# HardLog: Practical Tamper-Proof System Auditing Using a Novel Audit Device

Adil Ahmad[†]
Purdue University

Sangho Lee
Microsoft Research

Marcus Peinado
Microsoft Research

***Abstract*—Audit systems maintain detailed logs of security-related events on enterprise machines to forensically analyze potential incidents. In principle, these logs should be safely stored in a secure location (e.g., network storage) as soon as they are produced, but this incurs prohibitive slowdown to a monitored machine. Hence, existing audit systems protect batched logs asynchronously (e.g., after tens of seconds), but this allows attackers to tamper with unprotected logs.**

**This paper presents HardLog, a practical and effective system that employs a novel audit device to provide fine-grained log protection with minimal performance slowdown. HardLog implements *criticality-aware log protection*: it ensures that logs are synchronously protected in the audit device before an infrequent security-critical event is allowed to execute, but logs are asynchronously protected on frequent non-critical events to minimize performance overhead. Importantly, even on non-critical events, HardLog ensures *bounded-asynchronous protection*: it sends log entries to the audit device within a tiny, bounded delay from their creation using well-known real-time techniques. To demonstrate HardLog's effectiveness, we prototyped an audit device using commodity components and implemented a reference audit system for Linux. Our prototype achieves a bounded protection delay of 15 milliseconds at non-critical events alongside undelayed protection at critical events. We also show that, for diverse real-world programs, HardLog incurs a geometric mean performance slowdown of only 6.3%, hence it is suitable for many real-world deployment scenarios.**

## I. Introduction

System auditing is an essential security component of modern enterprises. It requires keeping detailed logs of security-related system events (e.g., root command execution and kernel module installation) that occurred on each enterprise machine. These system logs are the basis for critical analysis including endpoint threat detection and post-mortem forensic analysis or attack reconstruction [1]–[8].

Several recent concurrent trends increase the need for system auditing. First, a pandemic-driven spike in remote work [9] has made other auditing approaches infeasible. For instance, enterprise network monitoring systems cannot protect machines connected to home networks. Second, a surge in sophisticated attacks [10], e.g., Advanced Persistent Threats (APTs), has lead to recent high-profile system compromises [11]. In response, strict auditing polices have been enacted at the enterprise and government levels to ensure resilience by unconditionally monitoring all machines. Notably, a recent US presidential executive order on cybersecurity [12] specifically mentions that government machines must maintain and protect system logs.

TABLE I
Overview of HardLog and audit systems that prevent log tampering using remote storage (RS), tamper-proof (TP) local storage, and/or tamper-evident (TE) integrity proofs.

| **Audit System** | RS (sync.) | TP (sync.) [13] | TE+RS [14]–[17] | HardLog [this work] |
|---|---|---|---|---|
| **Properties** | | | | |
| Log Availability | **Fine** | **Fine** | Coarse | **Fine** |
| False Alarm | **No** | **No** | Yes | **No** |
| Perf. Slowdown | High | High | **Low**/Med.[1] | **Low** |
| Storage Reuse | – | No | **Yes** | **Yes** |
| Secure Retrieval | – | No | **Yes**[2] | **Yes**[2] |
| **Requirements** | | | | |
| Secure Hardware | ○ | ● | ○ | ● |
| Kernel Change | ○ | ○ | ●[3] | ● |

[1] TE systems [14]–[16] that protect integrity proofs using Trusted Execution Environments (TEEs) incur extra overhead to invoke their respective TEE.
[2] TE+RS systems and HardLog ensure log retrieval under normal machine operation (i.e., before system compromise).
[3] TE systems compute integrity proofs inside the modified kernel to secure these proofs against race condition attacks [17].

Unfortunately, a persistent hurdle to effective system auditing is *log tampering*: once attackers have obtained root privileges on a machine, they can easily modify or delete logs to hide their compromise and subsequent malicious activities. Of additional concern is that log tampering is wide-spread—forensic analysts uncovered tampering evidence in 72% of conducted attack investigations [18]. This is not surprising because popular malware (e.g., BlackEnergy [19]) automatically deletes logs [20], [21] to hide their tracks.

In principle, audit systems can prevent log tampering by immediately storing each log entry in a secure location (i.e., remote storage [14], [22] or tamper-proof local storage [13]). In practice, this results in unacceptable overheads given the low performance of existing networking and storage devices for large numbers of small messages. Thus, the audit systems batch log entries for considerable amounts of time and store far fewer, larger messages *asynchronously*. However, this delay creates a time window for the attacker to tamper with the batched events. Finally, if logs are stored in commercial tamper-proof local storage devices [13], which lack authentication and storage reusability, administrators are unable to remotely manage logs (i.e., securely retrieve and discard old logs).

Traditionally, the main defense against log tampering has been *tamper-evident auditing* [14]–[17], [23], [24]. These

[†] Work done while this author was an intern at Microsoft Research.

systems maintain a cryptographic integrity proof which is updated for every new event [25]–[30] and allows a verifier to ascertain that the logs have not been tampered with. However, such systems do not prevent the attacker from interfering with forensics by deleting logs. They also produce false alarms since benign system crashes typically result in failed integrity checks [14], [16], [23], [24]. Table I briefly summarizes existing audit system properties and limitations.

This paper presents HARDLOG, an audit system that uses a novel *audit device* with tamper-proof storage to protect system logs. HARDLOG enables the following properties:

**P1 Fine-grained log availability.** HARDLOG ensures that log entries are protected either synchronously or within a tiny, bounded delay from their creation. Our prototype achieves a 15 ms bounded delay which is significantly smaller than the amount of time the state-of-the-art log deletion attack requires to succeed [17].

**P2 No false tamper alarms.** HARDLOG secures logs in its protected audit device rather than detects tampering. Thus, administrators are not burdened with false tamper alarms.

**P3 Modest performance slowdown.** HARDLOG introduces modest overhead to a monitored machine. Our prototype only incurs a geometric mean slowdown of 6.3% over non-audited execution on diverse real-world programs.

**P4 Remote device management.** HARDLOG allows remote administrators to securely manage logs under normal operation (i.e., before system compromise). After a compromise, HARDLOG keeps the logs safe until the system is recovered at which time they can be securely retrieved again.

Our design includes deployment-friendly specifications for a tamper-proof *audit device* (§VI-B). These specifications enable flexible device-based access control policies like *append-only* log storage and authenticated deletion, and mutually-authenticated secure connections with administrators for remote log retrieval. Importantly, the specifications only require commodity hardware, making it simple for hardware vendors to (a) integrate an audit device on future main boards (e.g., for enterprise laptops) and (b) manufacture standalone audit devices that can be attached to existing machines.

HARDLOG implements *criticality-aware log protection* to avoid the performance impact of synchronously protecting all logs while maintaining effectiveness (§VI-C). In particular, HARDLOG classifies forensically-relevant system events as *critical* and *non-critical*. Critical events are infrequent events that are known precursors to attacks (e.g., binary execution) [1], [31], [32]. HARDLOG synchronously protects logs collected so far on critical events to ensure likely compromises are always logged, and asynchronously protects logs on the frequent non-critical events (e.g., file access) to maintain performance.

Importantly, despite asynchronously protecting logs on non-critical events, HARDLOG provides *bounded-asynchronous protection guarantees*—all log entries for non-critical events are protected within a tiny bounded delay from the events' execution (§VI-D). To achieve such bounded delay on mass market hardware, HARDLOG employs various well-known *real-time* techniques [33], such as task preemption, controlled resource usage, and bounded thread execution.

We prototyped an audit device using a commodity development board, ROCKPro64 [34], to demonstrate its feasibility and deployability (§VII). Our prototype receives logs from a monitored machine's kernel audit components through a USB On-The-Go (OTG) interface. Our prototype is only one possible implementation—vendors can implement custom cost-effective devices based on HARDLOG's specifications.

We analyze the security of HARDLOG thoroughly (§VIII). Also, we stress-test HARDLOG's log protection pipeline to show that, even under extreme stress, HARDLOG ensures fine-grained log availability for non-critical events within 15 ms alongside ensuring full log protection for critical events.

We evaluate HARDLOG's performance using microbenchmarks such as lmbench [35], and on popular applications such as Firefox [36], GNU Octave [37], NGINX [38], Redis [39], memcached [40], SQLite [41], 7-Zip [42], and OpenSSL [43]. HARDLOG's geometric mean overhead across these applications is 6.3%, which is comparable to that of widely-deployed audit systems [44]. The source code of our prototype is publicly available at https://github.com/microsoft/HardLog.

## II. BACKGROUND ON SYSTEM AUDITING

System auditing (also sometimes referred to as system logging or system monitoring) refers to employing the operating system to maintain a history of **security-related events** that occurred on a machine. Such events include the execution of system calls and root commands by users or processes, failed login attempts, and kernel module installations. This history is recorded as **logs** where each **log entry** describes a security-related event through useful information like a timestamp, process or user context, and system call parameters.

In enterprise settings, forensic analysis and periodic machine health check are two of the most important use cases of system auditing. Forensic analysis is crucial to answering questions including how an attack was carried out or how the machine should be patched. Moreover, system administrators routinely (e.g., every day) retrieve logs to check an enterprise machine's health (e.g., check machine compromise status or suspicious program behavior). Although system auditing is also used for real-time attack detection, this paper focuses only on forensic analysis and periodic machine health checks (§V).

## III. MOTIVATION

A significant concern for system auditing is *log tampering*—once attackers obtain root privileges on a machine, they can modify or delete system logs, thus creating non-trivial hurdles for attack reconstruction and forensic analysis. This section describes how existing approaches try to address log tampering and why they have limitations.

### A. Synchronous Log Protection and its Limitations

Log tampering is prevented by *synchronously* storing logs—*as soon as a log entry is produced for a security-related event and before the event is allowed to execute*—in a secure

TABLE II
TIME TO INDIVIDUALLY SEND DIFFERENT DATA SIZES THROUGH VARIOUS DEVICE INTERFACES USING THE DD COMMAND WITH DIRECT I/O.

| | **1 block (512 B)** | | **2000 blocks (1 MB)** | |
|---|---|---|---|---|
| **Device interface*** | **Elapsed ($\mu$s)** | **Throughput (blocks/s)** | **Elapsed ($\mu$s)** | **Throughput (blocks/s)** |
| PCIe 3.0 x4 | 19.4 | 54k | 534.9 | 4194k (**78**×) |
| PCIe 2.0 x4 | 35.2 | 30k | 1253.1 | 1634k (**54**×) |
| USB 3.0 | 90.4 | 12k | 4378.9 | 489k (**41**×) |

*These measurements were performed on the machines we used for implementation and evaluation (§VII and §IX). For the PCIe 3.0 and USB 3.0 experiments, we connected NVMe SSDs to the main machine using respective links (specifications listed in §VII-A). For the PCIe 2.0 experiment, we connected an NVMe SSD to the other machine using the link (specifications listed in §VII-B). Each interface measurement was averaged over 2000 runs.

location that a compromised machine cannot access, such as a network storage server [45]–[47] or a local tamper-proof storage device (e.g., Write Once Read Many (WORM) drives [13]). Unfortunately, both solutions incur a prohibitive slowdown. Tamper-proof local storage devices also require constant physical device management, which is impractical in remote scenarios (e.g., work from home employee machines).

