Sealing the Window: Efficient Tamper Protection for Provenance Logs*

Sagar Mishra and R. Sekar
Stony Brook University, NY, USA.
{*sagmishra,sekar*}*@cs.stonybrook.edu*

***Abstract*—Today's advanced cyber attacks routinely circumvent existing protection measures. Analysts must rely on after-the-fact detection, based on provenance logs, to understand and recover from these intrusions. Since attackers prize the ability to stay hidden, they take every measure to remove all signs of attacks from these logs. In this research, we begin with a study of previous work on protecting provenance logs from such tampering. Through a motivating experimental study, we show that audit logging systems deployed today are highly susceptible to tampering. Moreover, existing tamper detection measures either require specialized hardware and custom OS modifications, or they incur excessive performance costs.**

To overcome these challenges, we first analyze previous research to identify their key bottlenecks. We then present new techniques and algorithms that avoid these bottlenecks, while also providing several additional benefits. Our techniques have been implemented into a system *WinSeal* that achieves well over a 10× reduction in overhead as compared to previous tamper detection techniques. On the protection front as well, WinSeal improves a key metric, namely, tamper window duration, by an order of magnitude as compared to previous techniques compatible with stock hardware and software. Our software is being open-sourced along with this paper.

1. Introduction

Advanced Persistent Threats (APTs) are coordinated cyber-attacks that infiltrate target organizations over an extended time periods to achieve specific goals, such as stealing sensitive information, perpetrating financial fraud, deploying ransomware, or disrupting critical infrastructure. APTs prioritize persistence and stealth, often remaining hidden within compromised networks for weeks or even months.

Since APTs tend to bypass all deployed protection measures, organizations must rely on a post-attack analysis of provenance logs to understand how the attackers entered the system, and to determine the extent of damage they have inflicted. State-of-the-art techniques for APT detection and forensic analysis [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17] rely on system call logs, as they provide a *complete* record of all security-relevant activities on the system. This contrasts with application logs that only track specific applications, and that too is limited to a subset of actions deemed relevant by the application designer. Unfortunately, attackers exploit applications in unforeseen ways, leading to blind spots in application logs. Moreover, APT actors routinely gain shell access and/or the ability to run malware, at which point there are no application logs to capture their behavior.

APT actors' ability to carry out their mission is critically dependent on staying hidden for extended periods of time. Since an analysis of provenance logs can reveal their presence, these logs are often targeted by attackers for tampering.[1] Attackers can try to remove logs entirely, erase/alter specific records relating to their activity, or inject fabricated records designed to mislead and distract security analysts. Moreover, by erasing their activity, attackers can protect their "intellectual property" in the form of the tactics, exploits and toolkits used. Finally, alterations to the log can hide the full extent of damage and contamination, thereby affording continued footholds for attackers even past a forensic analysis and cleanup of the affected host(s). These factors motivate research into audit log integrity protection, the topic of this paper and many previous works, e.g., [18], [19], [20], [21], [22], [23].

Increasingly, log tampering is not just limited to highly sophisticated attackers. The ever-increasing availability of attack-related information on the internet, including automated tools and publicly available tutorials [10], [24], [25], [26], makes log tampering defense critical even when dealing with some of the garden-variety attacks.

1.1. Previous research in audit log integrity

Approaches for audit log protection fall into two main categories: *tamper prevention* and *tamper detection*. Tamper prevention is aimed at stopping attackers from removing or altering records that have been stored in the log. In contrast, tamper detection does not block log alterations, but guarantees that any and all changes to the log will be detected.

The simplest approach for tamper prevention is to use write-only storage, or to log to a remote server that isn't exposed to attackers. Even if storage is secured this way, Paccagnella et al [20] showed that log entries remain in the operating system buffers for a few hundred milliseconds before they reach any storage. This leads to a *log tamper window,* consisting of event records that can be erased/altered before they reach secure storage. On busy servers, this window can be large, reaching hundreds of thousands of system calls [27].

One way to eliminate the tamper window is to avoid

*This work was supported by NSF grants 1918667 and 2153056. This version includes minor revisions to the conference version in the form of an expanded related work section.

1. Log tampering requires attackers to gain root access, but most APT actors eventually achieve such access, as their missions typically depend on it.

buffering of log entries. This requires synchronous commiting of each system call record, which can be prohibitively expensive on commodity OSes. Consequently, researchers have proposed a combination of custom hardware and operating system changes in order to bring the overheads down. HardLog [21] uses a dedicated audit device that enforces an append-only logging policy. In addition, they modify Linux to (a) synchronously log the most critical events, and (b) to log the remaining system calls with a bounded latency of 15 milliseconds. Instead of a dedicated device, Omnilog [22] relies on hardware and software for trusted execution environments (TEEs) in order to isolate logging operations from an OS that may be compromised by an attacker. Unfortunately, both approaches require specialized hardware, making them difficult to deploy on commodity systems.

By targeting a simpler goal, tamper detection techniques avoid the need for any special-purpose hardware. They also dispense with the need for secure remote logging, which requires a high-bandwidth network connection with 100% availability. They achieve this using cryptographic techniques that are applied synchronously with system calls. In particular, KennyLoggings [20] attaches a Message Authentication Code (MAC) to every log entry in the kernel. Fresh keys are used for each log entry to ensure forward secrecy [18]. Quicklog2 [19] further improves on the security of this technique by replacing non-NIST-approved cryptographic algorithms with new and efficient cryptographic constructs based on AES [28]. These efficient cryptographic primitives also achieve improved performance over KennyLoggings.

The key benefit of these tamper detection approaches is that, without needing any special hardware, they can prevent undetected changes to the log. Their downside is that attackers can still remove records, thereby hiding their activities and preventing analysts from understanding the vulnerabilities that led to the intrusion. Consequently, the system remains vulnerable to recurrences of the same attack. Moreover, analysts lack directions for recovery: without a record of the resources accessed by the attacker, analysts must make the worst-case assumption that everything on the affected host is compromised, including the OS and all the data files must be discarded. Indeed, the primary benefit of tamper detection is its deterrence effect: attackers may prefer leaving trails in the audit logs since they may or may not be picked up, while log tampering can be deterministically detected.

1.2. Need for New Approach

Previous research has established the importance of audit log integrity and developed the foundations for addressing the problem. However, important challenges remain before these foundations can be translated into practical solutions that can be deployed on production systems. We elaborate on some of these problems below.

Performance, compatibility or integrity? Existing approaches require users to choose between performance, log integrity, and hardware/software compatibility. Audit logging techniques compatible with commodity OSes [29], [30], [27], [31] don't come with any significant integrity protections. On the other hand, techniques incorporating tamper defenses introduce major compatibility challenges, such as the need for special-purpose hardware [21], [22] and/or significant OS modifications [21], [20], [19] that aren't compatible with widely deployed OS kernel versions. Moreover, many of these techniques incur major performance overheads, slowing down workloads by $2\times$ to $10\times$ or more [30], [31], [29], [20], [19].

Tamper prevention, Tamper detection, or Both? Previous techniques require a choice between tamper prevention and detection. In particular, techniques for tamper detection [20], [19] do nothing to *prevent* the log from being tampered with. Although alterations are detected, the attacker can wipe out the log entries, and thus prevent security analysts from understanding the events that led up to the attack.

At the other end of the spectrum, tamper prevention techniques aim to reduce the length of the tamper window. For instance, HardLog [21] targets a tamper window of 15 milliseconds, while eAudit's [27] range is closer to 100 milliseconds. Unfortunately, events in this window are 100% vulnerable: attackers can erase or alter them at will, without any risk of detection. In contrast, we present low-cost tamper detection *and* tamper prevention techniques that can be deployed independently, or in combination to reinforce each other.

1.3. Approach Overview

In contrast with previous work, our approach does not require users to choose between the options discussed above. Instead, *WinSeal provides "all of the above:" effective, low-overhead tamper protection that is deployable "as is" on today's Linux distributions.* WinSeal's tamper prevention comes with a low overhead and a small tamper window, benefits that previously required customized hardware. For added protection, WinSeal can provide tamper detection for a modest additional overhead. This added protection is particularly important in environments where continuous network connectivity cannot be assured.

WinSeal is developed on top of our eAudit system [32], and thus inherits the latter's deployability benefits. By using safety-verified eBPF probes [33] in the OS kernel, eAudit avoids the need for custom kernel modifications that have stood in the way of widespread deployment of previous audit loggers [34], [35], [19], [20]. Although eAudit's tamper window was shown to be much smaller than previous works in [27], we show in this work that this claim holds only under certain conditions. In particular, while eAudit's tamper window is low under high workloads, it can become very large under low-load or idle conditions. In this paper, we investigate the reasons for this discrepancy, tease out the reasons for the large tamper window, and present new techniques to reduce it to a few milliseconds. Our techniques not only result in orders of magnitude reduction in the average tamper window of eAudit, but also reduce its variability.

WinSeal's *tamper prevention* is the combined result of a small tamper window and secure remote logging. In

particular, unlike eAudit which uses a local file for audit log persistence, WinSeal maintains an ssh connection to a remote server that is locked down and hence assumed to be beyond the attacker's reach. Log entries are sent over this ssh connection as soon as they are received by WinSeal's user-level agent from its in-kernel eBPF probe. This design presents two opportunities for attacks that we mitigate by developing an efficient *tamper detection* method. First, an attacker can still exploit the tamper window and delete all of the log entries in this window. This can allow attackers to erase activities leading up to their gaining root privilege, thereby preventing security analysts from inferring that the log is compromised from that point on. Since WinSeal's tamper window is under 10 milliseconds, this does require a fast-moving privilege escalation attack. In the context of high-value systems, protection against such attacks may be deemed necessary. More importantly, even transient, sub-second network delays and interruptions can significantly lengthen the tamper window, as log entries will build up on the victim during such periods. WinSeal's tamper detection features ensure that the attacker cannot hide their tampering in these scenarios. Thus, it can play a crucial role for high-value systems, as well as environments where a high bandwidth, low-latency network connectivity cannot be assumed 100% of the time. The threat of network interruptions can be particularly acute in the case of mobile devices such as laptops.

WinSeal's tamper detection and prevention mechanisms can operate independent of each other, and can each be deployed in isolation. Prevention mechanisms may be sufficient for non-security-critical servers in enterprise environments with robust local network connectivity. Devices with intermittent network connectivity may be primarily protected using WinSeal's tamper detection features, with periodic transfers of logs to secure servers. For the remaining systems, both detection and prevention features may be deployed together. The combination offers the following benefits. First, it maximizes evidence preservation: if the attacker acquires root privilege on a victim at time t, none of his/her activities prior to $t-t_w$ can be hidden or erased. (Here, t_w is the tamper window.) Although the attacker can erase activities past this point, the erasure will be reliably detected by WinSeal. Finally, we show how our detection techniques can piggyback over the prevention features to achieve increased security by frequently rotating master keys used for cryptographic protection.

1.4. Key Results and Contributions

We summarize our key results and contributions below.

Experimental study of previous techniques. We present an experimental study in §3 that analyzes the strengths and weaknesses of previous techniques. Our study makes several new findings that were missed in previous works. For instance, we show that KennyLoggings and Quicklog2 incur overheads that are several times more than that of the `auditd` system they build on. Our study also uncovers previously unknown weaknesses in eAudit's tamper defenses: while its tamper window is relatively small under high load, it can be as large as several seconds under light loads. Most other systems, e.g., `auditd` and sysdig [30], have the opposite behavior: large tamper windows under high load.

In order to assess the significance of tamper windows of previous techniques, we studied the available privilege escalation exploits for three recent Linux vulnerabilities. Our experiments show that all complete within 15 milliseconds, which is far below the tamper window of existing audit logging techniques deployable on commodity hardware, e.g., close to 100 milliseconds for eAudit.

