysmon in our performance comparisons does not seem appropriate. With regard to the suggestion that we more deeply examine performance bottlenecks in related tools, this is indeed what we have devoted most of the paper to. Specifically:
   - We established and explained three main contributors of performance overheads: (a) verbosity/redundancy in the data recorded for each system call, (b) frequent access to the message queue used to transmit this data to the user-level agent, and (c) task-switching to wake up the agent.
   - We presented and comprehensively evaluated *e*Audit's design features that tame these overheads.
   - We also discussed (and in most cases, quantified) the performance bottlenecks in ebpf-based designs, including the use of perf buffers, direct access to the ring buffer on each syscall instead of using a message cache, and waking up the user-level agent on each message.

   The role of a scientific paper is to establish the prevalence of a problem, and to present and evaluate a solution to this problem. We do not think it is helpful to cross over into dissecting specific software tools from the internet or reviewing the features/bugs that led to their slowdown — especially when we are unable to access detailed design documents or papers that help us understand their goals, objectives and design rationale.
2) While we agree that system-call prioritization can be strengthened by evaluating against real-world attacks, there is only so much space in this paper. Between the number of systems we have compared with, and the many experiments designed to establish the effectiveness of our design choices, there is no more room left.
3) We do not dispute that more in-depth evaluation can further strengthen the paper's contribution but space constraints prevent us from getting into more depth. More importantly, while we would understand skepticism on small improvements to the tamper window, we believe that a $100\times$ reduction is persuasive. Note that this reduction occurs *even before prioritization* is applied. With prioritization, the number of critical/important syscalls in the tamper window is another $100\times$ smaller.