**Prohibitive performance slowdown.** Both network and local storage provide low throughput for storing small messages due to high communication latencies [48] and setup cost. For instance, when sending small (512 B) and large (1 MB) messages through various device communication interfaces available on our evaluation machines, the large messages showed up to 78× higher throughput than the small ones (Table II). Since modern machines can produce hundreds of thousands of log entries each second [14], synchronously storing each entry (usually smaller than 512 B) while pausing event execution incurs a prohibitive performance slowdown to monitored systems.

**Physical device management.** Another problem with commercial tamper-proof storage devices (e.g., WORM drives [49], [50]) is that they require constant physical access, which does not correspond to accelerated remote work trends. These devices do not allow remote administrators to securely retrieve logs or reuse device storage. In particular, the administrator must ask a potentially compromised operating system to retrieve logs from the device and there is no proof that the logs were retrieved from the tamper-proof device. Moreover, these devices are built for archival purposes; hence, they only provide *write-once* use and must be physically replaced when they are full.

### B. Asynchronous Log Protection and its Limitations

Given the performance cost of synchronously storing logs, all existing commercial and academic audit systems implement *asynchronous* log protection—*log entries are gathered in memory buffers while allowing the immediate execution of corresponding events and periodically sent to a secure location in large blocks after lengthy intervals (e.g., tens of seconds)*. However, since an adversary can tamper with logs within hundreds of milliseconds [17] after initiating an attack, it is challenging to trust such logs without additional protection.

To enable trust in asynchronously stored logs, existing audit systems [14], [15], [17], [23], [51] implement *tamper-evident* mechanisms to *detect* log tampering. In particular, they synchronously create an integrity proof for each log entry [25]–[30]. When entries are consumed for analysis, their integrity is checked using the corresponding proofs. If the integrity proof does not match or the proof has been deleted, the audit systems consider that the machine is compromised.

**Coarse-grained log availability.** Existing audit systems, even with a tamper-evident detection mechanism, do not provide fine-grained guarantees on log availability. Hence, they deprive administrators of the ability to analyze attacks. For instance, some systems only guarantee 10 s old log availability [14]. Such coarse guarantees fail to prevent an adversary from clearing attack traces in system logs—a motivated attacker only needs a few hundred milliseconds to clear its trace from logs [17].

**False tamper alarms.** Existing audit systems that use tamper-evident mechanisms tend to produce false tamper alarms after benign crashes [14], [16], [23], [24]. This is concerning because (a) system crashes are common [52], [53] and (b) administrators are already fatigued by many false threat alarms [7].

False alarms are raised when existing systems are terminated mid-computation due to a kernel panic or audit software crash, which causes transient in-memory integrity proofs or logs to be lost. This results in incorrect integrity check failures. It is non-trivial to differentiate integrity check failures from benign crashes and log tampering incidents [24], particularly because the administrator cannot perform conclusive forensic analysis using these *potentially tampered* logs.

## IV. DESIGN PRINCIPLES

Motivated by the limitations of log protection in existing audit systems (§III), this section presents three design principles for a practical and effective audit system.

### A. Criticality-Aware Log Protection

Events differ in their ability to indicate possible compromise. We classify them into (a) critical or (b) non-critical events, and provide different log protection guarantees accordingly.

We define critical events as *binary execution or permission change events (e.g.,* `execve` *and* `setuid` *system calls) whose intrinsic system role is to enable code execution*. After extensively analyzing real-world attacks and exploit payloads [54]–[56], previous studies [1], [31], [32] show that attackers mostly use critical events to execute malicious code at high privilege (e.g., to tamper with logs stored in host kernel memory). To prevent a critical event from corrupting logs, we should synchronously protect all logs stored in the host before executing the critical event. This synchronous protection ensures that *the full attack trace is available if the attacker takes control at a critical event*. Critical events are infrequent (according to our evaluation in §IX-D); hence, the performance impact of synchronously protecting logs at such events is low and limited in time.

We define non-critical events as *events whose intrinsic system role is not to enable code execution, but they are forensically-relevant because attackers use them to prepare* *attack payloads*. Examples include file access (e.g., `read`) and network operations (e.g., `sendmsg`). These events frequently occur in many program executions (according to our evaluation in §IX-D). Hence, given their less critical nature and high frequency, non-critical log entries should be buffered and protected asynchronously (i.e., without blocking thread execution) to limit their impact on performance. The buffered log entries are protected at a critical event or periodically (§IV-B).

### *B. Bounded-Asynchronous Non-Critical Log Protection*

In *rare* scenarios, even non-critical events can enable code execution and allow the attacker to tamper with logs. Imagine a catastrophic vulnerability in the `read` system call, resulting in arbitrary code execution. Therefore, we should protect all buffered non-critical log entries within a small bounded time interval. This protection ensures that *the attack trace up to a bounded time interval before the attacker takes control at a non-critical event* is available. In practice, as long as the bounded interval is small, an overwhelming portion of the attack trace is available for forensic analysis (§VIII-B).

### *C. Local Log Storage with Remote Management*

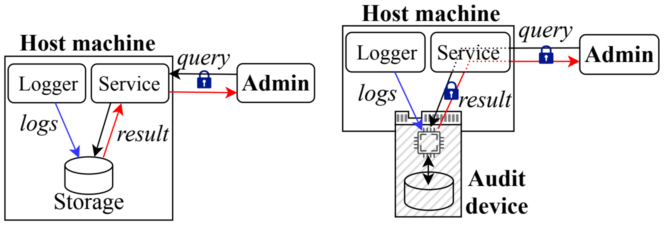
Protecting logs against host machine compromise requires the logs to be stored beyond the reach of the operating system. This storage location must be persistent to avoid losing logs on system crashes. Moreover, since many machines are remote, the administrator is expected to manage the protected logs remotely. In particular, remote administrators should be able to (a) securely retrieve logs and (b) discard stored logs when they are not needed anymore to reuse storage.

Since commercial tamper-proof devices lack remote management capabilities (§III-A), the remaining options are remote network storage or virtualization (e.g., trusted hypervisors or nested kernel [57]). These options make different trade-offs between performance, overhead when not in use, attack surface, system compatibility, and extra hardware requirements.

The feasibility of remote network storage depends on the host's network connection quality. Generally, network connections are too slow and unreliable in remote workplaces (e.g., employee homes) for this option to be feasible.

Remote storage problems are avoided by storing logs locally. If logs are stored in the machine's existing storage, the storage must be virtualized to restrict the operating system's access. However, such virtualization degrades throughput and latency noticeably for every operating system storage access, which is highly undesirable. Single Root Input/Output Virtualization (SR-IOV) [58] reduces this overhead, but only few server storage devices implement it. A simpler alternative is a separate storage device for logs and allowing the operating system *pass-through* access to its existing storage device. Nevertheless, this requires an extra dedicated storage device for logs.

Virtualization also imposes a non-negligible performance tax on the operating system, even if logging is disabled. For instance, the nested kernel (generally faster than trusted hypervisors) imposes up to a 20% performance penalty if invoked frequently [57]. Further, trusted hypervisors have attack surface and compatibility problems. For example, operating systems like Windows already leverage the hypervisor layer [59].



(a) Existing audit systems.

(b) HardLog.

Fig. 1. Comparison between system auditing models of widely-used existing audit systems [44] and HardLog.

Given that the most feasible existing option already requires extra hardware (i.e., virtualization with a separate storage device), we design new hardware that avoids its drawbacks. If the new hardware enables remote management and implements internal storage access control, it can (a) avoid system overhead when not in use, (b) have a clear small attack surface, and (c) be compatible with operating systems that are already virtualized.

## V. Threat Model

This paper considers a remote adversary who can *compromise the operating system* on an audited machine. Such an adversary obtains a non-privileged foothold on the machine (e.g., through a vulnerable network-facing process) and exploits a *privilege escalation* vulnerability in the operating system. The vulnerability exploit must (a) overcome standard defense mechanisms, such as Address Space Layout Randomization (ASLR) and non-executable pages, which takes time, and (b) execute security-related events that are logged by the audit system. The logs are kept in a place (e.g., kernel memory) inaccessible to non-privileged processes. However, once privilege escalation is achieved, the adversary attempts to tamper with logs as a *forensic countermeasure*.

Prior to operating system compromise, the audited machine is trusted and configured honestly (i.e., no rootkits or backdoors). Hence, the operating system correctly logs all security-related events and complies with all requests (e.g., task and resource prioritization) made by the audit system.

We assume that the adversary cannot compromise or corrupt the external audit device storing the logs even after the operating system is compromised. We discuss the device's concise and confined attack surface in §VIII-C. We ignore any log entries generated after an operating system compromise because they are not trustworthy. In addition, like prior audit systems, we do not consider physical attacks and make standard assumptions that the adversary is unable to break cryptographic protocols and compromise key management.

## VI. HardLog Design

HardLog is an audit system that leverages our proposed principles (§IV) to enable practical and effective log protection. With HardLog, an audited host machine stores its logs for security-related events in a new local *audit device*, which prevents overwrite of previous logs and allows remote administrators to securely retrieve logs and reuse storage (§VI-A). We also propose a small set of specifications for the audit device, requiring only existing well-known hardware and software features (§VI-B).

HardLog implements *criticality-aware log protection*, ensuring that the log entries for critical events are always synchronously written to the audit device (along with any non-critical log entries buffered in memory at that time). Conversely, to ensure minimal performance slowdown, frequent non-critical events are logged asynchronously (§VI-C), but with a *bounded-asynchronous* (§VI-D) guarantee—their log entries are written to the device within a tiny, bounded delay from their execution.

HardLog further enables system administrators to remotely retrieve logs from the audit device using a secure communication channel that is relayed by the host machine (§VI-E).

### A. System Model

HardLog considers an enterprise system auditing model (Fig. 1) with three participants: (a) host machine, (b) audit device, and (c) system administrator.

**Host machine.** This is an enterprise machine that must be audited (e.g., employee laptop or desktop, company servers). It has a system component called a *logger* that monitors security-related events and generates log entries for them. On machine startup, the logger waits for the audit device to be ready to receive logs before letting security-related events happen. Once the device is ready, the logger sends the logs to the device for secure storage. The host also runs an audit *service*, a communication relay between the device and the administrator that enables remote log retrieval.

**Audit device.** The audit device receives logs from the host machine via a device communication interface. Once logs are received, the device does not permit modification or deletion, except through commands sent by the system administrator through a secure cryptographic channel.

**System administrator.** The administrator consumes logs produced by the host machine for forensic analysis and periodic health checks. The administrator is remote and communicates with the host through the network. Then, the administrator uses the host machine's audit service (i.e., a relay) to communicate with the device. The connection between the device and the administrator is secured cryptographically. Through this secure connection, the administrator can retrieve system logs and ask the device to delete logs that are not required anymore.