New techniques for reducing the tamper window. To minimize the exposure of audit logs to tampering after an attacker gains root access, we introduce new techniques that significantly reduce the tamper window. Building on eAudit's high-performance architecture, we first identify a key weakness in its queuing design: under low-load conditions, per-CPU message caches are not flushed in a timely manner, resulting in disproportionately large tamper windows. To address this, we redesign the message cache to allow shared access and develop a family of flushing algorithms (A_1 to A_3) for the timely eviction of log entries, thus ensuring a small tamper window for WinSeal under all types of loads.

New techniques for efficient tamper detection. Like previous works, our approach also attaches a message authentication code (MAC) to each audit entry. However, previous methods incurred high overheads due to serialized MAC computation. WinSeal avoids this bottleneck by requiring serialization only for the generation of a unique sequence number, which is performed efficiently using eBPF's atomic `fetch-and-add` primitive. Once the sequence number is assigned, MAC computation proceeds independently and in parallel across CPU cores, enabling the system to scale effectively under multicore workloads. This enables WinSeal to achieve over an order of magnitude reduction in overheads as compared to previous work [20], [19]. Moreover, our design achieves fault tolerance to dropped records. (Occasional drops of log entries may be unavoidable under peak loads.) With previous techniques, the loss of a single record means that the integrity of subsequent records cannot be verified. But in our design, since signing keys are based on sequence numbers, the integrity of subsequent records can be verified despite drops of a few previous records.

Our second contribution in this context is that we achieve strong cryptographic primitives similar to those of Quicklog2 [19] despite the constraints of the eBPF environment. In particular, eBPF does not permit loops. No standard libraries can be used either. Cryptographic primitives aren't available, so everything needs to be broken down into simple arithmetic, logical and bit operations. To cope with these restrictions, our design divides MAC generation into two parts. The first part involves generating signing keys for each audit record. These signing keys are generated by our user-level agent using AES in the manner suggested by the authors of Quicklog2, and securely communicated to the eBPF probe. However, our in-kernel MAC computation cannot use typical cryptographic techniques

due to the above-mentioned eBPF restrictions. Instead, we compute a MAC with a universal hashing technique that uses the signing key. Because of the properties of universal hashing, this construction is unconditionally secure (under the assumption that the signing keys are random).

Finally, if tamper detection and prevention features are deployed simultaneously, then our design can provide additional security by rotating initial keys frequently. In particular, we show how to embed an initial key for the next batch of records in a log entry of the current batch. As we describe later, WinSeal's short tamper window protects this key from attackers wanting to tamper with the records protected by this key.

Experimental Evaluation. We evaluate the performance and effectiveness of WinSeal using benchmarks and compare it with previous systems. Our techniques achieve a tamper window between 0.7 and 5 milliseconds, with the peak remaining below 10 milliseconds. To interpret this number in context, note that HardLog is designed to provide a 15 millisecond tamper window for most syscalls, but requires dedicated hardware and kernel modifications to achieve this.

Tamper window reduction mentioned above is achieved while increasing runtime overheads by just 1.1% over eAudit. Tamper detection contributes a modest additional 5.5% overhead. Our benchmarks use all available cores on our experimental platform, and are system-call intensive in nature. Despite this, the total overhead (including the overhead of eAudit) is 9.5% with tamper-window reduction, and 15% when tamper detection is also enabled.

WinSeal's features are now fully integrated into eAudit, and can be found in the eAudit's repository at https://eprov.org and our lab website http://seclab.cs.stonybrook.edu/download. A snapshot of our source code used in the artifact evaluation is available at https://doi.org/10.5281/zenodo.17283436.

1.5. Paper Organization

We begin with some background and our threat model in §2. Next, we detail our motivating study of existing audit logging systems in §3, followed by our approaches for tamper window reduction in §4 and tamper detection in §5. Experimental evaluation is described in §6, followed by related work (§7) and conclusions (§8).

2. Background

2.1. Threat model

We consider three classes of adversaries on a protected host computer system: (1) a remote attacker with no privileges on the target system, who attempts to gain access via network-based exploits or stolen credentials; (2) a local attacker who can already access the system as an ordinary user that lacks administrator privileges; and (3) an attacker that has gained root-level access. On contemporary OSes, there is no feasible approach for protecting any resources from the third category of attackers. For this reason, *our only goal for attackers who have gained root privilege is that they cannot delete, insert or modify audit records logged before root access was gained.*

We also assume that the host is free of any root-level attackers at the point when audit logging begins, and for a few seconds afterwards, so as to give time for the audit log initialization to complete. Since audit logging usually begins right after the host boots, this essentially amounts to assuming that the attacker hasn't gained root privilege before the machine even starts up.

In our system, audit log entries are securely transmitted to a remote server that stores them in a file. We assume that this remote server is beyond the attacker's reach, and is configured so that hosts can write new audit records but can never modify or remove existing records.

With respect to the connectivity to the remote server, we consider two cases below that we call Threat Models A and B. Both of them incorporate all of the assumptions described so far in this section.

Threat Model A: Always-on network connectivity. In this case, we assume a high-bandwidth, always-on connectivity between the protected host and a secure server. Under this threat model, WinSeal's tamper prevention techniques can be deployed in isolation. In most environments, it can provide sufficient security on its own, without requiring the use of tamper detection techniques. This is because log entries are written to the secure server within a very short tamper window. From this point on, they are out of reach for an attacker on the victim machine, including an attacker that has root privilege. (While an attacker with root privilege can corrupt audit records from that point on, they cannot corrupt or remove past records.)

Threat Model B: Intermittent network connectivity. Because network connectivity is intermittent, there may be long periods of time when the logs have to be stored locally on the protected host. To ensure its security, tamper detection techniques become essential. Tamper prevention techniques can optionally be deployed. It is particularly useful in settings where network connectivity is on most of the time, with occasional disruptions. In this case, the prevention features minimize the duration for which log entries are vulnerable to corruption or removal. During disconnected periods, tampering can be detected, but the entries remain vulnerable to removal.

To summarize, we expect WinSeal's tamper prevention techniques to be active on all hosts subject to threat model A. Optionally, detection features may be activated on a subset of hosts deemed to be of high value. The situation reverses for hosts under threat model B: detection features should be enabled for all hosts, while prevention features become optional. (In practice, due to the low cost of the prevention features, we recommend that it be left on for all hosts.)

2.2. Extended Berkeley Packet Filter (eBPF)

The eBPF framework [33], [36], [37], [38], integrated into the Linux kernel for nearly a decade, is enabled by default on most modern Linux distributions. It allows user-defined code, known as probes, to be safely attached to kernel events such as system call entries and exits. This flexibility

has made eBPF popular for system observability, network monitoring, and security analysis.

eBPF programs are written in a restricted subset of C and compiled into an assembly program that uses the eBPF virtual instruction set. Before they are loaded into the kernel, a formal verifier in the Linux kernel verifies key safety properties, including bounded execution time, absence of invalid memory accesses, etc. Once the program passes verification, it is translated into native code and then loaded into the running kernel.

To provide these guarantees, the eBPF verifier imposes strict constraints: it prohibits loops, while also limiting the number of instructions that can be executed. It caps the stack size at 512 bytes. Global variables are not permitted, so any data that needs to be shared across system call invocations must be stored in eBPF *maps*, which can either be arrays or hash tables. Of particular use are LRU maps that automatically evict the least recently used entries when an attempt is made to insert entries into a table nearing its capacity. Maps are stored in kernel memory, which is not subject to paging [39]. They can also be accessed by user level programs using the `bpf` syscall, and thus provide a mechanism to share data between the user level and the kernel.

The most problematic eBPF restriction for the purposes of this paper is that the eBPF code cannot make use of any standard libraries (beyond the few operations provided by the eBPF environment). While these restrictions ensure safe execution, they also make it impossible to implement most conventional cryptographic algorithms such as AES, SHA-256, or RSA. As a result, tamper detection schemes that rely on chaining MACs or computing complex hashes cannot be directly realized in this environment.

3. Motivating Experimental Study

The goal of our study is to demonstrate the stark *security* versus *deployability* choice faced in choosing a system for audit logging. Secure logging systems such as Quicklog2, HardLog, Omnilog and KennyLoggings aren't easy to deploy: they face either compatibility or performance challenges, or both. On the other hand, easy-to-deploy audit logging systems are weak against log tampering.

From a deployability perspective, the need for special hardware features are already acknowledged by the authors of HardLog and Omnilog, so there is no need for an experimental evaluation here. We do point out that compatibility concerns go beyond hardware:the need for a custom OS modifications are also a major factor that discourages deployment: these modifications are typically limited to specific kernel versions that may not match that of the system on which audit logging is to be deployed. For instance, KennyLoggings is available for just a single version of the Linux kernel, specifically, 4.15.0-47 that was released in 2018. Quicklog2 is also limited to a single version, Linux 3.10.0-1160, that was released in 2014.

We hence focus our motivating study on five audit logging systems whose software is compatible with mainstream hardware platforms:

Benchmark	Parameters
postmark	Ran postmark with 450 files, 15K transactions, using N parallel tasks with 9 iterations each.
redis	Ran N copies of redis-benchmark that exercised a single server. Each process modeled N clients and ran 1M set and 1M get operations.
httperf	Ran N copies of httperf that exercised a single `nginx` server. Each httperf copy sent 200 requests.

Table 1: Workload configuration for different benchmarks

- KennyLoggings [20] and Quicklog2 [19], two tamper detection systems;
- `auditd` and sysdig, two readily deployable audit logging systems that don't include specialized defenses against log tampering; and
- eAudit [27] that is deployable *and* incorporates some techniques for tamper window reduction.

3.1. Experimental Setup

All experiments were carried out on an i7-12700 processor (12 cores) with 64 GB of RAM and a 500 GB SSD running Ubuntu 22.04. KennyLoggings and Quicklog2 depend on older Linux kernels, so they had to be run within VMs on the same platform, using all 12 cores, almost all of the 64GB memory. Virtualization was provided through KVM/QEMU, which offers near-native performance. All systems were configured to record the same set of system calls, specifically, those logged by eAudit (Table 1 in [27]).

Benchmarks and Workloads. Table 1 lists the benchmarks and the workloads used. These benchmarks were taken from previous works [27], [20], [22], [21]. For each benchmark, we chose a single-core workload intensity matching that of a previous work. We then increased the workload in proportion to N, the number of cores. We varied N from 1 to 8. (We don't use $N>8$ because cores 9–12 are efficiency cores on our platform, and have a performance profile that deviates markedly from the trends observed for $N=1$ to 8.)

The logging overhead of all of these systems increases in proportion to the number of events logged. For this reason, we measure workload intensity in terms of the number of logged system calls made by a benchmark. For *postmark,* a widely used file system benchmark simulating a mail server, we matched its event rate with that indicated in [20] at 96K events/sec. For *redis*, an in-memory key-value store benchmark, we tried to match the event rate of 300K/sec used in [21]. For *httperf*, a web workload generator that we tested with `nginx`, we matched the event rate of 22K/sec reported in [22].

Postmark models a single-threaded server, so we simply ran N copies of the benchmark to model an N-core workload. In contrast, httperf and redis benchmarks target a multi-threaded server, so there was no need to run N copies. Instead, we ran N copies of the workload generator.

3.2. Overhead of Tamper Detection Systems

First, we studied the slowdown in the execution of workloads due to audit logging using KennyLoggings and

Quicklog2. Degradation is measured in terms of runtime *overhead* defined as follows:

$$\text{Overhead} = \frac{\text{Benchmark runtime with audit logging}}{\text{Benchmark runtime without audit logging}} - 1$$

We measured the actual time for benchmarks to complete, also called as elapsed time or wall-clock time.