### B. Audit Device Specifications

We design a *trusted* audit device which supports tamper-proof storage with remote device management, unlike existing tamper-proof storage devices. Importantly, the audit device is designed using commodity hardware features, so hardware vendors can easily implement it either directly on main boards or as a standalone device that can be connected to machines through a generic communication interface (e.g., USB). In fact, we built a prototype device using a commodity development board to demonstrate its feasibility (§VII). Note that vendors routinely implement extra security hardware (e.g., Trusted Platform Module (TPM) [60], Microsoft Cerberus [61], Google Titan [62], and Apple T2 [63]) for various machines.

Generally, the audit device has four specifications: (a) device-interposed storage, (b) multitasking processing unit, (c) device key pair, and (d) fail-safe power management.

**Device-interposed storage.** The device interposes between the host machine and the persistent storage medium under its control to store logs. Using trusted software, the audit device receives logs in its memory and then stores them in the storage medium. This interposition allows the device to implement comprehensive access-control policies on the logs.

The audit device enforces an *append-only* policy for all received logs, which ensures tamper-proofness of all stored logs. The append-only policy is enabled by a monotonically-increasing index to describe the position (e.g., a file offset) where an incoming log entry is stored. This index is safely stored inside the audit device and only the system administrator can re-initialize the index through a secure communication channel (§VI-E) to reuse device storage.

The storage medium for the logs is contained in the device. There are no special requirements for the storage medium—any regular storage medium like SD card, eMMC, or SSD is sufficient. In practice, however, if the storage is slower than the communication interface between the device and the host, it would affect performance. In particular, the host might have to wait for the device to be able to accept new log entries by flushing log entries in its memory to the storage (§VI-D).

When the storage medium is *almost* full, the audit device does not accept new log entries to preserve all stored log entries. Simultaneously, the device (a) sends a request to the system administrator using a secure communication channel (§VI-E) to ask them to retrieve stored logs and (b) signals the host that it has no storage left. Upon receiving this device signal, the host (using its logger) blocks any threads that execute security-related events to do not generate new log entries. The device reserves a small space (e.g., a few megabytes) in the storage medium to store the last few log entries generated before the host blocks the threads. After the administrator retrieves and deletes stored logs, the device allows the host to generate new log entries (e.g., resume the execution of blocked threads). A compromised host may ignore the signal and keep sending logs, but these are simply discarded by the device. As stated in §V, post-compromise logs have no value.

It is the administrator's responsibility to establish a logging policy that does not exhaust log storage before the logs are consumed (e.g., sent off to network storage). In general, such policies result in only up to a gigabyte of logs each day in enterprise machines [8], [14], [17]. Hence, in benign situations, even a moderately-sized (e.g., 64 GB) storage medium gives the system administrator ample time (e.g., a few months) to consume logs and ensure the storage medium is never full.

**Multitasking processing unit.** The audit device performs multiple tasks simultaneously, thus it should either be able

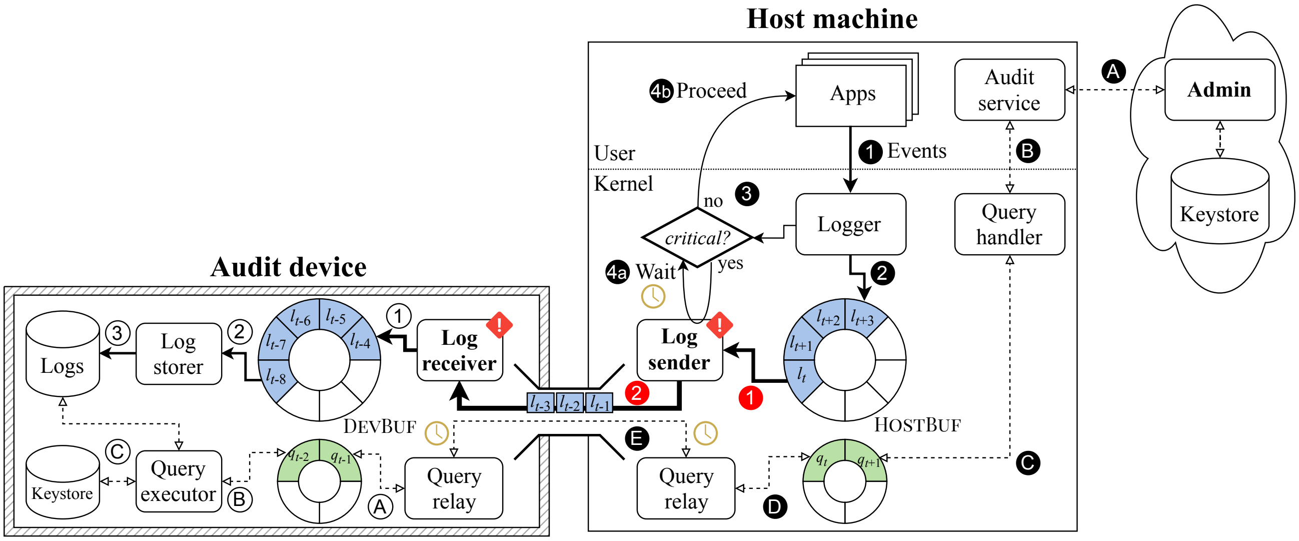


Fig. 2. Overall design of HardLog.

to time-share a single processing unit between these tasks or have dedicated processing units for each task. In particular, the audit device performs three tasks: (a) receive logs from the host machine in its memory, (b) securely store collected logs in its storage medium, and (c) respond to system administrator requests. In practice, since all tasks could simultaneously arrive, having dedicated processing units improves performance but it is not an essential requirement.

**Device key pair.** The system administrator and the device must mutually authenticate each other to realize secure remote management. To achieve this, while provisioning each audit device (e.g., the first time an enterprise host machine is set-up), the administrator generates an asymmetric key pair for the device and certifies its public portion using their signing key. The administrator also embeds their public key in the device to allow mutual authentication (§VI-E). These keys are kept in the device's storage, which is shielded from the adversary.

**Fail-safe power management.** The audit device can be powered by the host machine for normal operation, but it requires a power backup to gracefully handle power failures. Otherwise, the device could lose logs that are still in the device's memory or its monotonically-increasing index for append-only storage policy (mentioned prior). This is also important because the adversary might try to shut down the audit device as an attack (§VIII-A). In practice, the device only needs a very short period (e.g., 1 s) power backup to ensure graceful and secure shutdown, which can be achieved using a capacitor [64] even for our unoptimized prototype (§VII).

### C. Log Protection and Storage

HardLog employs criticality-aware log protection (§IV). On every infrequent critical event, HardLog synchronously sends the event's log entry and all buffered (non-critical) log entries to the audit device before the event is allowed to proceed and the adversary could potentially gain control over the machine. In contrast, on frequent non-critical events, HardLog asynchronously sends buffered log entries within a tiny bounded delay (discussed in §VI-D). HardLog ensures that the audit device flushes the received log entries, currently in device memory, to persistent device storage as fast as possible in an asynchronous manner.

**Synchronous critical event log protection.** Fig. 2 (❶–❹) illustrates how log entries are sent synchronously to the audit device on critical events. In particular, an application thread tries to execute a security-related event (e.g., system call), thus the operating system moves its execution to the kernel logger (❶). The logger creates an entry for the security-related event and stores it within HostBuf, a circular buffer that keeps transient log entries in the host machine's memory (❷). Then, the logger checks whether the event is critical based on a provisioned list of critical events (❸, §IX-A). If the event is critical, the logger waits for *log sender* to protect log entries (❹a). Log sender transfers all entries stored within HostBuf (i.e., prior non-critical entries and the current critical entry) to the audit device (❶, ❷). After log sender receives confirmation from the audit device, the logger allows the application thread to proceed and execute the critical event (❹b).

**Asynchronous non-critical event log protection.** Fig. 2 (❶–❹) also illustrates how non-critical log entries are created and sent asynchronously to the audit device. After creating a log entry for an event and storing it in HostBuf, the logger checks whether the event is critical (❶–❸). If the event is non-critical, the logger allows the application to proceed its execution (❹b) without waiting for log sender to have sent the non-critical entry to the audit device (i.e., skip ❹a). Log sender asynchronously sends the buffered non-critical log entries within a bounded time interval (§VI-D) (❶, ❷).

**Asynchronous log storage.** Fig. 2 (①–③) illustrates how all log entries (critical and non-critical) are asynchronously stored at the audit device. In particular, HardLog implements *log receiver*, a device thread that accepts log entries from the host machine (i.e., log sender). Log receiver stores all received entries in a circular in-memory buffer, DevBuf, instead of the contained storage medium (①). This allows log receiver to quickly acknowledge to the host machine that the log entries have arrived on the device (❷). The log entries (stored in DevBuf) are flushed to the storage medium by another thread, *log storer* (②, ③), which appends log entries to the storage using the monotonic index (§VI-B) and increases the index based on how many entries were stored.

### D. Bounded-Asynchronous Protection Guarantees

HardLog provides a *bounded-asynchronous* guarantee on non-critical log entry protection—each non-critical log entry is sent to the audit device within a tiny bounded delay from the execution of the corresponding non-critical event. HardLog achieves bounded-asynchronous guarantees by *controlling* each aspect of a non-critical log entry's protection pipeline, from its creation on the host machine to its storage within the device, using well-known *real-time* techniques. Such control restricts a non-privileged adversary and other background tasks from slowing down non-critical log entry protection. This is important since otherwise the adversary would attempt to detain as many attack-related log entries as possible in the host to tamper with them after gaining privilege [17].

The simplest real-time technique is to exclusively dedicate resources—CPU, communication interface, and storage—across a log entry's protection pipeline, but this is not always feasible since the resources are required for multiple purposes. First, the host machine's and audit device's CPU cores are shared with other user, system, and HardLog tasks. Second, the host uses the single communication interface to send both logs and administrator queries to the audit device as well as to receive responses from it. Third, the device uses its storage medium to store logs received from the host and retrieve logs on administrator queries.

To reason about controlling these resources given HardLog's constraints, we classify them based on whether they can be acquired (or *preempted*) when needed into (a) preemptible resources (i.e., CPU) and (b) non-preemptible resources (i.e., communication interface and storage). For preemptible resources, HardLog employs task prioritization to control these resources whenever needed. In contrast, HardLog controls all usage of non-preemptible resources to assign them to the critical tasks with a small, bounded delay. Furthermore, HardLog isolates micro-architectural components (e.g., Translation Lookaside Buffer (TLB) and caches) at the host to ensure that the adversary and background tasks do not affect log protection.

**Preemptible resource task prioritization.** HardLog employs CPU preemption to ensure that log transmission and handling proceed whenever necessary. For robust prioritization, HardLog ensures the following two configurations with the help of the host and device operating systems.

First, HardLog executes the log sender and log receiver threads at the highest priority on the host and device, respectively. Thus, other processes (both benign and malicious) cannot preempt the execution of these threads. Note that assigning such high thread priority requires root privileges [65], [66]; hence, it cannot be manipulated by the non-privileged adversary. However, a system might have multiple highest priority threads. In such scenarios, HardLog restricts the remaining (non-HardLog) highest priority threads to a number less than the machine's CPU cores, ensuring it can always preempt a CPU core (e.g., to run log sender when needed). Importantly, HardLog does not indefinitely reserve a CPU core. Instead, its threads go to sleep when no log entries are being produced, allowing other threads to use the CPU core.