Fig. 1 displays our experimental results. We show the average overheads observed across the three benchmarks. Note that Quicklog2's performance is consistently better than KennyLoggings. This is to be expected since Quicklog2 introduces several optimizations over KennyLoggings. However, the raw overheads are very high. At $N=1$, they slow down benchmarks by an average factor of $6\times$ to $10\times$. One contributing factor is that they both build on `auditd`, which is itself slow. However, it should be noted that Quicklog2 and KennyLoggings add substantial overhead on their own, bringing up the overhead from about 50% to 500% or more.

The overheads increase steadily with the number of cores, reaching about 3000% for $N = 8$. (Note the logarithmic scale on the Y-axis.) The overhead increases by approximately a factor of 7 for Quicklog2. This linear increase is explained by the fact that their implementation serializes cryptographic computations, so for $N=8$, it will take $7\times$ the time to compute the checksums. Note that KennyLoggings's degradation at about $4\times$, is less severe, perhaps because it starts at a higher baseline. Note that `auditd`'s performance also degrades nearly linearly with N, with the overhead increasing from about 40% to 350%. However, it is much faster than systems incorporating tamper detection, which are a further $10\times$ slower.

We note that the respective papers (i.e., [20], [19]) report overheads in the 10% range. We were able to produce numbers in the same range *when* the benchmark intensities were set as per their papers. For instance, Quicklog2 configured redis with just a single client that produced just 4K events per second. In contrast, we used N clients, which leads to system call rates that are orders of magnitude higher. Since the overheads increase linearly with the number of events logged, it should be no surprise that we observe much higher overheads in our experiments.

To provide some context about the intensity of these workloads, note that as we increase N from 1 to 8, the average event rate increases by $4\times$ to $8\times$, indicating that these loads are well within the peak capacity of the platform. Indeed, we consistently observe CPU utilization rates in the 30% to 70% range for all the benchmarks. A practitioner would want to test audit logging performance at these loads to ensure that its deployment would not seriously impair the workload processing capacity of their system. Indeed, eAudit's average overhead remains below 10% for $N=1$ to 8, showing that it is the implementation of tamper detection in these systems that is responsible for low performance.

3.3. Tamper Windows on Easily Deployable Systems

One of the first questions here is that of deciding when a tamper window is too long. We begin this section with a few real-world exploits that help us answer this question (§3.3.1). Next, we describe the measurement approach, and proceed to present tamper window measurements on existing audit loggers (§3.3.2).

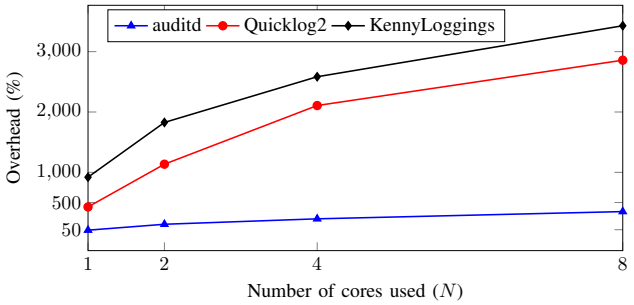

Fig. 1: Overhead of tamper detection techniques. Overhead of `auditd` is included to provide a baseline.

3.3.1. Using real-world exploits to assess tamper windows

As suggested in previous works [20], [21], one of the best ways to assess the size of a tamper window is to compare it with the time required to carry out an attack step. If an attacker gains root access in less time than the tamper window, they can terminate the logging process before records are written to disk, permanently erasing traces of the attack and thus evading post-attack detection.

To assess the tamper window, we study recent local privilege escalation vulnerabilities. These exploits represent realistic attack steps that a local or credentialed adversary might take to escalate privileges and immediately disable the logging process in order to erase evidence of their actions. For each exploit, we measure the time taken from launch to successful privilege escalation and termination of the logger. This duration serves as a benchmark to assess whether logs generated during the attack can be securely captured before the logger is compromised.

In order to exploit the tamper window, it is advantageous for attackers to select exploits that take the least amount of time. Accordingly, we selected the three fastest exploits out of 15 recent privilege escalation vulnerabilities on Ubuntu Linux 20.04 with a working exploit on our experimental platform. Table 2 lists these exploits. Each row shows a CVE identifier and the time from the start of the exploit to the time the logger is killed. *Note that the exploits complete in a small fraction of the tamper window that we report for eAudit,* `auditd` *and sysdig in the next section.* This means that the attacker can wipe out the entries pertaining to their exploit before they reach the disk.

3.3.2. Tamper window measurement

To measure the tamper window, we use the following procedure. A measurement script is started along with the

Sn	CVE	Exploit Time
1	PwnKit (CVE-2021-4034)	8ms
2	Dirty Pipe (CVE-2022-0847)	10ms
3	Sudoedit (CVE-2023-22809)	15ms

Table 2: List of CVEs with exploit time

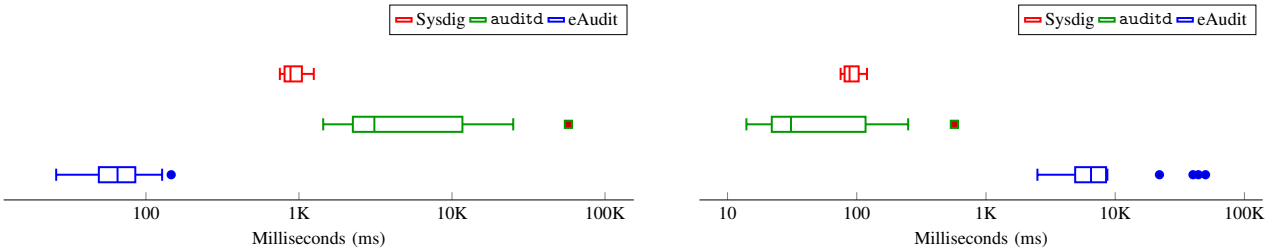

Fig. 2: Log tamper window size distribution for different systems under high load (left) and low load (right).

benchmark. Every second, this script records the current time t and the length l of the audit log. After the benchmark finishes execution, each such pair (t,l) is examined to determine the lag experienced in writing records at time t. This lag is given by the difference between t and the timestamp of the last audit log entry that occurs before the offset l.

Suppose that each run of the benchmark takes n seconds. We repeat the benchmarks m times, so that we have mn measurements. (For our experiments, we set $m = 10$.) Then we use a box plot to visualize these mn data points. As it typical, the whiskers in these plots go from 10 percentile to 90 percentile, while the bar goes from 25 percentile to 75 percentile. Some readings can fall well outside this distribution. Such significant "outliers" are depicted in the box plot using bullets.

Tamper window under high load. We first measured the tamper window under high workloads, as they represent the worst case for most audit logging systems. Intense workloads produce audit records at a rate that is hard to process and write to the disk. As a result, a backlog of in-memory event records build up, leading to a large number of buffered records that are yet to be written, and hence a large tamper window. Fig.2 highlights that `auditd` and sysdig exhibit large tamper windows ranging from 1 second to 10 seconds.

In contrast, eAudit is able to keep up with the rate of creation of audit log entries, so its internal queues don't build up. As a result, its tamper window stays well below that of `auditd` and sysdig. *But it is still* 5 *to* $10\times$ *the time taken to complete the exploits discussed in the previous section.* Thus, all three systems are highly susceptible to log tampering attacks.

Tamper window under low load. Interestingly, we observed the exact opposite behavior with eAudit: its tamper window is much larger under idle conditions than peak loads. (We elaborate further on this behavior in §4.1.) Fig.2 shows that `auditd` and Sysdig achieve significantly smaller tamper windows compared to eAudit under low load conditions. Both `auditd` and Sysdig perform well in this scenario, as the low rate of system calls enables them to process logs without significant buffering delays, resulting in small tamper windows. (However, they are still much longer than the exploit times shown in Table 2.)

In contrast, eAudit suffers under low load conditions due to its internal architecture that is tuned for high throughput. However, this leads to high latencies under low loads, leading to an average tamper window close to 10 seconds, with some outliers approaching 100 seconds!

3.4. Summary

Our study establishes that easy-to-deploy audit logging systems suffer from large tamper windows that are many times larger than the time to carry out several real-world exploits studied in this section. This means that skilled attackers can erase activities pertaining to their attack long before they reach the disk.

While tamper detection techniques offer the ability for deterministic detection of tampering attacks, our evaluation shows that previous works [20], [19] come with high overheads that slow down workloads by over $10\times$.

4. Techniques for Tamper Window Reduction

This section presents our techniques for achieving a tamper window that falls below the time to complete the exploits studied in §3.3.1. One way to do this is to study the reasons for the large tamper window in existing loggers, and to develop new algorithms that can substantially reduce the window. Among the audit loggers discussed in the previous section, eAudit comes with a much lower overhead than the other approaches, so it becomes our obvious starting point.

Recall that tamper window reduction techniques target our Threat model A, which assumes continuous and reliable network connectivity in order to promptly transfer audit log entries to a secure remote server. We begin this section with a short description of this network logging approach.

Logging to a secure remote server. In addition to the new capabilities we build on top of eAudit, we also configure it to log the audit records remotely. Logically, the remote server provides an append-only storage. Consequently, even if the attacker gains root privilege on the host running WinSeal, he/she cannot tamper with the stored log records. The host uses `ssh` to authenticate itself to the secure server and to encrypt audit records over the network. The server is configured to prevent any access by the host beyond using its append-only store.

4.1. Understanding eAudit's idle-time tamper window

Our investigation into eAudit's tamper window began with an unexpected observation: although eAudit's tamper window was small under intense loads, it experienced long delays during idle periods. This was first observed accidentally:in one of the runs, our measurement script failed to launch the benchmark, and we noticed that the

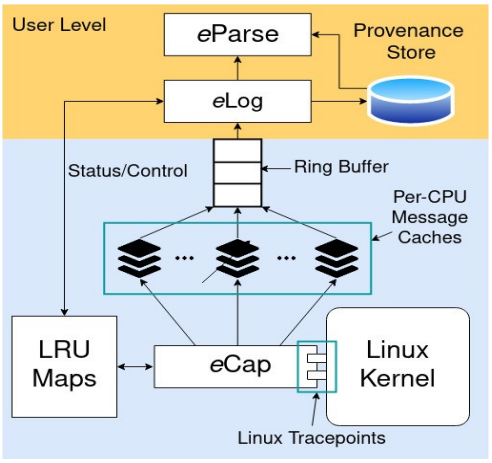

Fig. 3: Architecture of eAudit (reproduced from [27]).

measured tamper window was much larger than usual!

Through controlled experiments, we systematically varied the workload intensity by artificially slowing down benchmarks by inserting delays. The results revealed a counterintuitive pattern: eAudit's tamper window remained small during intense system activity, but gradually expanded as the workload intensity was decreased. The largest tamper windows occurred at the lowest intensity, which corresponds to background activities of core OS-related processes.

To understand the reason for this behavior, we looked further into eAudit's two-level queue structure shown in Fig. 3 (reproduced from [27]). The first level queue, called a *message cache,* is private to each CPU (i.e., each core). The second level queue is the ring-buffer data structure provided by eBPF. This buffer provides efficient shared-memory communication between the kernel and the user level.

eAudit's eBPF probes execute at system call entry and/or exit points to create in-memory audit log entries. These entries are first stored in the message cache, which is an eBPF map implemeting a simple array of bytes. eAudit uses a per-CPU map, which means that each CPU can access it without requiring locks or possible contention by other cores. The use of message cache greatly reduces the number of accesses to the global ring buffer, and is the biggest contributor to eAudit's high performance.

By default, eAudit sets the cache size to correspond to about 100 system calls. Under idle conditions, it may take a while for 100 system calls to be made. This provides a possible explanation of eAudit's large tamper windows under light loads. However, eAudit already has a mechanism that limits the maximum caching time to 16 milliseconds. After this, a cache is flushed to the ring buffer even if it is not full. In principle, this should limit the maximum tamper window to 16 milliseconds, but our experiments in §3.3 indicate that the tamper window can be as much as $1000\times$ larger than this!

After further analysis, we realized that the problem was the per-CPU nature of the message cache. In particular, under low loads, the operating system may use only a subset of CPUs at any time. Unused CPUs may remain idle for extended periods of time, during which no system call may be executed by them. Since a CPU's message cache is only checked by that CPU, message caches of idle CPUs may not be examined for very long periods (that approached many tens of seconds in our experiments).

To solve this problem, we redesign the message cache so that it is no longer private to each CPU. We also develop new mechanisms for active cores to flush all the message caches. We describe this new design below.

4.2. Message cache design and Flushing algorithms

One approach for overcoming the above problem is to modify the OS scheduler so that cores don't go idle, but this runs counter to our goal of avoiding changes to the OS. Moreover, such changes can have wide-ranging impacts, including power usage. Hence, we opt for a less intrusive approach that dispenses with message caches that are exclusive to each core. Instead, our new design uses shared message caches accessible to any of the N CPUs.

To reduce contention, our design ensures that CPU i will always log its system calls to message cache i. Even so, a shared message cache architecture implies that other CPUs may access it occasionally, and hence a locking protocol is needed to ensure safe access. However, routine locking can lead to contention and reduce performance, negating the whole purpose of message caches. Thus, one of our design goals is to develop a protocol that safely avoids locks on the most frequently used paths.

Next, we design *flushing algorithms* to ensure that all message caches are cleared periodically, with the log entries copied to the ring buffer. These flushing algorithms are *tunable*, with a user-specified *maximum cache time*. We designed and implemented four such algorithms, A_0 to A_3 as described below. *Note that A_0 is a baseline that is always enabled: each of A_1 through A_3 incorporates A_0.*

4.2.1. Self-flushing: Algorithm A_0

Self-flushing simply replaces the hard-coded 16 ms maximum time in eAudit for flushing the message cache with a parameter t_m selected by the sysadmin. We refer to this time as *max cache time.* Since it is just a parameterized version of a cache flushing algorithm already included in eAudit, we don't study self-flushing separately in this paper. Instead, we focus our evaluation on the three new algorithms described below.

4.2.2. Community flushing: Algorithm A_1

A_1 is a simple distributed cache flushing algorithm: each CPU C that makes a system call picks a second CPU C' for flushing. Note that CPUs are numbered from 0 to $N-1$, where N is the number of available CPUs. A_1 achieves strict round-robin semantics by setting $C' = S \bmod N$, where S is the sequence number of the system call. (eBPF's fetch-and-add instruction provides an efficient primitive for generating these sequence numbers.) If C' equals C, nothing is done.

Challenge: Avoiding lock contention. Before a core j can flush a cache $i \neq j$, it needs to acquire the lock for the cache i. Moreover, it is possible that before the flushing is

completed, core i may begin to process a system call and hence access its own cache. Because of this possibility, a core cannot access its own cache without a lock. However, if we require a lock for every access, then the situation may be no different from directly writing to the ring buffer. In that case, the need for locking on every system call led to an order of magnitude reduction in performance in eAudit. We therefore present a better alternative that avoids locking in most cases.

Our technique relies on the *age*[2] of the cache to determine if locking is required. If the age is more than t_m, the max cache time, then locks will be used, but not otherwise. Secondly, core j will flush a cache i only if the latter's age is above $1.5t_m$. This difference of $0.5t_m$ between the two thresholds ensures the safety of lockless access to the cache: in particular, concurrent access can happen only if a core begins to enqueue a message in its cache just before time t but does not finish until after $t+0.5t_m$. This is unlikely since message preparation times in eAudit are in the range of microseconds at the most, while the recommended value of t_m is in the range of a millisecond (as per §6.1.3).

This design avoids the use of locks under all but very light loads. Specifically, our experiments show that system call rates per core range from tens of thousands to several hundreds of thousands per second under moderate to high loads. This translates to tens to several hundreds of system calls over t_m at its recommended value of about a millisecond. All but one of these will happen when the age of the core's cache is less than t_m and hence will avoid a lock. In other words, somewhere between a fraction of a percent to 10% of the log entries will require locking. This fraction can increase to 100% under very light loads but this is acceptable since the system has enough spare capacity under light loads.

4.2.3. Flushing with a User-level daemon: Algorithm A_2

The combination of A_1 and A_0 ensures that under most loads, all message caches are flushed consistently. But it has its weaknesses with very light loads. This is due to the fact that each system call results in the flushing of just one other message cache. If the system is so lightly loaded, e.g., there is one system call every 10 milliseconds, then, on a 16-core system, as many as 160 milliseconds may go by before an inactive core's message cache is flushed. This problem can be addressed using a different approach: a user level daemon that periodically triggers the flushing of all message caches.

Algorithm A_2 consists of a user-level process that wakes up every f milliseconds, and notifies the kernel component to flush all message caches.[3] As with A_1, the age of each message cache is checked, and an actual flushing is initiated only if it is above $1.5t_m$. For this reason, values of $f < 1.5t_m$ aren't useful since the age check would fail, and hence no caches will actually be flushed. In order to have a reasonable probability of satisfying this age check, we recommend setting f to twice the value of this age threshold, i.e., $f = 3t_m$.

4.2.4. Hybrid flushing: Algorithm A_3

By combining A_1 and A_2, one can achieve low tamper windows across the entire spectrum of workloads, going from idle to peak workloads. However, when both algorithms are operational, it leads to an increased level of contention and hence increases performance overheads. Our hybrid algorithm is aimed at reducing this overhead.

The high level idea is to avoid running A_2 whenever A_1 is already effective. In particular, A_3 is a combination of A_1 and a modified version of A_2 as follows. We keep track of n, the number of system calls made since the last time A_2 triggered a message cache flush. If n exceeds the number of CPUs, then A_1 would have checked all the CPUs at least once in the last f milliseconds, so there is no need to check them for flushing again.

5. Tamper Detection

The techniques in the previous section are aimed at Threat Model A, with a high-bandwidth, always-on network connection. For Threat Model B, logs will need to be stored locally on a host during periods of network disruptions. Without additional protection, attackers can freely target and compromise these locally stored logs. Even for brief network outages lasting minutes, this translates to an expansion of the tamper window by several orders of magnitude (from milliseconds to minutes). Moreover, even for hosts that can count on full network availability, the techniques from the last section reduce but don't eliminate the tamper window. Although the publicly available exploits described in §3.3.1 took several milliseconds to succeed, one cannot rule out the possibility of faster exploits, possibly succeeding in the sub-millisecond timeframe.

To defend against all these attacks, we present new techniques in this section for *synchronous* yet *efficient* protection of log entries by attaching message authentication codes (MACs) to each entry. A new *signing key* is used for each entry in order to achieve forward secrecy, i.e., a compromise of a future signing key won't allow an attacker to tamper with the past records.

5.1. Key Challenges

Our goal is to combine known cryptographic primitives in novel ways to address the *practical challenges* that arise in realizing a *robust, efficient* and *deployable* tamper detection system. Specifically, we address the following challenges left open by previous works:

- *Efficiency and scalability:* Previous works [20], [19] compute per-event MACs using cryptographically linked keys. We rely on the same overall scheme, but address practical bottlenecks in their design. Specifically, in order to ensure correct ordering in a multi-threaded kernel, previous works serialized MAC generation via locking mechanisms, placing this logic inside the kernel's audit critical section [20]. This serialization prevents scaling

2. The age of a cache is defined to be the difference between the current time and the timestamp of the earliest message stored in the cache.

3. Notification takes the form of a kill system call made by the user level daemon targeting itself with (an invalid) signal argument of -1. The kernel component, when it sees this particular system call from the user-level daemon, initiates a flush of all cores.

on multicore workloads. In contrast, WinSeal requires serialization just for generating a 64-bit sequence number and picking a cryptographic key indexed by this number. Actual MAC computations proceed in parallel, enabling our approach to scale well to multicore workloads.

- *Supporting strong cryptography in an eBPF environment:* While eBPF is advantageous from a deployment and compatibility perspective, it does impose strict rules that prohibit loops. Deeply nested conditionals may run afoul of the verifier. No libraries can be used. Coping with these limitations require the MAC computation to be decomposed into two parts, one that operates at the user level and can use arbitrary computation, and a kernel component that uses a straight-line sequence of simple arithmetic and logical operations. We show that universal hash functions *can* be implemented under these constraints in such a way that they provide unconditional security based on an AES-derived cryptographically random message signing keys generated at the user level. The combination of user level and eBPF can thus provide strong security.
- *Key management and rotation:* Previous methods require an out-of-band mechanism to send the initial cryptographic keys to the audit log verifier. Successive keys used for MAC computation are all generated from this initial key. If the attacker somehow manages to obtain this key, he/she can change all the entries in the log, *including those that were made before the key was compromised.* Although this attack is considered out of scope in their threat model, its possibility cannot be excluded in real-world settings.
  In contrast to these works, WinSeal generates new random keys frequently, nominally every 2^{16} messages. A novel feature of our approach is that we rely on tamper prevention mechanisms from the last section to securely embed these keys within the audit logs, obviating the need for any out-of-band key transmission mechanism. (Any need for such a mechanism will discourage key rotation.)
- *Fault tolerance:* Under peak workloads, audit record volumes can be extremely high. Especially in cases where such a volume has to be logged remotely, one cannot rule out errors such as dropped records. Any such drop would break the signing key chaining mechanism in previous works. In contrast, since our approach associates signing keys with message sequence numbers, it can tolerate lost messages without compromising its ability to verify subsequent entries.[4]

5.2. Approach Overview

There are three main steps:

- *Signing key generation:* Starting from a seed key S_0, a sequence of signing keys K_i are generated. S_0 itself is obtained from a cryptographically secure random number source, specifically, `/dev/random`. To generate successive signing keys and seeds, we use an AES-based construction developed in Quicklog2 [19]:

$$\begin{aligned} K_i &= S_i \oplus AES(\overline{S_i}, F) \\ S_{i+1} &= S_i \oplus AES(S_i, F) \end{aligned}$$

  This construction uses AES with a fixed key F, which behaves like a random permutation over the space of all possible 128-bit blocks. Here, $\oplus$ denotes bit-wise xor operation, while $\overline{X}$ denotes the bit-wise complement of X. Quicklog2 guarantees *forward secrecy* in the construction of K_i (see QuickLog [19, §4, 5] for a proof), provided that previously used keys are securely deleted. Importantly, each key K_i reveals *no information* about any preceding keys. As a result, even if an attacker obtains a signing key K_i, they cannot forge or tamper with any records that were signed using a prior key K_j for $j < i$.
- *MAC generation:* In the second phase, MACs are computed and attached to each audit log entry. Since WinSeal constructs log entries in eBPF probes, MAC computation must also be implemented in eBPF code. The MAC algorithm used in Quicklog2 namely, AES with a fixed key, is inviable because AES can neither be implemented nor be called from eBPF code. KennyLoggings doesn't use AES either, but uses SipHash [40]. However, the authors of Quicklog2 argue the importance of avoiding less well-known cryptographic algorithms and using NIST-approved algorithms instead. Indeed, Quicklog2's main contribution over KennyLoggings is that of showing how AES can be adapted for this problem. For these reasons, falling back to SipHash wasn't a viable choice either. We therefore utilize an alternate construction, based on universal hashing, that lends itself to an effective eBPF-based implementation. An interesting property of this construct is that even if an adversary finds a key value that produces the correct MAC value of the current message, there is a very low probability that they found the "right" key that allows them to tamper with other messages.
- *Verification:* To verify the log integrity, the exact same procedure is used by the verifier, except that the initial seed S_0 needs to be provided. In this sense, S_0 is an *initial key* that needs to be sent securely to the verifier.