Second, HardLog disables all hardware device interrupts, except interrupts from the audit device, and Inter-Processor Interrupts (IPIs) on a CPU core when it runs log sender. This avoids uncontrolled delays during log sender's execution (e.g., due to adversarial interrupts [67]).

Both hardware interrupts and IPIs are disabled using the Local Advanced Programmable Interrupt Controller (LAPIC) on CPU cores [68], [69]. Log sender can ignore IPIs [70] because they are for (a) process rescheduling, which must not happen to log sender (given its highest priority), (b) TLB shootdown, which is irrelevant to log sender because it does not share memory with other user or non-HardLog threads as well as it allocates or frees memory by itself, or (c) signaling, which is not required by log sender.

Although other interrupts—System Management Interrupts (SMIs) and Non-Maskable Interrupts (NMIs)—cannot be disabled, they are not useful to the non-privileged adversary. In particular, to arbitrarily insert SMIs via software (to stall log sender's execution), the adversary must modify the LAPIC configuration [71] which requires root privileges. Also, NMIs are generally raised for critical hardware errors (e.g., power and RAM failure [72]), so the machine halts.

**Non-preemptible resource usage control.** HardLog uses two non-preemptible resources, namely the host to device communication interface and the storage medium at the audit device. Since requests sent to these resources are handled by Direct Memory Access (DMA) engines, they cannot be *directly* preempted after requests are sent. This can result in *priority inversion* [73], where higher-priority requests (e.g., sending logs to the device) are delayed since they are enqueued behind lower-priority requests (e.g., sending queries to the device).

To prevent priority inversion, HardLog controls all threads that access non-preemptible resources, allowing it to implement *indirect* resource preemption with a tightly bounded delay. In particular, at the host, log sender and *host query relay*, a kernel thread that forwards administrator queries to the device, are the only threads allowed to send DMA requests to the device. At the device, only log receiver and *device query relay*, a thread that sends query responses to the host, can send messages across the device interface. In addition, only log storer and *query executor*, a thread that responds to system administrator queries, are allowed to send DMA requests to the storage medium. Please refer to §VI-E for additional details about the query relays and query executor threads.

HardLog uses *chunking* [74] to ensure that lower-priority threads (i.e., both query relays and query executor) breakdown their requests to non-preemptible resources into small chunks (512 B for storage and communication devices). These requests complete in a small bounded amount of time, allowing HardLog to preempt a lower-priority thread's execution and resource usage when higher-priority threads must execute.

Importantly, the interface between host machine and audit device must be dedicated for HardLog to avoid uncontrolled transmission delays. This is trivial if the device is embedded in a main board or connected using a link that provides a dedicated point-to-point connection (e.g., PCIe). For other shared bus interfaces (e.g., USB), the machine or user should ensure that no other device shares the same bus.

**Micro-architectural component isolation.** HardLog isolates the use of shared micro-architectural components (e.g., TLBs and caches) to ensure the non-privileged adversary cannot abuse them to slow down log sender's execution [75]–[78]. For instance, if Simultaneous Multithreading (SMT) is enabled at the host, the adversary might try to abuse sharing of *per-core* components like L1/L2 caches to evict log sender's cache-lines, which forces log sender to retrieve its cache-lines from memory. Similar eviction scenarios can occur at the *cross-core* Last-Level Cache (LLC) if the adversary accesses a lot of memory from another CPU core to fill up the LLC.

To isolate per-core components, HardLog preempts any thread that is executing simultaneously on log sender's CPU core when log sender is scheduled. This prevents the adversary from abusing any per-core components. Furthermore, to prevent the abuse of the cross-core LLC, HardLog employs *cache partitioning* using Intel Cache Allocation Technology (CAT) [68], which is widely available in recent Intel machines. This ensures efficient and comprehensive partitioning of log sender's LLC lines from all other CPU cores [79].

**Bounded thread execution.** Once HardLog ensures that the adversary and other tasks cannot interfere with its log sender and log receiver threads, as mentioned in the previous headings, it must bound the execution time of these threads to set the maximum protection delay for non-critical log entries.

Bounded thread execution for log sender and receiver requires (a) no indefinite code execution paths (e.g., no infinite loops without sleep or reschedule) and (b) a bounded amount of data is sent and received by these threads. The former is achieved by carefully writing the log sender and receiver code. The latter is achieved by controlling the HostBuf size to ensure, even in the worst case where HostBuf is full, log sender takes a fixed amount of time to send log entries.

After bounding thread execution, the maximum log protection delay $d_p$ is $2t_b + c$, where $t_b$ is the maximum time it takes for log sender to send the entire HostBuf across the device communication interface and receive an acknowledgment from log receiver, while $c$ represents the (constant) maximum time it takes to preempt other resources. It is $2t_b$ because, in the worst case, a non-critical log entry might arrive right after log sender has started to send the previous entries.

$t_b$ is decided by whether log receiver is ready to receive logs in memory (DevBuf). Log receiver is *always* ready if DevBuf is large and storage write throughput is higher than that of the communication interface, ensuring DevBuf always has free space. In this case, $t_b$ is the time to send HostBuf across the communication interface. Otherwise, $t_b$ is both the time it takes (a) to send HostBuf across the communication interface and (b) to flush DevBuf to storage at the device.

HostBuf's size also affects application performance. In particular, if HostBuf is full, the logger blocks application threads that execute *any* security-related events until all log entries kept in HostBuf are sent to the device in order to not drop them. Thus, the administrator should select HostBuf's size based on the maximum acceptable log protection delay and performance requirements. We show that our prototype implementation achieves a 15 ms bounded protection delay (§VIII-B) with only a 6.3% performance slowdown (§IX).

### *E. Remote Log Retrieval and Filtration*

HardLog enables the system administrator to securely retrieve all or a filtered set of log entries from the audit device through a remote connection. HardLog achieves this using (a) a host-based communication relay between the device and administrator and (b) a device query execution library that handles queries from the administrator.

**Secure host-based communication relay.** HardLog enables the host machine to forward administrator queries using the host's communication interface to the device. Importantly, the communication between the administrator and the device is mutually-authenticated and encrypted based on an Authenticated Key Exchange (AKE) protocol such as Transport Layer Security (TLS). The administrator can initiate the AKE protocol whenever they want to manage the audit device (e.g., retrieve logs and free device storage). The audit device does the same to notify the administrator when its storage is full.

HardLog uses the device key pair and administrator's public key provisioned on the audit device (§VI-B) to create the AKE channel, which is terminated *inside* the device. This is similar to how Intel Software Guard Extensions (SGX) employs the host machine as an untrusted network transport [80], [81]. Even after a host compromise, the attacker cannot pretend to be the administrator. Nevertheless, attackers can stop relaying network communication and prevent remote administrators from retrieving logs until a compromised machine is recovered (using mechanisms discussed in §X).

Fig. 2 (🅐–🅔, Ⓐ–Ⓒ) illustrates how the communication relay works. In particular, HardLog implements an *audit service* at the host machine which creates a network channel with the system administrator to receive queries (🅐). The audit service sends received queries to the in-kernel *query handler* (🅑) which buffers the query in host memory (🅒). Then, *query relay* in the host kernel sends the query to the audit device (🅓, 🅔). The audit device then receives the query (Ⓐ, Ⓑ) and executes it using *query executor* (Ⓒ). Afterwards, the

TABLE III
SOURCE LINES OF CODE [82] OF THE HARDLOG IMPLEMENTATION.

| Component | Base | SLoC |
|---|---|---|
| **Host machine** | | |
| Logger | Linux kernel audit | 1613 |
| Audit service | N/A | 148 |
| **Audit device** | | |
| Receiver | USB mass storage gadget | 626 |
| Query executor | N/A | 410 |

device sends back the query result to the host audit process which relays the result to the administrator.

**Device query execution.** HARDLOG implements a query execution library on the audit device to handle administrator queries, which supports log retrieval and filtration as well as authenticated log deletion. The library also compresses query results to reduce data transmitted through the network. Like some prior audit systems [44], HARDLOG supports the following parameters for log filtration: (a) process name or identifier and (b) event type. Also, HARDLOG allows the administrator to delete logs that are no longer needed to reuse storage on the audit device.

## VII. IMPLEMENTATION

We implemented HARDLOG for a Linux host machine to demonstrate its feasibility and effectiveness (Table III).

### A. Host Machine Components

**Hardware.** This machine featured an Intel Core i7-8700 CPU with 6 cores (12 SMT threads), 12 MiB LLC, 16 GiB RAM, and a 512 GB NVMe SSD (Samsung 970 Pro) connected through a PCIe 3.0 (x4) interface.

**Software.** The host machine ran Elementary OS 5.1.7 (Hera) with Linux kernel v5.4.97. We implemented two components for the host machine: (a) a logger and (b) an audit service.

The logger extends the Linux kernel Audit (kaudit) using a loadable kernel module to send log entries to the audit device (§VI-C). The logger (a) creates and manages HOSTBUF and (b) spawns log sender and host query relay threads (Fig. 2). Log sender and query relay use Block IO (BIO) [83], which bypasses the Virtual File System (VFS), to directly send the audit device log entries and queries.

The logger uses task priority and interrupt control to implement real-time techniques. Log sender is set to the maximum (*real-time*) priority using Linux scheduling primitives. To ensure selective interrupt delivery, the logger sets log sender's affinity to a CPU core and configures the operating system to only send audit device interrupts to the core when it runs log sender. Specifically, it uses `smp_affinity` to control hardware interrupts and `preempt_disable` to control IPIs.

The logger uses Intel CAT [68] to isolate the LLC. In particular, it uses Model-Specific Registers (MSRs) to allocate a small partition (LLC/8) to the CPU core where log sender is always scheduled and the rest to the other CPU cores. We experimentally verified (using the workloads in §VIII-B) that HARDLOG's log protection delay did not change when we partitioned log sender's core to LLC/8 (1.5 MiB) or left it unpartitioned. This shows that log sender's working set is smaller than 1.5 MiB. Based on our code inspection, it must be significantly smaller than 1.5 MiB because log sender only accesses a few statically allocated variables and BIO structures. More importantly, it is effectively constant: it does not depend on background tasks or anything the attacker could affect.

The audit service relays administrator queries to the audit device. It employs the IOCTL interface with the logger kernel module to send queries to the device and receive responses. This allows the logger to break down and pause query-related traffic to prioritize the sending of log entries (§VI-D).

### B. Audit Device Components

**Hardware.** We used a ROCKPro64 development board [34] for our audit device. The board is significantly more powerful than what HARDLOG requires (§VI-B)—we chose it because it supports a USB 3.0 OTG connection that we used as a device-interposed communication interface. The board has two big (Cortex A72 at 1.8 GHz) and four little (Cortex A53 at 1.4 GHz) cores with 2 GiB of on-board system memory. We attached a 250 GB NVMe SSD (WD SN550) to the board using its PCIe 2.0 (x4) interface. In practice, the audit device does not require such a large storage (§VI-B). We provisioned the device key pair in the storage.