Since the verification step is straightforward, our description below focuses on the first two steps, with an emphasis on how the key challenges mentioned above are addressed.

5.3. Key generation and management at the user-level

To enhance the security of the whole scheme, we frequently rotate the initial key. Specifically, signing keys are generated in *batches.* Each batch starts with its own fresh cryptographically random seed S_0. Our implementation uses a tunable batch size of n, which defaults to 2^{16}.

Due to eBPF constraints, key generation and manage-

4. Note that such fault tolerance does have a downside: an unprivileged attacker can attempt to overload the system to the point that some of the entries in the tamper window will be dropped. Our key point is that it should be up to the sysadmins to decide whether to consider any missing record as a sign of an attack, rather than having the decision forced on them.

ment are handled by a user-level daemon, while MAC computations are performed by eBPF code in the kernel. The daemon preloads batches of signing keys into an eBPF map and monitors key usage to replenish them before depletion.

Generation of batch k begins by reading `/dev/random` and recording this seed S_0^k in the audit logs. (We explain below why it is safe to do this.) Next, K_i's are generated as described above and stored into the ebpf map. Successive values of S_i are stored in the same variable, so the random initial seed is immediately destroyed.

Note that the seed recorded in the *current* batch of log entries feeds into the signing keys for the *next* batch. We must consider the following three cases to establish the safety of recording the seed in the audit logs:

- *Case 1: The attacker already has root privilege at the time of new seed generation.* Per our threat model, there are no guarantees on the integrity of future log entries. In particular, the attacker can compromise the next batch of audit log entries even without capturing S_0^k.
- *Case 2: The attacker gains root privilege after seed generation but before it is stored on the secure server.* This means that the attacker gained root privilege within t_w after the generation of S_0^k. (Here, t_w denotes WinSeal's tamper window.) Batch sizes in WinSeal are designed so that, relative to t_w, the next batch will start far into the future. This means that the attacker can already compromise log entries in the *current* batch. As per our threat model, such attackers can compromise all future batches, regardless of whether they stole S_0^k or not.
- *Case 3: The attacker gains root privilege after S_0^k is recorded on the server.* In this case, S_0^k is already safe from being accessed or used by the attacker.

Thus, it is safe to record the seeds in the audit log. This feature enables WinSeal to frequently rotate keys without inconveniencing users.

Note that there is still the issue of the initial seed. When WinSeal starts up, it generates initial keys for the first and second batches simultaneously. Signing keys for the first batch are used immediately, but this is fine since our threat model assumes that attackers don't gain root privilege for several seconds after WinSeal starts up. As long as we switch into the first batch within this time, tamper detection guarantees are preserved. (The size of the first batch can be kept smaller than the default batch size if so desired.)

Threat Model A Vs B. In the context of Threat Model A, log entries are written out immediately to a secure server over an ssh connection. This implementation remains the same as §4. In the context of Threat Model B, two additional measures are needed in periods of network disruption. First, log entries are stored in local files during disruptions. Second, the rotation of seed keys is suspended. In particular, a new seed key will be generated as usual, but if its successful storage on the secure server isn't confirmed, then we won't switch to the new seed for the next batch.

5.3.1. Coordination between user and kernel components

An eBPF map, a shared-memory communication primitive, is used to send the signing keys from the user-level to the eBPF code that computes the MACs. Conceptually, this map is simply an array of size $2n$ that implements a circular queue. In particular, let the pair (s, i) denote the start of this queue. This means that the first element of the queue is at index i in the array, and it contains the signing key for the log entry with the sequence number s. For a sequence number $s' > s$, its signing key will be stored at index $(i+s'-s)$ **mod** $2n$.

Our user-level process generates signing keys in batches of size n. At the start, both s and i are zero. The queue is initialized with the signing keys for sequence numbers 0 through $n-1$. To compute the MAC for an entry with sequence number s', our eBPF probe reads the signing key from index $(i+s'-s)$ **mod** $2n$ and immediately overwrites it with zero. The user level monitors the consumption of signing keys, and when there are just $n/2$ of them left, it generates the next batch of n keys and adds them to the end of the queue. In addition, when the eBPF probe reaches the signing keys at index n, it advances the starting sequence number as well as the starting index by $n/2$.

As more and more signing keys get used up, the starting index advances in increments of $n/2$. Note that this approach allows for signing keys to be used out of order by as much as $n/2-1$. While such a large skew is unlikely, small differences in the order do occur because we cannot predict the order of completion of concurrent signing operations due to factors such as scheduling, log entry size, etc.

5.3.2. Secure key erasure

Although our code erases signing keys as soon as possible, factors such as compiler optimizations, virtual memory and the implementation of eBPF maps have the potential to leave behind sensitive data in memory or on the disk. To understand the issues involved, more information about our implementation is needed. Specifically, our implementation uses two variables `S` and `K` to store sensitive data. As noted earlier, successive S_i and K_i values update these two variables, thus ensuring prompt overwrite of these sensitive values.

Our implementation generates n signing keys at a time. As each key is generated, it is copied into an array `A`. For security, we allocate this array using `mmap` and then use `mlock` to lock it into the RAM, thereby preventing its contents from ever being stored on the disk. After populating this array, we use the `bpf` system call to copy signing keys into the eBPF map used in the kernel. (We found that this bulk-update of the eBPF map is many times faster than an approach that would use n separate `bpf` calls to write successive `K` values into the map.) When this syscall returns, we use `bzero_explicit` to securely erase `A`. Note that this function was introduced in GCC many years ago with the explicit purpose of securely erasing sensitive contents.[5] Otherwise, compiler optimizations can eliminate the code for erasing, based on the reasoning that

5. C23 standard introduces `memset_explicit` for the same purpose.

the new values are never used again in the program.

Once in the eBPF map, we need to ensure that signing keys are securely erased after their use. In this regard, note that we use eBPF array maps, which reside in non-pageable kernel memory [39] that is never stored on the disk, or moved in physical memory. This addresses OS-related concerns on secure erasure of map contents. Compiler optimization related concerns are also addressed because compilers cannot optimize away map operations. In particular, array maps store information shared across multiple eBPF programs. At the time of compiling or loading an eBPF program, future uses by another as-yet-unknown eBPF program cannot be determined, so its updates cannot be optimized away.

Similar to other implementations for supporting secure erasure (e.g., OpenSSL), there are two potential leak sources that aren't fully mitigated. First is that Linux *may* still choose to move an mlocked page in memory, and when it does so, it does not clear the old contents. However, such moves are rare. Second, compilers *may* spill sensitive values on the stack as a result of how register allocation works. However, since stack memory is not reallocated while a program is in execution, it is unlikely that an attacker's process will be able to get hold of this memory. Thus, in our opinion, neither source poses a serious concern, especially in light of the steps we have taken to minimize the lifetime of sensitive data.

5.4. MAC computation in the kernel (eBPF)

We address previously noted challenges, including (a) parallelism, (b) fault-tolerance, and (c) supporting sound cryptography despite eBPF restrictions.

5.4.1. Parallelism and fault-tolerance

As noted earlier, the key idea is to assign a unique 64-bit sequence number to each log entry, and select the signing keys based on this number. Once the sequence number is generated, the corresponding signing key is retrieved and then erased from the map. The rest of the MAC computation proceeds independently on different cores. With this approach, concurrency control is needed only in generating the sequence number. eBPF provides a highly efficient primitive for this, in the form of a *fetch-and-add* instruction.

For fault tolerance, note that the verifier uses the sequence numbers stored within the log entries to determine the signing key used in MAC computation. As a result, if one log entry is lost due to any reason, this will not affect the ability to verify the integrity of a subsequent message. This fault tolerant behavior of our approach contrasts with KennyLoggings and Quicklog2, where the index of the signing key is implicitly incremented after each audit log entry. Because of this, a lost message will lead to a loss of synchrony between the producer and the verifier, thereby causing verification to fail on all subsequent entries.

5.4.2. Realizing strong cryptography within eBPF

Similar to previous works [20], [19], we opt for a 64-bit MAC that is a good balance between security and storage overhead. But unlike previous approaches, we opt for a MAC scheme based on universal hash functions [41], [42]. This is due to two key reasons:

- its simplicity makes it a good candidate for an eBPF implementation; and
- it is unconditionally secure if signing keys are random and aren't reused across messages.

Specifically, we use a Carter-Wegman style MAC construction that is based on interpreting message blocks $m_0,...,m_r$ as coefficients of a polynomial $M(x)$:

$$M(x) = \sum_{i=0}^{r} m_i x^i \ \textbf{mod}\ p$$

This polynomial is evaluated at $x = K$, the MAC key. The evaluation is done in $GF(p)$, i.e., all arithmetic operations are carried out modulo a prime number p. This polynomial construction has been well studied in the literature [43], [44], [45], [46], [47], and results on collision probabilities and unconditional security established. Even so, it is useful to understand how these properties manifest in the specific setting of our MAC scheme, so we include a full justification below.

The collision probability of two distinct messages M and M' on a randomly chosen key k is very small:

$$Pr[M(k) = M'(k)] \leq \frac{r}{2^w}$$

(Here, w is the width of the MAC:64-bits in our work.) This is because a key that satisfies this condition is a root of the polynomial $M(x) - M'(x) = 0$. A polynomial of degree r over $GF(p)$ has at most r roots. So, at most r of possible 2^w keys will result in a collision between M and M'.

In our system, we split the 128-bit signing key K into two 64-bit halves K' and K''. The final MAC is defined as:

$$MAC(M,K) = MAC(M,K'||K'') = M(K') \oplus K'' \qquad (1)$$

(Here, $\|$ denotes concatenation.) Since K'' acts as a one-time pad on the evaluated polynomial value, the resulting MAC $\mathbf{M}$ can be any arbitrary 64-bit value. To confirm this, let $\mathbf{M}$ and $\overline{K}'$ be arbitrary 64-bit values, and pick $\overline{K}'' = M(\overline{K}') \oplus \mathbf{M}$. Now:

$$\begin{aligned} MAC(M,\overline{K}) &= M(\overline{K}') \oplus \overline{K}'' && \text{(by Eq (1))} \\ &= M(\overline{K}') \oplus M(\overline{K}') \oplus \mathbf{M} && \text{(substitute for } \overline{K}'') \\ &= \mathbf{M} && \text{(properties of exor)} \end{aligned}$$

Thus, an attacker able to produce a matching key $\overline{K}$ for an observed $(M, \mathbf{M})$ pair has gained no useful information about the real key K used to produce $\mathbf{M}$. In other words, $\overline{K}$ has a negligible probability (less than $r/2^{64}$) of being the right key. To see this, let $\mathcal{B}$ denote the set of all 64-bit blocks, and let $M(\mathcal{B})$ denote the image of $\mathcal{B}$ under the mapping M. Note that $M : \mathcal{B} \rightarrow M(\mathcal{B})$ is an onto mapping. Moreover, because of the property about the maximum number of roots of $M(x)$, at most r distinct elements of $\mathcal{B}$ can map to a single element of $M(\mathcal{B})$. Thus, $|M(\mathcal{B})| \geq 2^{64}/r$, i.e., when M is evaluated at all possible values of $\overline{K}'$, at least $2^{64}/r$ values will result, each corresponding to distinct

value of $\overline{K}''$. In other words, there are at least $2^{64}/r$ distinct values of $\overline{K}$ that will produce $\mathbf{M}$ from the message M.

One weakness of universal hash based MAC algorithms is that they are particularly vulnerable to attacks if keys are reused for multiple messages [47]. An algorithm such as SipHash is more robust against key reuse. However, our design has already opted for unique keys for each message in order to achieve forward secrecy goals mentioned earlier. In this setting, we find the simplicity and unconditional security of universal hashing based MAC to be compelling.

6. Experimental Evaluation

We first evaluate our techniques for tamper window reduction in §6.1. Next, we evaluate our tamper detection techniques in §6.2. Finally, we compare the performance of WinSeal with that of tamper prevention systems in §6.3.

Experimental setup. All experiments were conducted on a 12-core Intel i7-12700 machine with 64GB RAM and a 500GB SSD running Ubuntu 22.04. All log records are transmitted over an `ssh` connection to a secure remote server, which runs a vanilla Linux setup, with its ssh configuration set up to permit hosts to append to their audit logs but not to remove existing records or files. For the experiments, we use default parameters of $t_m = 0.67$ms and $f = 3t_m = 2$ms, and select the hybrid algorithm (A_3) as the default flushing strategy. The rationale for these parameter values is discussed in §6.1.3.

Benchmarks. To the list of benchmarks from Table 1, we added a fourth benchmark, sqlite. It models the performance of a database server. (This benchmark was omitted in the motivating study because it generates events at too high a rate for some of the compared systems.) As noted earlier, we choose system-call intensive multicore workloads in order to assess the worst case impact of audit logging. As shown in Fig. 4, the average syscall rate is about 250K/sec for $N = 1$, and increases to 1.27M/sec at $N = 8$.

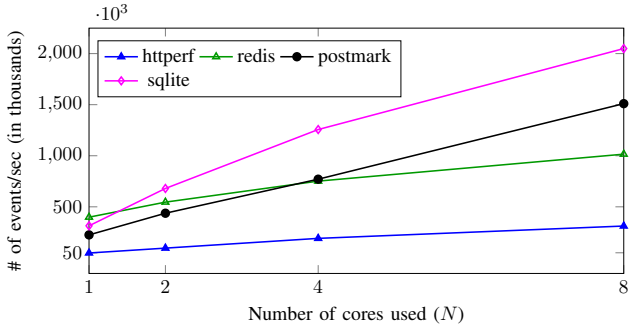

Fig. 4: System call rate for benchmarks

6.1. Evaluating tamper window reduction

Our evaluation answers the following questions:

1) How does the overhead of WinSeal compare with the state-of-the-art audit logging systems that are deployment ready? Our comparison includes eAudit, chosen because it has the lowest overhead among similar systems [27]; and `auditd`, because it is the oldest and most widely deployed audit collection system.
2) How effective are our techniques in reducing the tamper window of WinSeal? Since the tamper window of competing systems has already been studied in §3.3.2, we don't include them here.
3) How does one select the parameter values used for tamper window reduction? Which of the flushing algorithms offer the best combination of security and performance?

6.1.1. *Overhead of tamper window reduction*

Fig. 5 presents the runtime overhead across our four benchmarks. The average overhead across all of the data points is 9.5%. Remarkably, the average overhead rises by only 2.6% from $N = 1$ to 8, even as the event rate has climbed to about 6× the rate.

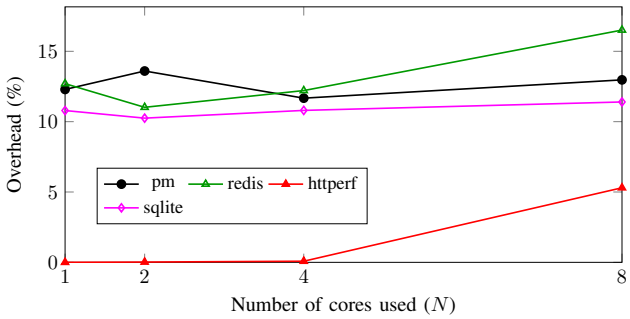

Fig. 5: WinSeal's overhead for tamper reduction.

The average overhead at $N = 1$ is 8.9%, and it falls slightly at $N = 4$ and then rises back to 11.5% at $N = 8$. This demonstrates that our flushing algorithms scale very well with N, and that the measures taken to reduce contentions have generally worked well.

Fig. 6 compares the overhead of WinSeal with eAudit and `auditd`. To reduce clutter, we plot just the average overhead across the four benchmarks. Our tamper window reduction techniques add slightly to the overhead, increasing it to about 9.5% from eAudit's 8.4%. As noted in previous works, `auditd`'s overhead is much higher, at an average of over 300%.

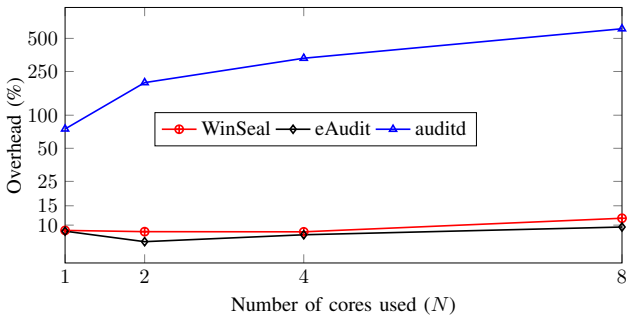

Fig. 6: Overhead comparison of WinSeal with eAudit and `auditd`

6.1.2. *Tamper window size*

As in the motivating study, we measure the tamper window separately under high load (i.e., with postmark using all

eight cores) and low load (i.e., no benchmarks).

High load. Fig. 7 compares the performance of the flushing algorithms A_1 (community flushing), A_2 (using user-level daemon) and A_3 (the hybrid scheme) under *high* loads. Note that our community and hybrid algorithms perform well, achieving a median tamper window under 1 millisecond. This is a major improvement over eAudit's 75 milliseconds (Fig. 2, on the left). In terms of the peak tamper windows, both the community and hybrid algorithms limit it to below 10 milliseconds, with the 90-percentiles at 6.5 and 4ms respectively. Note that the user-level algorithm performs somewhat worse, with a median of about 3 ms.

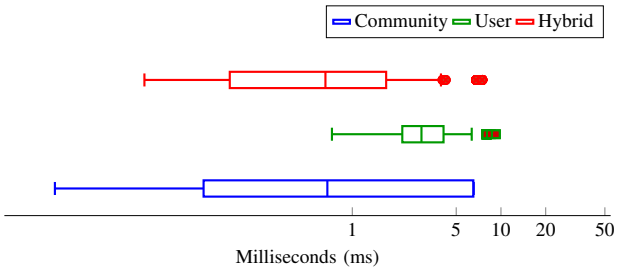

Fig. 7: Log tamper window size under high load conditions.

As in Fig. 2, these boxplots explicitly show outliers — formally, points below $Q1 - 1.5(Q3 - Q1)$ or above $Q3+1.5(Q3-Q1)$. All of the outliers in our measurements fall on the right. As a result, the average tamper window is generally higher than the median. This effect is most pronounced for the community algorithm, with a mean of 3 ms, which is more than $4\times$ the median. This skew is less pronounced for the user-level algorithm, with its mean of 3.23 ms only a bit above the median of 2.9 ms. The hybrid algorithm comes in the middle, with its mean of 1.52 ms that is about $2.3\times$ the median.

Although the community and hybrid algorithms have comparable median tamper windows under high load conditions, the hybrid's mean is $2\times$ better and its third quartile is $4\times$ better than that of community flushing. Hybrid's performance is also significantly better than that of the user level algorithm: $4\times$ lower median, $2\times$ lower average and $2.5\times$ better third quartile. From this analysis, the hybrid flushing algorithm emerges as the clear winner under high load conditions.

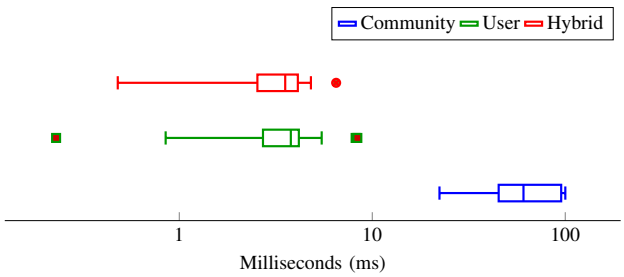

Fig. 8: Log tamper window size under idle conditions.

Low load. Under low loads, the user-level and hybrid algorithms perform well, achieving a median of 3.9 and 3.6 ms and a 90-percentile of 4.4 ms. Although community flushing performs worse, its median of 60 ms is still two orders of magnitude better than eAudit's 7 seconds.

Looking at the outliers, they are less pronounced under low load. Once again, hybrid flushing performs best. As compared to community flushing, its median, mean and third-quartile are reduced by factors of $17\times$, $20\times$ and $23\times$ respectively. As compared to the user-level algorithm, its median, mean and third-quartile are improved by 7%, 5% and 1.2% respectively.

Summary. The hybrid algorithm A_3 performs best under low as well as high workloads, with the 90th percentiles staying under 5 ms across varying loads. This falls well below the fastest exploit time of 8 ms in our evaluation. (See Table 2.) It is also lower than the 15 milliseconds targeted by the authors of HardLog *with* specialized hardware.[6]

6.1.3. Algorithm and parameter selection

In addition to the tamper window, performance overheads need to be factored into the choice of a flushing algorithm. Fig. 9 shows that the user-level daemon has the lowest overhead. This is understandable since it is triggered periodically, and does not add to the work performed on each syscall. In contrast, community flushing requires additional work on each syscall, and hence incurs a higher overhead. Hybrid flushing falls in the middle in terms of overhead.

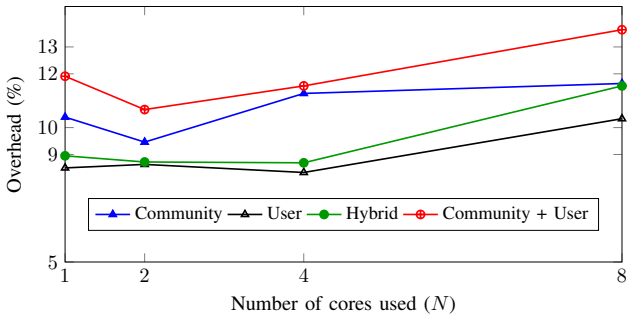

Fig. 9: Average overhead across all benchmarks for different algorithms.

Fig. 9 also shows the overheads when A_1 and A_2 are both run. We show this to illustrate the savings achieved by the hybrid algorithm A_3: although A_3 is functionally the same as the combination, our optimization (which avoids invoking A_2 when A_1 is already clearing the caches effectively) results in significant savings in terms of overhead. (Since overheads don't matter when the system is idle, we focus on performance measurements taken at high load.)

Next, we turn to the choice of the parameter t_m, also called the *max cache time*. Under consistent high load, the tamper window will closely track t_m because all the cores are frequently checking and flushing the message caches. However, under light loads, when system calls are sporadic, it is harder to predict the relationship between t_m and the tamper window size, so we used measurements. These are

6. HardLog also guarantees synchronous logging of so-called critical system calls, e.g., execve and kill. But it should be that eAudit [27] also prioritizes the same system calls and reports very small tamper windows for the same.

shown in Fig. 10. The figure shows that, in order to achieve an average tamper window below 10 milliseconds, t_m need to be set to 1 millisecond or less. We set it to 0.67 ms in order to add a margin of safety to our target of 10 ms.