Our prototype is powered independently from the host machine. Thus, the attacker cannot shut it down. Based on our analysis, a capacitor [64] can also power our prototype for a few seconds. This does not affect our evaluation.

The monetary cost for our prototype device was $120 ($60 each for the ROCKPro64 and storage). However, this is an unoptimized implementation with components purchased at retail cost. We expect that hardware vendors can build an optimized implementation at a significantly lower cost.

**Software.** The audit device ran Armbian 21.05.1 (Buster) with Linux kernel v4.4.213 and presented itself as a USB mass storage drive to the host. We implemented two components for the audit device: (a) log receiver and (b) query executor.

Log receiver extends the USB mass storage gadget [84] to implement append-only storage for incoming log entries and respond to queries from the administrator. Log receiver exposes three interposed device endpoints—log, query, and response—to the host. The log endpoint is for receiving log entries, while the rest are used for queries and responses, respectively.

Log receiver keeps a monotonically-increasing index to enforce append-only semantics for received logs. The index is persisted to storage. On startup, it (a) allocates DEVBUF to store incoming log entries in device memory, (b) spawns a *log storer* thread to asynchronously flush log entries from DEVBUF to a persistent file, and (c) spawns a *query relay* thread to deliver a query to query executor.

We configured DEVBUF's size to 256 MiB—this was enough to ensure the device always had space to keep incoming log entries in memory. As mentioned in §VI-D, if DEVBUF is small and storage write throughput is less than device communication

TABLE IV

Potential attacks on HardLog and defenses. The horizontal line demarcates attacks before and after privilege escalation.

| Attack | HardLog defense |
|---|---|
| **Log protection slowdown** | |
| Run many threads [17] | Prioritize log sender (§VI-D) |
| Raise interrupts [67] | Do not send to log sender (§VI-D) |
| Abuse micro-arch. components | Isolate log sender components (§VI-D) |
| Execute many events [17] | Bound HostBuf size (§VI-D) |
| **Log deletion** | |
| Delete entry at host | Protect before attack (§VI-C–§VI-D) |
| Delete entry in device | Require signed requests (§VI-B) |
| Shut down device | Use fail-safe power (§VI-B) |
| **Log modification** | |
| Insert entry in prior logs | Only append entries (§VI-B) |
| Reorder prior entries | Only append entries (§VI-B) |
| Change a prior entry | Only append entries (§VI-B) |
| Fill device with entries | Do not overwrite entries (§VI-B) |
| **Relay communication** | |
| Forge entries sent to sysadmin | Authenticated secure conn. (§VI-E) |
| Replay sysadmin commands | Authenticated secure conn. (§VI-E) |

throughput, DevBuf can get full. This is not the case for our prototype as the observed write throughput from DevBuf to the device storage (PCIe 2.0, $\sim$800 MB/s) is higher than the observed write throughput from the host machine to the audit device (USB 3.0, $\sim$240 MB/s) as shown in Table II.

Query executor responds to queries (i.e., log retrieval, filtration, and authenticated deletion) from the system administrator. For log filtration, it accepts two filters: process identifier and event type. Once a query is serviced, query executor compresses the response using the `pigz` compression software and writes the result to the response endpoint. While reading from or writing to storage, query executor breaks down its storage accesses into 512 B chunks, allowing rapid preemption (§VI-D). The host machine (using its audit service) polls the response endpoint periodically to check whether a response is ready.

## VIII. Security Analysis and Validation

This section analyzes HardLog's security through possible attacks, evaluates its bounded-asynchronous guarantees, and discusses its concise attack surface.

### A. Defense Analysis

The adversary (whose capabilities are mentioned in §V) can launch attacks before and after privilege escalation. Table IV provides a list of possible attacks and HardLog's defenses.

**Prevent log protection slowdown attacks.** Before escalating privilege, the adversary can stress HardLog's log entry protection pipeline (§VI-C) to delay protection of non-critical log entries to tamper with as many log entries as possible. They can try to (a) create many threads to stress CPU resources and potentially hinder log sender's execution, (b) raise frequent interrupts to slow down log sender, (c) abuse micro-architectural contention to delay log sender, and (d) execute many security-related events to cause a delay in sending HostBuf. To prevent these attacks, HardLog uses robust techniques (§VI-D), including real-time ones, to ensure that log protection is not affected by the adversary. Please refer to §VIII-B for an experimental validation.

First, HardLog ensures that log sender always executes at the highest priority and the number of non-HardLog threads at the highest priority is always smaller than the CPU cores. Hence, log sender is not preempted by adversary-controlled threads, no matter how many threads the adversary creates.

Second, HardLog ensures all IPIs and hardware device interrupts, except from the audit device, are not sent to the CPU core running log sender. Therefore, the adversary cannot slow down log sender's execution through these interrupts. Also, recall that the remaining interrupts (i.e., SMIs and NMIs) cannot be controlled by the adversary (§VI-D).

Third, HardLog isolates the per-core micro-architectural components (e.g., TLB and L1/L2 cache) and cross-core LLC partition used by log sender. Hence, the adversary cannot affect log sender's execution through these components.

Finally, HostBuf's size is fixed by the system administrator to a certain worst-case delay bound ($t_b$) for sending the entire buffer. If HostBuf is full, HardLog pauses application threads which try to execute security-related events to ensure that no log entry is dropped. Hence, even if the attacker executes many security-related events, they can neither slow down log protection nor cause log entries to be dropped.

**Prevent log deletion attacks.** After escalating privilege, the adversary can try to (a) delete log entries on the host machine (in HostBuf), (b) ask the audit device to delete logs, or (c) cut off the audit device's power to make the device lose its transient in-memory log entries (in DevBuf).

To prevent log deletion, HardLog ensures that logs are synchronously sent to the audit device before critical events are executed and the adversary potentially gains privilege (§VI-C). For non-critical events, HardLog ensures a bounded-asynchronous protection delay (§VI-D). Our experiments suggest that this delay can be as small as 15 ms for our prototype, suitable to prevent strong attacks (§VIII-B).

Once entries are protected inside the audit device, it does not accept log deletion requests that are not from the administrator. Since the adversary cannot circumvent the secure connection between the administrator and the audit device (§VI-E), they have no way to send valid log deletion requests.

In addition, HardLog requires that the audit device has fail-safe power management (§VI-B). Thus, transient in-memory logs (in DevBuf) are stored persistently, even if the adversary cuts off the device's main power supply.

**Prevent log modification attacks.** A privileged adversary can try to tamper with existing logs (already stored on the audit device). HardLog implements a simple defense for all log modification attacks: the device only accepts logs from the host in an append-only manner (§VI-B).

The adversary can also try to fill the device with log entries to overwrite their attack trace. However, if the audit device's storage is full, it will stop accepting new log entries. Hence, the adversary is unable to overwrite existing logs.

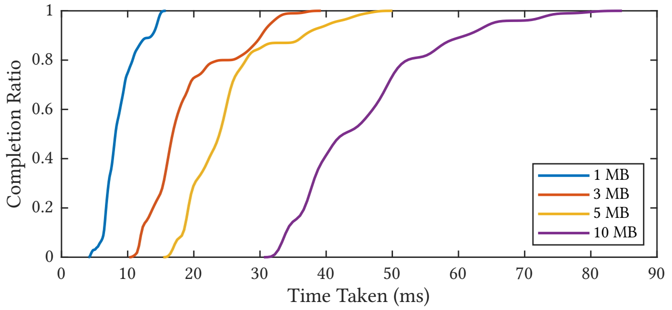


Fig. 3. CDF shows relation between HostBuf size and HardLog's bounded-asynchronous guarantees for non-critical events.

Recall that, if the adversary fills up device storage before compromise to evade attack logging, the host pauses any thread trying to execute security-related events until the administrator has freed up enough storage space on the device (§VI-B). Thus, no log entries before the compromise are lost.

**Prevent relay communication attacks.** A privileged adversary can attempt to subvert the host-relayed communication between the device and the administrator. In particular, they can attempt to forge query responses or replay previous commands. To prevent this, HardLog establishes a secure mutually-authenticated connection (§VI-E) between the device and the system administrator (using their public-private key pairs).

### B. Bounded-Asynchronous Guarantee Validation

HardLog provides a bounded-asynchronous delay on non-critical log protection (§VI-D). In this regard, this section first describes the maximum delay for non-critical log entry protection when the adversary, before escalating privilege, is actively trying to stress HardLog's log protection pipeline by producing many log entries. Then, it discusses why HardLog ensures real-world attack traces are protected even under asynchronous protection delays.

**Experiment description.** Our adversarial process spawns 12 threads, one for each logical core of our host machine (§VII). Each thread sequentially executes the `getpid` system call 10,000 times. Since `getpid` is the fastest system call [85], the machine produces log entries as fast as possible.

Note that this is an aggressive test of HardLog's protection pipeline by a non-privileged attacker. In particular, HardLog disables attacker-controlled interrupts against its log sender and isolates micro-architectural components (§VI-D), hence attacks through them are entirely ineffective.

Given bounded-asynchronous protections (§VI-D) and our implementation with sufficient DevBuf (§VII-B), the maximum log protection delay should be decided by HostBuf's size, hence we tested various sizes from 1 MB to 10 MB. We also configured the audit service to continuously send administrator log retrieval queries to the device for observing the impact of relay query communication on the maximum delay. We repeated this experiment 1000 times for each buffer and measured the maximum log protection delay each time.

**Maximum log protection delay results.** Fig. 3 shows the Cumulative Distribution Function (CDF) of non-critical log entry protection delay on each run with different HostBuf sizes. We observe that a 1 MB HostBuf results in a maximum delay of less than 15 ms for non-critical log entries, while a 10 MB HostBuf has a maximum delay of 83 ms. In addition, we noticed a negligible difference with or without background query communication, hence we did not show the curve without background query communication in the figure.

The observed maximum delay is close to the theoretical maximum delay, $d_p = 2t_b + c$ (§VI-D), as $t_b$ for a 1 MB buffer sent to our prototype audit device is ~6 ms (Table V). This shows that HardLog's real-time techniques (§VI-D) are effective at preventing the adversary from affecting the protection delay as we analyzed in §VIII-A.

**Attack trace protection.** Attackers must bypass existing kernel protections, such as Kernel ASLR and Supervisor Mode Access Prevention (SMAP), before having enough control over the system to destroy logs. This may involve many security-related events. For example, both in our experiments with real exploits [86] and in the proof-of-concept of prior research [17], we observe that attacks consist of long system call sequences which can easily take more than hundreds of milliseconds. Hence, even if attacks only use non-critical events and attempt to remove log entries immediately after gaining privileges, HardLog's small protection delay (e.g., 15 ms for a 1 MB HostBuf) ensures that the vast majority of attack-related events are logged safely.

Importantly, HardLog synchronously protects all buffered logs on critical events. Hence, for attacks where a machine is compromised at a critical event—most real-world attacks [1], [31], [32]—the entire attack trace is available.