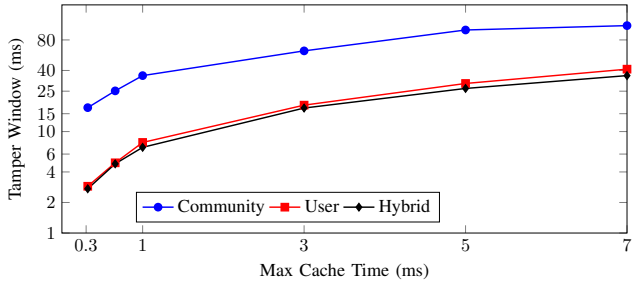

Fig. 10: Max cache time vs. Tamper window for different algorithms

6.2. Evaluating tamper detection

Tamper detection is secure by design, so our experimental evaluation is focused on the two key goals of *efficiency* and *scalability*. Fig. 11 summarizes our evaluation results. Tamper detection has a moderate effect on the overhead of WinSeal, increasing the overhead from 9.5% to 15%. This compares to the 2000% to 3000% measured in our motivating study (Fig. 1) on previous tamper detection techniques KennyLoggings and Quicklog2.

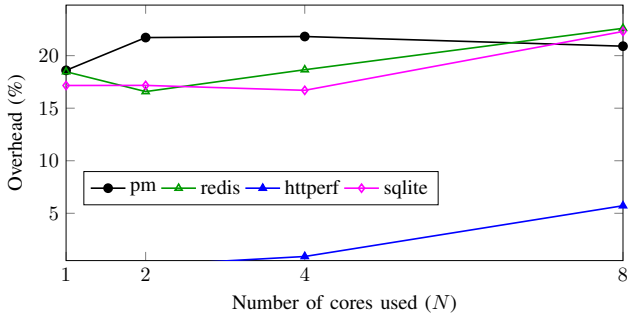

Fig. 11: WinSeal's Overhead as a result of tamper detection with remote logging.

Recall that our evaluation of previous work, KennyLoggings and Quicklog2 had to be performed on a VM due to their dependence on older Linux kernels. To assess the impact of using a VM instead of native execution, we reran the workload from Fig. 11 while WinSeal was deployed within a VM. This added another 5% to the overhead, showing that virtualization isn't a major contributor towards performance degradation.

Our results demonstrate that from a performance standpoint, tamper detection can be deployed on production systems handling intense loads. Fig. 11 also shows that our design successfully avoids serialization bottlenecks: the overhead grows only 4.3% as the workload increases from one to eight cores. There are variabilities across the benchmarks, with some showing an increase and others showing a decrease with the number of cores used.

Finally, we studied the impact of remote versus local logging on the tamper detection overhead. As shown in Fig. 12, the difference is marginal: the gap in overhead between remote and local logging is less than 1%.

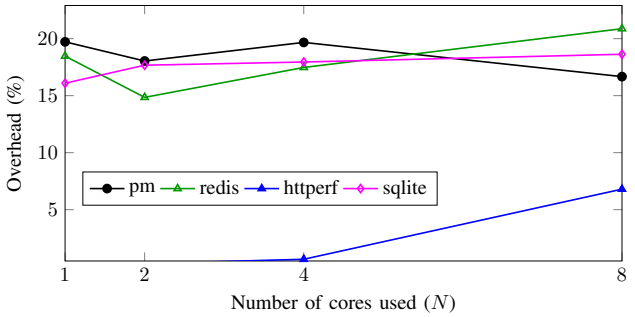

Fig. 12: WinSeal's Overhead as a result of tamper detection with local logging.

6.3. Comparison with HardLog and Omnilog

Since we do not have access to the specialized hardware needed to run these systems, our comparison relies on the numbers published in their respective papers, specifically, Fig. 5 in [21] and Fig. 4b in [22]. However, a direct comparison of the numbers is not meaningful: even though we share three benchmarks with them, there are large differences in the workloads used. So, we rely on an indirect method. In particular, both works include a comparison with the overhead of `auditd`. This enables a comparison on the ratio of overhead reported by the respective system as a fraction of the overhead incurred by `auditd`. The experimental platforms use between 4 and 6 cores, so we decided to compare the numbers for $N = 4$ in the case of WinSeal.

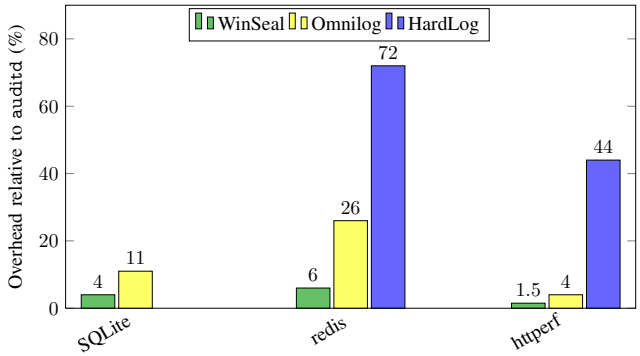

Fig. 13: Comparison of WinSeal overhead with HardLog and Omnilog

Fig. 13 shows the comparison. (Numbers for SQLite are too small to be readable in [21], so we don't show those numbers in the chart.) Note that WinSeal's tamper detection provides security that is comparable to that of these two systems, and is hence the most appropriate measure to compare. On this metric, WinSeal incurs an average of 3.3% of the overhead of `auditd`. (This is the geometric mean across the three benchmarks used in this chart.) In comparison, the geometric mean for Omnilog is 10% of `auditd`'s overhead, while it is 56% for HardLog. Clearly, there are caveats in this indirect comparison using `auditd`. Nevertheless, it is clear that WinSeal performs as

well or better than these two systems *without* requiring specialized hardware *or* OS-level modifications.

7. Related Work

Prior work in this area can be broadly classified into three categories: systems that offer only best-effort integrity, systems that prevent tampering through early persistence or hardware isolation, and systems that detect tampering through cryptographic verification. We next describe representative systems from each category and explain how WinSeal advances the state of the art by combining their benefits without inheriting their limitations.

Best effort. Systems in this category provide minimal or no protection against log tampering. Most existing audit logging frameworks fall into this class, allowing logs to be modified or deleted without detection. For example, Linux `auditd`, one of the most widely deployed audit logging systems, records comprehensive audit trails but does not cryptographically protect them, allowing a privileged attacker to alter or erase records without evidence. Similarly, eAudit [27], though designed for high-performance system-call logging using eBPF, does not incorporate any form log integrity protection, making it susceptible to log forgery or truncation. Sysdig [30], another popular system-call tracing tool, focuses on observability and debugging but lacks guarantees on the integrity or authenticity of the logs it generates.

All of these techniques could be combined with remote logging to a secure server, thus protecting log entries once they are transmitted over the network. However, as discussed before, system calls remain vulnerable for the duration of the tamper windows of these systems. As shown in Fig. 2, this can be as large as 1 to 10 seconds. eAudit's tamper window is also large, averaging 70ms at high loads and about 10 seconds at low loads. The window can increase further by several orders of magnitude if network connectivity is intermittent (i.e., under threat model B).

Similar to HardLog, eAudit does include a tamper window reduction technique for important system calls: system calls such as `execve` and `setuid` that are often used in privilege escalation are sent to the user level without any in-kernel buffering. However, the vast majority of system calls continue to experience long tamper windows. Consider an attack that involves the download and execution of malware. If operations such as read and write have longer tamper windows, an attacker may be able to wipe out any sign of a malware download from the logs, thus preventing an analyst from understanding the attack vector. For this reason, we have developed techniques for substantially reducing the tamper window for *all* system calls.

Tamper prevention. A complementary line of work seeks to prevent tampering outright by removing an attacker's opportunity to access raw logs, typically by storing or forwarding entries to a protected location immediately upon generation [48], [49]. SGX-Log [48] routes logs into an SGX enclave, computes MACs over the entries, and then exports the signed blocks to untrusted storage; however, it runs on a single host and performs sealing asynchronously via a user-space daemon, leaving a short interval in which unsealed entries can be modified or deleted before enclave commit. Custos [49] improves scalability, deployability, and verifiability, but retains the same user-space path and asynchronous commitment, leaving the exposure window between event generation and enclave commitment unaddressed.

To further shrink this window, HardLog [21] synchronously persists a subset of system calls to a write-only hardware device and provides a bounded 15 ms commit delay for the rest, but it requires specialized hardware and only some calls receive immediate, synchronous protection. OmniLog [22] synchronously protects logs by interrupting system call execution to transition into a privileged execution layer. It then copies each log record into a secure enclave that remains inaccessible to a compromised OS. However, this approach incurs substantial per-event interruption and copying overhead under high log rates. HitchHiker [50] improves upon this design by enforcing a strict, hardware-bounded protection deadline that revokes OS access to log buffers via memory-permission switching, thereby eliminating per-event interruptions and data copying from the syscall path. Although these systems improve coverage and usability, they still rely on specialized hardware, and hence aren't applicable to the vast majority of systems that run commodity hardware. Moreover, they do not address the problem of securely transmitting the log from the isolated environment to the site where attack detection and forensic investigation take place.

These tamper-prevention systems inspire our goal of minimizing the exposure window: we likewise strive to commit audit records to a safe location as soon as possible. At the same time, we address the key drawback of these previous works, namely, the need for specialized hardware. In particular, WinSeal's tamper window reduction and detection mechanisms offer a similar level of protection, while retaining their compatibility with commodity hardware and OSes, and incurring significantly ower overheads.

Tamper detection. Tamper-evident systems use cryptographic primitives to detect log manipulation after an attack, ensuring that any modification or deletion of prior entries is discoverable. The idea was first formalized by Bellare and Yee [18], [51], who introduced forward security for audit logs. A large body of work follows this paradigm [52], [53], [54], [55], [56]: log messages are recorded together with integrity proofs so that later tampering is revealed by verification. However, Paccagnella et al. [20] showed that such asynchronous designs suffer from large tamper windows. KennyLoggings eliminates this window by synchronously attaching a cryptographic signature to each log entry. Each log entry is authenticated using a forward-secure MAC chain where $K_{i+1} = H(K_i)$ (with BLAKE2), and the message tag $\text{MAC}_i = \text{SipHash}(K_i, M_i)$. After signing, K_i is securely erased, ensuring forward secrecy. Signing keys are precomputed in user space, while the MACs are computed in the kernel.

Building on this foundation, Hoang et al. [19] observed that KennyLoggings's reliance on non-standard

primitives (BLAKE2, SipHash) limited its compliance with government and industry cryptographic standards. Their successor system, QuickLog, replaced these primitives with an AES-based one-time MAC (*XMAC*) built from a fixed-key block cipher, and used AES-NI instructions for hardware acceleration. Unlike KennyLoggings, which derives each key via iterative hashing, QuickLog evolves keys through a pseudorandom function F built from AES using Even-Mansour construction, generating fresh states and keys for each log entry. Using a fixed AES key avoids repeated setup costs and allows pipeline execution on modern CPUs. An optimized variant, QuickLog2, further aggregates tags across multiple entries to reduce storage overhead and accelerate verification. Building on these systems, our design addresses the open challenges identified in §5.1: combining synchronous cryptographic protection with multicore scalability and verifier-safe operation inside the eBPF environment.

Nitro [57] is concurrent work that is also motivated by the performance bottlenecks of previous tamper detection techniques. Like WinSeal, it builds on eAudit's architecture and Quicklog2's overall cryptographic design, and hence the two systems share many common benefits. At the same time, there are several important differences as well. First, Nitro relies on the deterrence effect of tamper detection, and hence offers no protection against attackers whose sole interest is to hide the specifics of their attack. In contrast, WinSeal's remote logging and tamper window reduction techniques prevent such attackers from erasing any activity that happened more than a few milliseconds before their gain of root privilege. Indeed, Nitro's performance optimization techniques, while reducing overheads as compared to WinSeal (6% versus 15%), end up significantly increasing eAudit's already large tamper window. As a result, Nitro's tamper window is a thousand times larger than than of WinSeal.

Second, while Nitro protects the ordering among system calls executed by a single core, our design protects system call *ordering across different cores* through its association of signing keys with sequence numbers. Finally, Nitro doesn't provide some of the key features of WinSeal: frequent key rotation, in-band transmission of seed keys, and fault tolerance in the face of dropped records.

8. Conclusion

WinSeal advances the state of the art in audit log integrity by combining low-cost tamper prevention with efficient, cryptographically verifiable tamper detection—all within the constraints of commodity Linux systems. Building on eAudit's verified eBPF framework, WinSeal eliminates the need for specialized hardware or kernel modifications while offering protection comparable to trusted hardware solutions. Specifically, WinSeal offers the following benefits:

- *Low overhead and short tamper window:* Efficient cache and flushing algorithms reduce the tamper window to a few milliseconds with modest runtime overhead.
- *No specialized hardware or kernel changes:* Built entirely on verified eBPF probes, WinSeal runs unmodified on stock Linux systems, ensuring deployability across diverse environments.
- *Unified prevention and detection:* By integrating bounded-latency flushing with forward-secure cryptographic authentication, WinSeal provides continuous protection across both Threat Models A and B.

References

[1] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. Stoller, and V. Venkatakrishnan, "SLEUTH: Real-time attack scenario reconstruction from COTS audit data," in *USENIX Security*, 2017.

[2] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "NODOZE: Combating threat alert fatigue with automated provenance triage," in *NDSS*, 2019.

[3] P. Gao, X. Xiao, D. Li, Z. Li, K. Jee, Z. Wu, C. H. Kim, S. R. Kulkarni, and P. Mittal, "SAQL: A stream-based query system for real-time abnormal system behavior detection," in *USENIX Security*, 2018.

[4] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "HOLMES: Real-time APT detection through correlation of suspicious information flows," in *IEEE S&P*, 2019.

[5] M. N. Hossain, S. Sheikhi, and R. Sekar, "Combating dependence explosion in forensic analysis using alternative tag propagation semantics," in *IEEE S&P*, 2020.

[6] W. Ul Hassan, M. Lemay, N. Aguse, A. Bates, and T. Moyer, "Towards scalable cluster auditing through grammatical inference over provenance graphs," in *NDSS*, 2018.

[7] J. Zeng, Z. L. Chua, Y. Chen, K. Ji, Z. Liang, and J. Mao, "WATSON: Abstracting behaviors from audit logs via aggregation of contextual semantics," in *NDSS*, 2021.

[8] Z. Xu, P. Fang, C. Liu, X. Xia, Y. Wen, and D. Meng, "DEPCOMM: Graph summarization on system audit logs for attack investigation," in *IEEE S&P*, 2022.

[9] P. Fang, P. Gao, C. Liu, E. Ayday, K. Jee, T. Wang, Y. F. Ye, Z. Liu, and X. Xiao, "Back-propagating system dependency impact for attack investigation," in *USENIX Security*, 2022.

[10] J. Zeng, X. Wang, J. Liu, Y. Chen, Z. Liang, T.-S. Chua, and Z. L. Chua, "Shadewatcher: Recommendation-guided cyber threat analysis using system audit records," in *IEEE S&P*, 2022.

[11] X. Han, T. Pasquier, A. Bates, J. Mickens, and M. Seltzer, "UNICORN: Runtime provenance-based detector for advanced persistent threats," in *NDSS*, 2020.

[12] A. Alsaheel, Y. Nan, S. Ma, L. Yu, G. Walkup, Z. B. Celik, X. Zhang, and D. Xu, "ATLAS: A sequence-based learning approach for attack investigation." in *USENIX Security*, 2021.

[13] Y. Liu, M. Zhang, D. Li, K. Jee, Z. Li, Z. Wu, J. Rhee, and P. Mittal, "Towards a timely causality analysis for enterprise security," in *NDSS*, 2018.

[14] B. Bhattarai and H. Huang, "Steinerlog: prize collecting the audit logs for threat hunting on enterprise network," in *ACM CCS*, 2022.

[15] M. Rehman, H. Ahmadi, and W. Hassan, "Flash: A comprehensive approach to intrusion detection via provenance graph representation learning," in *IEEE S&P*, 2024.

[16] Z. Cheng, Q. Lv, J. Liang, Y. Wang, D. Sun, T. Pasquier, and X. Han, "Kairos: Practical intrusion detection and investigation using whole-system provenance," in *IEEE S&P*, 2024.

[17] L. Wang, X. Shen, W. Li, Z. Li, R. Sekar, H. Liu, and Y. Chen, "Incorporating gradients to rules: Towards lightweight, adaptive provenance-based intrusion detection," *arXiv preprint arXiv:2404.14720*, 2024.

[18] M. Bellare and B. Yee, "Forward integrity for secure audit logs," Citeseer, Tech. Rep., 1997.

[19] V. T. Hoang, C. Wu, and X. Yuan, "Faster yet safer: Logging system via fixed-key blockcipher," in *USENIX Security*, 2022.

[20] R. Paccagnella, K. Liao, D. Tian, and A. Bates, "Logging to the danger zone: Race condition attacks and defenses on system audit frameworks," in *ACM CCS*, 2020.

[21] L. S. Ahmad Adil and P. Marcus, "Hardlog: Practical tamper-proof system auditing using a novel audit device," in *IEEE S&P*, 2022.

[22] A. Varun Gandhi, Sarbartha and M. Adil, Sangho, "Rethinking system audit architectures for high event coverage and synchronous log availability," in *USENIX Security*, 2023.

[23] P. Jiang, R. Huang, D. Li, Y. Guo, X. Chen, J. Luan, Y. Ren, and X. Hu, "Auditing frameworks need resource isolation: A systematic study on the super producer threat to system auditing and its mitigation," in *USENIX Security*, 2023.

[24] G. Belding, "Ethical hacking: Log tampering 101," https://resources.infosecinstitute.com/category/certifications-training/ethical-hacking/covering-tracks/log-tampering-101/, 2020, accessed: 2025-03-26.

[25] S. Hales, "Last door log wiper," https://packetstormsecurity.com/files/118922/LastDoor.tar, 2015, accessed: 2025-03-19.

[26] K. Haniradi, "Mig log cleaner resurrected," https://github.com/Kabot/mig-logcleaner-resurrected, 2015, accessed: 2025-03-19.

[27] R. Sekar, H. Kimm, and R. Aich, "eAudit: A fast, scalable and deployable audit data collection system," in *IEEE S&P*, 2024.

[28] N. I. of Standards and Technology, "FIPS 197: Advanced Encryption Standard (AES)," https://csrc.nist.gov/publications/detail/fips/197/final, 2001, accessed: 2025-09-19.

[29] S. Grubb, "Linux audit," https://people.redhat.com/sgrubb/audit/, accessed: 2025-11-19.

[30] A. Kili, "Sysdig – a powerful system monitoring and troubleshooting tool for linux," https://www.tecmint.com/sysdig-system-monitoring-and-troubleshooting-tool-for-linux/, accessed: 2024-07-19.

[31] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, "Practical whole-system provenance capture," in *SoCC*, 2017.

[32] S. S. Lab, "eAudit: a fast, scalable and deployable audit data collection system," http://eprov.org/, 2024, accessed: 2024-04-26.

[33] eBPF, "eBPF — introduction, tutorials & community resources," https://ebpf.io, 2022, accessed: 2025-01-13.

[34] S. Ma, X. Zhang, and D. Xu, "ProTracer: Towards practical provenance tracing by alternating between logging and tainting," in *NDSS*, 2016.

[35] D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, "Hi-Fi: Collecting high-fidelity whole-system provenance," in *ACSAC*, 2012.

[36] Microsoft, "eBPF for Windows," https://github.com/microsoft/ebpf-for-windows, 2022, accessed: 2024-11-17.

[37] Iovisor, "High-level tracing language for linux eBPF," https://github.com/iovisor/bpftrace, 2022, accessed: 2024-07-21.

[38] B. Gregg, "Linux extended BPF (eBPF) tracing tools," https://www.brendangregg.com/ebpf.html, 2021, accessed: 2024-07-21.

[39] D. Alden, "A proposal for shared memory in bpf programs," https://lwn.net/Articles/961941/, accessed: 2025-09-16.

[40] J.-P. Aumasson and D. J. Bernstein, "Siphash: a fast short-input prf," in *INDOCRYPT*, 2012.

[41] J. L. Carter and M. N. Wegman, "Universal classes of hash functions (extended abstract)," in *ACM TOC*, 1977.

[42] M. N. Wegman and J. L. Carter, "New hash functions and their use in authentication and set equality," *Journal of computer and system sciences*, vol. 22, no. 3, pp. 265–279, 1981.

[43] B. den Boer, "A simple and key-economical unconditional authentication scheme." *Journal of Computer Security*, 1993.

[44] J. Bierbrauer, T. Johansson, G. Kabatianskii, and B. Smeets, "On families of hash functions via geometric codes and concatenation," in *Advances in Cryptology—CRYPTO'93*, 1994.

[45] T. D. Krovetz, "Software-optimized universal hashing and message authentication," Ph.D. dissertation, University of California, Davis., 2000.

[46] D. J. Bernstein, "Polynomial evaluation and message authentication," *Unpublished manuscript*, 2007.

[47] H. Handschuh and B. Preneel, "Key-recovery attacks on universal hash function based mac algorithms," in *Annual International Cryptology Conference*. Springer, 2008, pp. 144–161.

[48] V. Karande, E. Bauman, Z. Lin, and L. Khan, "Sgx-log: Securing system logs with sgx," in *ASIA CCS*, 2017.

[49] R. Paccagnella, P. Datta, W. U. Hassan, A. Bates, C. W. Fletcher, A. Miller, and D. Tian, "Custos: Practical tamper-evident auditing of operating systems using trusted execution," in *NDSS*, 2020.

[50] C. Zhang, J. Zeng, Y. Zhang, A. Ahmad, F. Zhang, H. Jin, and Z. Liang, "The hitchhiker's guide to high-assurance system observability protection with efficient permission switches," in *ACM CCS*, 2024.

[51] M. Bellare and B. Yee, "Forward-security in private-key cryptography," in *CT-RSA)*. Springer, 2003.

[52] J. E. Holt, "Logcrypt: Forward Security and Public Verification for Secure Audit Logs," in *AISW-NetSec*, 2006.

[53] B. Schneier and J. Kelsey, "Cryptographic Support for Secure Logs on Untrusted Machines," in *USENIX Security*, 1998.

[54] ——, "Secure Audit Logs to Support Computer Forensics," 1999.

[55] Balázs Scheidler, "syslog-ng: Open Source Log Management Solution," https://www.syslog-ng.com/, Accessed: 2025-10-09.

[56] Lennart Poettering and Kay Sievers, "systemd Journal: Logging and Journal Service for Linux," https://www.freedesktop.org/software/systemd/man/systemd-journald.service.html, Accessed: 2025-10-09.

[57] R. Zhao, M. Shoaib, V. T. Hoang, and W. U. Hassan, "Rethinking tamper-evident logging: A high-performance, co-designed auditing system," in *ACM CCS*, 2025.

Appendix A.
Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

The paper presents WinSeal, a system for protecting provenance logs by combining low-overhead tamper prevention with cryptographic tamper detection. Building on eAudit, it redesigns cache flushing to minimize tamper windows even under low workloads and introduces a parallelizable MAC scheme for efficient log authentication. Experiments show WinSeal achieves stronger integrity guarantees with minimal overhead and no need for specialized hardware or kernel changes.

A.2. Scientific Contributions

- Provides a Valuable Step Forward in an Established Field

A.3. Reasons for Acceptance

1) Practicality: WinSeal achieves shorter tamper windows and lower overhead than prior systems, which makes it a realistic and deployable solution without requiring specialized hardware or kernel modifications.
2) Strong Motivation: The paper clearly demonstrates the limitations of existing systems, notably eAudit, and uncovers a previously unknown issue with its log flushing algorithm through careful experimental analysis.