### C. Attack Surface Analysis

HardLog's audit device attack surface is tiny and confined. In particular, unlike the host machine that exposes a broad collection of interfaces to potential attackers (e.g., applications and hardware devices), the audit device only has a single exposed interface (e.g., a USB interface for our prototype). Also, although we used Linux to rapidly prototype the audit device, in practice, verified operating system kernels [87] or tiny real-time operating systems [88] can support our simple requirements. The device can be further shielded using existing interface hardening techniques [89]–[92], verified parsers [93], and verified cryptographic libraries [94].

## IX. Performance Evaluation

This section describes HardLog's application performance (using several micro-benchmarks and real-world programs) and query handling performance.

### A. Setup

**Monitored critical and non-critical events.** We monitored forensically-relevant system calls using the rule set employed by prior audit systems [3], [14], [17], [95]. Amongst these, we selected 11 system calls to be critical since they are prevalent in exploit payloads [1], [31], [32], [54]–[56]. These system calls are for process creation, binary execution, or tracing (`fork`, `vfork`, `clone`, `execve`, `execveat`, and `ptrace`) and permission changes (`chmod`, `setgid`, `setreuid`, `setresuid`, and `setuid`). The remaining 33 system calls—`open`, `close`, `creat`, `openat`, `read`, `readv`, `write`, `writev`, `sendto`, `recvfrom`, `sendmsg`, `recvmsg`, `connect`, `accept`, `accept4`, `mmap`, `link`, `symlink`, `mknod`, `mknodat`, `dup`, `dup2`, `dup3`, `bind`, `rename`, `pipe`, `pipe2`, `truncate`, `sendfile`, `unlink`, `unlinkat`, `socketpair`, and `splice`—were configured as non-critical events. They include important file system and network system calls that are used by attackers to prepare exploit payloads.

**Audit system configurations.** We compared HardLog with Linux's default audit system, Auditd. We chose this system because it is widely deployed in enterprise machines (e.g., Red Hat Enterprise Linux uses Auditd [96]). Additionally, other audit systems [14], [17] are even slower than Auditd because they incorporate tamper-evident protection into Auditd. Hence, HardLog's comparison to Auditd also provides a performance reference against such audit systems.

Auditd employs a user-level daemon to collect and asynchronously store all log entries in the host's *unprotected* storage. With Auditd, the kernel keeps *all* log entries in an in-memory *backlog* queue before sending them to the user-level daemon. We configured a 1 million entry space in the backlog queue (taking up to ~300 MB memory) and ensured that Auditd waits for entries to be flushed if the queue is full.

We configured HardLog with a 1 MB HostBuf, ensuring a small maximum protection delay of 15 ms (§VIII-B). In practice, keeping a larger buffer improves performance at the cost of a higher maximum protection delay and vice versa.

### B. Synchronous Log Protection Microbenchmarks

HardLog pauses application threads to synchronously protect logs on critical events, which reduces performance. This section quantifies the pause time for synchronous protection. Auditd does not synchronously protect any logs, so it is irrelevant to this experiment.

**Settings.** On a critical event, the pause time depends on (a) the number of log entries residing in HostBuf and (b) the speed of the storage device and interface. To measure the worst-case protection time, we ran a workload that executes numerous non-critical events (i.e., `getpid`) to fill HostBuf with non-critical log entries and then executes a critical event (i.e., `execve`). At the critical event, we measured the time taken to send all buffered log entries to a storage device using DMA requests. This is the additional time that a thread is paused for. We repeated this experiment with various HostBuf sizes (1–10 MB). For comparison, we used four different storage devices—state-of-the-art commercial WORM 32 GB USB 2.0 drive [49] and SD card [50], a Toshiba 256 GB BG3 NVMe SSD [97] connected using a USB 3.0 enclosure, and our prototype USB 3.0 audit device. We ran each experiment 100 times and averaged the results.

**Results.** Table V shows the results. The audit device's pause time is up to 23.6× and 3.1× lower than that of the WORM drives. This is expected since both WORM drives are built for archival purposes and not optimized for fast storage. Compared to the USB 3.0 SSD, the audit device takes up to 40% more time. We believe this is because the SSD firmware is specialized for fast storage operations, unlike our audit device's USB mass storage gadget [84] that emulates USB storage.

TABLE V
Average synchronous log protection time for various HostBuf sizes and storage devices.

| **Storage Device** | **Delay (ms) per HostBuf size** | | | |
|---|---|---|---|---|
| | **1 MB** | **3 MB** | **5 MB** | **10 MB** |
| WORM USB 2.0 Drive [49] | 128.2 | 364.6 | 651.0 | 1392.3 |
| WORM SD Card [50] | 23.9 | 52.4 | 96.2 | 228.4 |
| USB 3.0 NVMe SSD [97] | 4.2 | 12.8 | 21.4 | 46.1 |
| HardLog's USB 3.0 Audit Device | 5.9 | 16.7 | 28.7 | 56.5 |

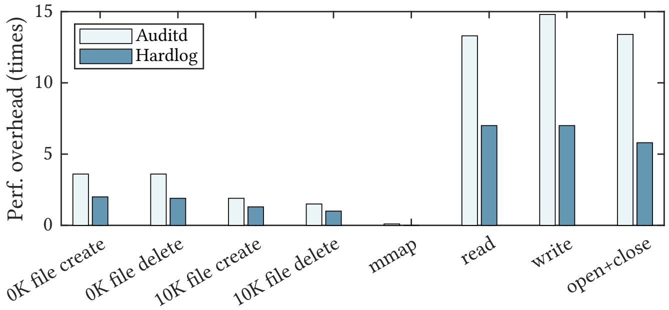


Fig. 4. Comparison between HardLog and Auditd on lmbench file creation and system call benchmarks.

### C. lmbench System Call Benchmarks

Since systems calls are typically monitored by enterprise audit systems, we measured the slowdown of individual system calls using both HardLog and Auditd.

**Settings.** We ran lmbench [35], a popular operating system test suite. In particular, we ran its latency benchmarks for (a) file system creates and deletes and (b) general system calls, which were most relevant to our monitored events. Note that all these benchmarks produced non-critical events only, and both HardLog and Auditd asynchronously handled them.

**Results.** Fig. 4 shows the results of our experiment. In particular, over native (non-audited) Linux execution, HardLog incurs 0.0–6.9× performance overhead while Auditd incurs 0.1–14.8× performance overhead.

This performance overhead is expected because, for both HardLog and Auditd, the Linux kernel performs two heavy tasks [14]: (a) passes each system call invocation through a filter to determine whether it must be monitored and (b) creates an ASCII-based log entry (of 200–1024 B) for each monitored system call. The log entry has this size since it contains detailed information to aid analysis, such as a timestamp, system call parameters, and process (or user) context (§II).

In addition, we observe that some tests (`read`, `write`, and `open+close` in Fig. 4) incur higher overheads than the rest. The reason is that these tests are very short system call tests, which perform fast in-memory computations. For example, the `read` test measures how long it takes to read one byte from `/dev/zero` while the `write` test measures the time to write one byte to `/dev/null` [98]. Thus, the performance impact of filtering and creating a log entry for these tests is high.

Interestingly, HARDLOG's overhead is up to 2.1× *lower* than AUDITD's in these tests. Our analysis revealed two main reasons for such lower overhead.

First, AUDITD sends log entries from the kernel to its user-level daemon—to store them in host storage—via the Linux Netlink interface [99]. However, AUDITD's use of Netlink is inefficient because it encapsulates each log entry into a network packet along with a large packet header [100].

Second, AUDITD uses the Linux event queues for its backlog queue, requiring multiple memory copies to enqueue or dequeue log entries. In contrast, HARDLOG directly sends log entries from the kernel to the audit device using a circular buffer (i.e., HOSTBUF) that is directly accessible to the DMA engine. Thus, it realizes *zero-copy* within the host kernel memory.

### D. Real-world Programs

We evaluate HARDLOG's real-world performance using eight popular applications in various categories for enterprise machines and compared this with AUDITD.

**Settings.** Web browsers like Firefox [36] and scientific computation software like GNU Octave [37] are common productivity tools. We evaluated Firefox using the Speedometer [101] benchmark which measures the responsiveness of web applications. For GNU Octave, we used the built-in benchmark to run reference GNU Octave scripts.

Network-facing programs like web servers are employed in many enterprise servers and face threats from remote attackers. Thus, we evaluated NGINX [38] using the ApacheBench (ab) benchmark [102] to send 10,000 requests for a 1 kB file using 12 concurrent threads in a local setting.

Like web servers, key-value stores and database systems are also essential to enterprise servers. Hence, we evaluated two key-value stores, Redis [39] and memcached [40], and a database program, SQLite [41]. We ran the key-value stores using their official benchmarks, redis-benchmark [103] and memaslap [104], respectively. In particular, for redis-benchmark, we ran its individual GET and SET benchmarks using default benchmark settings. Also, we configured memaslap to retrieve a 9:1 split of GET and SET key-value pairs for 1 minute using a concurrency level of 16. Finally, we ran the official speedtest for SQLite.

We also evaluated file compression (7-Zip [42]) and cryptographic programs (OpenSSL [43]) using their benchmarks written for the Phoronix Test Suite [105].

We ran each application ten times and averaged the results.

**Results.** Fig. 5 illustrates HARDLOG and AUDITD's performance overhead over native Linux execution. Fig. 6 shows how many log entries were produced by each program per second and how many of these entries were critical.

Considering all evaluated programs, HARDLOG's geometric mean performance overhead was 6.3%, while AUDITD's was 9.5%. The performance difference between HARDLOG and AUDITD is especially pronounced in memcached (10% versus 50%) and Redis (48% versus 64%). The reason is that these key-value store programs executed a significant number of `read` and `write` system calls, thus they produced many log entries per second (487k and 313k, respectively).

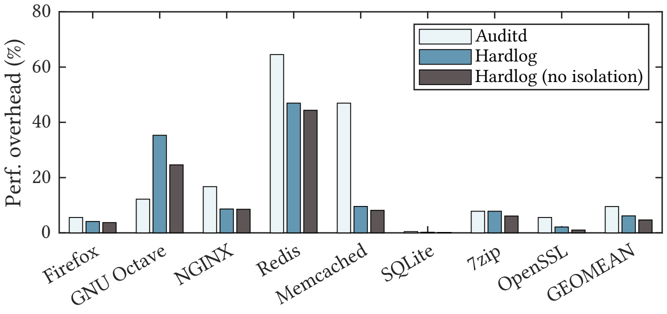


Fig. 5. Performance of HARDLOG and AUDITD on diverse real-world programs. The time taken to execute each program without auditing from left to right was (in seconds) 120.3, 39.9, 3.8, 60.0, 5.4, 64.9, 82.2, and 82.0.

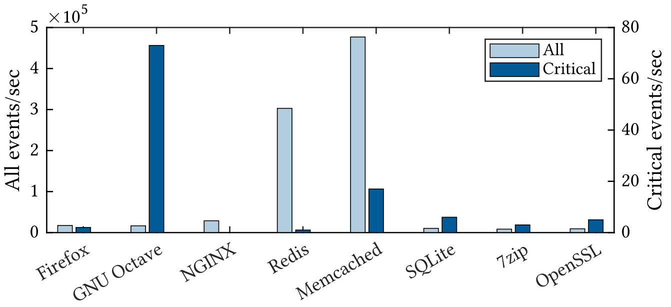


Fig. 6. Event statistics for real-world programs.

We also ran the real-world programs while disabling HARDLOG's micro-architectural component isolation, illustrated as HARDLOG (no isolation) in Fig. 5, which showed an overhead of 4.7% (a 1.6% improvement). This is expected since HARDLOG's micro-architectural component isolation protections are lightweight. Specifically, HARDLOG only preempts a CPU core (both SMT threads in it) when needed for log entry protection. Moreover, HARDLOG allocates a small (1/8) LLC partition to log sender. This has a low system impact on the remaining threads (for most programs) because they can still access a significant portion (7/8) of the LLC [79], [106].

Our evaluated real-world programs executed critical events significantly less frequently than non-critical events (Fig. 6). Except for GNU Octave, all programs executed less than 20 critical events per second, thus the impact of synchronously storing critical log entries was minimal. GNU Octave (the worst case) produced 73 critical events per second on average, mostly for binary execution (i.e., `execve`). This explains why GNU Octave was slower with HARDLOG than AUDITD which does not synchronously protect log entries.

**Takeaway.** Across diverse programs, HARDLOG performs comparably to AUDITD, a widely deployed audit system in enterprise machines [44]. These results indicate that there is not a significant performance hurdle towards HardLog's practical deployment on enterprise machines.

### E. Query Handling

This section describes HardLog's performance related to administrator's log retrieval queries.

**Settings.** We ran an assorted set of programs continuously to send 1 GiB worth of logs (the average size of logs generated daily on an enterprise machine [8]) to the audit device. Then, we sent a log retrieval query to the device from the host's audit service. Concurrently, to keep the device communication interface busy, we ran a stress testing program at the host machine that executes 100,000 `getpid` system calls each second while logging them using HardLog.

**Results.** The response was received at the host machine's audit service in 26 s. In particular, the device's query executor took 9 s to read logs from its storage and 5 s to compress them. Once compressed, the response size became ∼30 MB only because logs are ASCII strings that can be compressed significantly. These operations took long, based on the storage performance (∼1 GB/s read throughput), because query executor reads from storage in 512 B chunks (§VI-D).

When the response was ready, the host's query relay thread took 12 s to retrieve it through the USB 3.0 interface to the audit device. This is expected since host query relay's access to the USB interface is paused frequently by log sender to prioritize log protection. Without concurrent logging, it only took 8 s to retrieve this response. While our experiment did not include sending data to a remote system administrator, in practice, this time merely depends on the uplink speed.

## X. Discussion

**Frequent critical events.** Certain programs could execute *some* critical events frequently and introduce a non-negligible synchronous log protection overhead. Thus, the administrator can decide to asynchronously protect some frequent critical events for such programs. Recall that HardLog's bounded-asynchronous guarantees ensure that an overwhelming portion of an attack trace is still protected (§VIII-B).

**Other micro-architectural components.** Apart from the LLC, other components (e.g., cache directories [107] and CPU ring interconnects [108]) are shared between different CPU cores, but the hardware does not currently support isolating these components. If such isolation is supported in the future, HardLog can be extended to apply it. Note that contention on many of these remaining components likely introduces a minor delay because their contention does not evict cached contents [108]. Thus, even with such contention, HardLog's log protection delay is still likely bounded to a low number.

**Post-compromise log retrieval.** HardLog securely keeps logs even after a host compromise. However, if the compromised host stops relaying network communications, an administrator must recover the machine to retrieve logs either physically or through remote recovery mechanisms (e.g., CIDER [109], Baseboard Management Controller (BMC) [110], Intel vPRO [111], and AMD PRO [112]). Alternatively, the audit device could be extended with a network card to enable direct remote management from a system administrator.

**System slowdown through HostBuf.** Every audit system, including HardLog, must pause threads that execute any security-related event when the in-memory log buffer (e.g., HostBuf) is full to avoid losing log entries. Attackers might abuse this feature to slow down the system. One way to prevent such attacks is to establish rate-limited quotas on how many security-related events allowed per second by profiling benign program behavior [113]. We leave this to future work.

## XI. Other Related work

**Efficient logging.** Existing audit systems (e.g., Auditd) rely on user-level software components to store logs in local or remote storage. Thus, their performance bottleneck is the slow communication channel between the kernel and user-space (e.g., Netlink) [100]. To avoid this bottleneck, more efficient audit systems [100], [114], [115] employ shared memory between kernel and user-space components. Unlike them, HardLog enables its kernel component to directly send logs to the audit device without relying on user-space components.

Log storage overhead is also a problem for audit systems. To decrease storage overhead, some audit systems use either lossy [116]–[120] or lossless [121]–[125] log reduction techniques. If a powerful audit device is used, HardLog can be programmed to employ such extensive compression techniques to reduce device storage utilization.

**Secure storage and TEEs.** Researchers have proposed secure storage for TEEs and general-purpose secure storage based on TEEs. For example, several researchers have developed secure local storage for SGX with a custom FPGA [126] or SSD [127]. An audit system can potentially use these techniques as a secure location for system logs (i.e., first send log entries to a user-space SGX application and then to a secure storage device), but this will incur significant protection delay due to multiple privilege transitions and data copies. General-purpose secure storage based on Intel Trusted Execution Technology (TXT) and a Self-Encrypting Drive (SED) [128] does not require a user-space service. However, its overhead is still substantial because all CPU execution context must be suspended to securely store or retrieve data [129], [130]. Although Arm TrustZone-based solutions [131]–[134] are generally efficient, TrustZone is unavailable on most enterprise machines (which are overwhelmingly x86-based). In contrast, HardLog's audit device enables fast secure storage on any architecture.

## XII. Conclusion

HardLog employs a novel storage device to protect system logs in the case of a machine compromise. On critical events, HardLog synchronously protects all buffered log entries. On all other events, HardLog bounded-asynchronously protects log entries (i.e., within a small delay from their creation). Our evaluation across diverse real-world programs shows that HardLog ensures a protection delay of only 15 ms for non-critical entries with a small performance overhead of 6.3%.

# Acknowledgment

We want to thank the anonymous reviewers for their valuable comments and suggestions, which significantly improved the presentation and quality of this work.

# References

[1] S. T. King and P. M. Chen, "Backtracking Intrusions," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[2] S. Ma, X. Zhang, and D. Xu, "ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[3] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu, "MPI: Multiple Perspective Attack Investigation with Semantic Aware Execution Partitioning," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[4] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee, "RAIN: Refinable Attack Investigation with On-Demand Inter-Process Information Flow Tracking," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[5] Y. Ji, S. Lee, M. Fazzini, J. Allen, E. Downing, T. Kim, A. Orso, and W. Lee, "Enabling Refinable Cross-Host Attack Investigation with Efficient Data Flow Tagging and Tracking," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[6] 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 *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[7] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[8] W. U. Hassan, A. Bates, and D. Marino, "Tactical Provenance Analysis for Endpoint Detection and Response Systems," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[9] G. Hombrebueno, "Securing Remote Work: Protecting Endpoints the Right Way - Cisco Blogs," 2020, https://blogs.cisco.com/security/securing-remote-work-protecting-endpoints-the-right-way.

[10] Symantec, "ISTR 24: Symantec's Annual Threat Report Reveals More Ambitious and Destructive Attacks," https://symantec-enterprise-blogs.security.com/blogs/threat-intelligence/istr-24-cyber-security-threat-landscape.

[11] CIS Center for Internet Security, "The SolarWinds Cyber-Attack: What You Need to Know," 2021, https://www.cisecurity.org/solarwinds/.

[12] The White House, "Executive Order on Improving the Nation's Cybersecurity," 2021, https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/.

[13] Flexxon, "WORM," https://www.flexxon.com/worm/.

[14] R. Paccagnella, P. Datta, W. U. Hassan, A. Bates, C. Fletcher, A. Miller, and D. Tian, "CUSTOS: Practical Tamper-Evident Auditing of Operating Systems Using Trusted Execution," in *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2020.

[15] V. Karande, E. Bauman, Z. Lin, and L. Khan, "SGX-Log: Securing System Logs with SGX," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS)*, 2017.

[16] A. Sinha, L. Jia, P. England, and J. R. Lorch, "Continuous Tamper-Proof Logging Using TPM 2.0," in *International Conference on Trust and Trustworthy Computing (Trust)*. Springer, 2014, pp. 19–36.

[17] R. Paccagnella, K. Liao, D. Tian, and A. Bates, "Logging to the Danger Zone: Race Condition Attacks and Defenses on System Audit Frameworks," in *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2020.

[18] C. Cimpanu, "Hackers are increasingly destroying logs to hide attacks." [Online]. Available: https://www.zdnet.com/article/hackers-are-increasingly-destroying-logs-to-hide-attacks/

[19] "BlackEnergy APT Attacks in Ukraine," 2021. [Online]. Available: https://www.kaspersky.com/resource-center/threats/blackenergy

[20] MITRE ATT&CK, "Indicator Removal on Host: Clear Windows Event Logs, Sub-technique T1070.001 - Enterprise," https://attack.mitre.org/techniques/T1070/001/.

[21] ——, "Indicator Removal on Host: Clear Linux or Mac System Logs, Sub-technique T1070.002 - Enterprise," https://attack.mitre.org/techniques/T1070/002/.

[22] rsyslog, "Receiving Messages from a Remote System," https://www.rsyslog.com/receiving-messages-from-a-remote-system/.

[23] S. Marwedel, "Secure logging with syslog-ng: Forward integrity and confidentiality of system logs," Free and Open Source Software Developers' European Meeting (FOSDEM), 2020.

[24] E.-O. Blass and G. Noubir, "Secure Logging with Crash Tolerance," in *2017 IEEE Conference on Communications and Network Security (CNS)*, 2017.

[25] B. Schneier and J. Kelsey, "Cryptographic Support for Secure Logs on Untrusted Machines," in *Proceedings of the 7th USENIX Security Symposium (Security)*, San Antonio, TX, Jan. 1998.

[26] D. Ma and G. Tsudik, "A new approach to secure logging," *ACM Transactions on Storage (TOS)*, vol. 5, no. 1, pp. 1–21, 2009.

[27] S. A. Crosby and D. S. Wallach, "Efficient Data Structures for Tamper-Evident Logging," in *Proceedings of the 18th USENIX Security Symposium (Security)*, Montreal, Canada, Aug. 2009.

[28] A. A. Yavuz, P. Ning, and M. K. Reiter, "Efficient, Compromise Resilient and Append-Only Cryptographic Schemes for Secure Audit Logging," in *International Conference on Financial Cryptography and Data Security (FC)*. Springer, 2012, pp. 148–163.

[29] K. D. Bowers, C. Hart, A. Juels, and N. Triandopoulos, "PillarBox: Combating Next-Generation Malware with Fast Forward-Secure Logging," in *International Workshop on Recent Advances in Intrusion Detection (RAID)*. Springer, 2014, pp. 46–67.

[30] T. Pulls and R. Peeters, "Balloon: A Forward-Secure Append-Only Persistent Authenticated Data Structure," in *European Symposium on Research in Computer Security (ESORICS)*. Springer, 2015, pp. 622–641.

[31] M. Bernaschi, E. Gabrielli, and L. V. Mancini, "Enhancements to the Linux Kernel for Blocking Buffer Overflow Based Attacks," in *Proceedings of the 4th Annual Linux Showcase & Conference*, 2000.

[32] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal System Call Specialization for Attack Surface Reduction," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug. 2020.

[33] "Real-time systems," http://www.cse.unsw.edu.au/~cs9242/08/lectures/09-realtimex2.pdf.

[34] Pine64, "RockPro64 | Pine64," https://www.pine64.org/rockpro64/.

[35] L. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," in *Proceedings of the 1996 USENIX Annual Technical Conference (ATC)*, Jan. 1996.

[36] "Firefox - protect your life online with privacy-first products." [Online]. Available: https://www.mozilla.org/en-US/firefox/

[37] J. W. Eaton, "GNU Octave." [Online]. Available: https://www.gnu.org/software/octave/index

[38] NGINX Inc., "NGINX High Performance Load Balancer, Web Server, & Reverse Proxy," https://www.nginx.com.

[39] Redis Ltd., "Redis," https://redis.io/.

[40] Dormando, "memcached - a distributed memory object caching system," https://memcached.org/.

[41] SQLite Consortium, "SQLite home page." [Online]. Available: https://www.sqlite.org/index.html

[42] I. Pavlov, "7-Zip." [Online]. Available: https://www.7-zip.org/

[43] OpenSSL Software Foundation, "OpenSSL: Cryptography and SSL/TLS Toolkit," https://www.openssl.org/, 2017.

[44] SUSE, "Understanding Linux Audit," https://documentation.suse.com/sles/12-SP4/html/SLES-all/cha-audit-comp.html.

[45] MITRE ATT&CK, "Remote Data Storage, Mitigation M1029 - Enterprise," https://attack.mitre.org/mitigations/M1029/.

[46] A. Vahldiek-Oberwagner, E. Elnikety, A. Mehta, D. Garg, P. Druschel, R. Rodrigues, J. Gehrke, and A. Post, "Guardat: Enforcing Data Policies at the Storage Layer," in *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*, Bordeaux, France, Apr. 2015.

[47] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer, "PESOS: Policy Enhanced Secure Object Store," in

*Proceedings of the 13th European Conference on Computer Systems (EuroSys)*, Porto, Portugal, Apr. 2018.

[48] B. Gregg, *Systems Performance: Enterprise and the Cloud*. Pearson Education, 2014.

[49] Flexxon, "WORM (Write-Once-Read-Many) USB Device," https://www.flexxon.com/worm-usb-data-storage-integrity/.

[50] ——, "Read-only Mode Memory Card," https://www.flexxon.com/read-only-mode-memory-card/.

[51] rsyslog, "KSI Signature Provider," https://www.rsyslog.com/doc/master/configuration/modules/sigprov_ksi12.html.

[52] R. N. Williams, "Common reasons for computer failure in business industry," https://www.streetdirectory.com/travel_guide/116177/computers/common_reasons_for_computer_failure_in_business_industry.html.

[53] Pew Research Center, "When technology fails," https://www.pewresearch.org/internet/2008/11/16/when-technology-fails/.

[54] Metasploit, "Metasploit | Penetration Testing Software, Pen Testing Security," https://www.metasploit.com.

[55] Offensive Security, "Exploit Database - Exploits for Penetration Testers, Researchers, and Ethical Hackers," https://www.exploit-db.com.

[56] J. Salwan, "Shellcodes database for study cases," http://shell-storm.org/shellcode/.

[57] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation," in *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, Mar. 2015.

[58] Microsoft Docs, "Overview of single root i/o virtualization (SR-IOV)," https://docs.microsoft.com/en-us/windows-hardware/drivers/network/overview-of-single-root-i-o-virtualization--sr-iov-.

[59] ——, "Virtualization-based Security (VBS)," https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs.

[60] Trusted Computing Group, "TPM Main Specification Version 1.2 Rev. 116," 2011, https://trustedcomputinggroup.org/resource/tpm-main-specification/.

[61] K. Vaid, "Microsoft Creates Industry Standards for Datacenter Hardware Storage and Security," 2018, https://azure.microsoft.com/en-us/blog/microsoft-creates-industry-standards-for-datacenter-hardware-storage-and-security/.

[62] U. Savagaonkar, N. Porter, N. Taha, B. Serebrin, and N. Mueller, "Titan in Depth: Security in Plaintext," 2017, https://cloud.google.com/blog/products/gcp/titan-in-depth-security-in-plaintext.

[63] Apple, "Apple T2 Security Chip: Security Overview," 2018, https://www.apple.com/mac/docs/Apple_T2_Security_Chip_Overview.pdf.

[64] Mouser Electronics, "SCCT47B356SRB AVX," https://www.mouser.com/ProductDetail/AVX/SCCT47B356SRB?qs=vmHwEFxEFR8mN0UvpotaNw==.

[65] Red Hat Customer Portal, "Red Hat Enterprise Linux for Real Time 7: 4.5. Setting Real-time Scheduler Priorities," https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/tuning_guide/setting_realtime_scheduler_priorities.

[66] Microsoft Docs, "Increase Scheduling Priority (Windows 10) - Windows Security," https://docs.microsoft.com/en-us/windows/security/threat-protection/security-policy-settings/increase-scheduling-priority.

[67] Y. Lee, C. Min, and B. Lee, "ExpRace: Exploiting Kernel Races through Raising Interrupts," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Aug. 2021.

[68] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual," *Volume 3A: System Programming Guide*, 2016.

[69] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune, "Building Verifiable Trusted Path on Commodity x86 Computers," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[70] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel, 3rd Edition*. O'Reilly, 2006.

[71] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun, "Using Hardware Features for Increased Debugging Transparency," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[72] Red Hat Customer Portal, "Red Hat Enterprise Linux for Real Time 7: 3.3. Non-Maskable Interrupts," https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/reference_guide/non-maskable_interrupts.

[73] B. W. Lampson and D. D. Redell, "Experience with Processes and Monitors in Mesa," *Communications of the ACM*, vol. 23, no. 2, pp. 105–117, 1980.

[74] S. J. Daigle and J. K. Strosnider, "Disk Scheduling for Multimedia Data Streams," in *High-Speed Networking and Multimedia Computing*, vol. 2188. International Society for Optics and Photonics, 1994, pp. 212–223.

[75] D. Grunwald and S. Ghiasi, "Microarchitectural Denial of Service: Insuring Microarchitectural Fairness," in *Procedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2002.

[76] J. Hasan, A. Jalote, T. Vijaykumar, and C. Brodley, "Heat Stroke: Power-Density-Based Denial of Service in SMT," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA)*, 2005.

[77] T. Allan, B. B. Brumley, K. Falkner, J. van de Pol, and Y. Yarom, "Amplifying Side Channels through Performance Degradation," in *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC)*, 2016.

[78] A. C. Aldaya and B. B. Brumley, "HyperDegrade: From GHz to MHz Effective CPU Frequencies," in *Proceedings of the 31st USENIX Security Symposium (Security)*, Aug. 2022.

[79] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[80] Intel, "Code sample: Intel® software guard extensions remote attestation end-to-end example," https://www.intel.com/content/www/us/en/developer/articles/code-sample/software-guard-extensions-remote-attestation-end-to-end-example.html.

[81] The Gramine Project, "RA-TLS Minimal Example," https://github.com/gramineproject/graphene/tree/master/Examples/ra-tls-mbedtls.

[82] D. Wheeler, "Sloccount," https://dwheeler.com/sloccount/.

[83] N. Brown, "A block layer introduction part 1: The bio layer," https://lwn.net/Articles/736534/, 2017.

[84] The kernel development community, "Mass Storage Gadget (MSG)," https://www.kernel.org/doc/html/latest/usb/mass-storage.html.

[85] S. Soller, "Measurements of System Call Performance and Overhead – Arakanis Development," http://arkanis.de/weblog/2017-01-05-measurements-of-system-call-performance-and-overhead.

[86] A. Popov, "Four Bytes of Power: Exploiting CVE-2021-26708 in the Linux kernel," https://a13xp0p0v.github.io/2021/02/09/CVE-2021-26708.html.

[87] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an OS kernel," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.

[88] E. Lubbers and M. Platzner, "ReconOS: An RTOS supporting hard- and software threads," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, 2007.

[89] S. Angel, R. S. Wahby, M. Howald, J. B. Leners, M. Spilo, Z. Sun, A. J. Blumberg, and M. Walfish, "Defending Against Malicious Peripherals with Cinch," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[90] D. J. Tian, N. Scaife, A. Bates, K. Butler, and P. Traynor, "Making USB Great Again with USBFILTER," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[91] D. J. Tian, G. Hernandez, J. I. Choi, V. Frost, P. C. Johnson, and K. R. Butler, "LBM: A Security Framework for Peripherals within the Linux Kernel," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[92] K. Zhong, Z. Jiang, K. Ma, and S. Angel, "A file system for safely interacting with untrusted USB flash drives," in *Proceedings of the 12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2020.

[93] T. Ramananandro, A. Delignat-Lavaud, C. Fournet, N. Swamy, T. Chajed, N. Kobeissi, and J. Protzenko, "EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats," in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[94] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. M.

Wintersteiger, and S. Zanella-Béguelin, "EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[95] A. Gehani and D. Tariq, "SPADE: Support for Provenance Auditing in Distributed Environments," in *Proceedings of The Annual ACM/IFIP Middleware Conference (Middleware)*, 2012.

[96] Red Hat Customer Portal, "Red Hat Enterprise Linux 7: 7.4. Starting the Audit Service," https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/security_guide/sec-starting_the_audit_service.

[97] "Toshiba BG3 SSD," https://pcpartsoutletstore.com/toshiba-kbg30zmv256g-256gb-ssd-m-2-2280-nvme-pcie-solid-state-drive-l22028-002.html.

[98] C. Staelin and L. McVoy, "Lat_syscall (man page)," http://lmbench.sourceforge.net/man/lat_syscall.8.html.

[99] "netlink(7) - linux manual page," https://man7.org/linux/man-pages/man7/netlink.7.html.

[100] S. Ma, J. Zhai, Y. Kwon, K. H. Lee, X. Zhang, G. Ciocarlie, A. Gehani, V. Yegneswaran, D. Xu, and S. Jha, "Kernel-Supported Cost-Effective Audit Logging for Causality Tracking," in *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, Jul. 2018.

[101] "Speedometer 2.0," https://browserbench.org/Speedometer2.0/.

[102] The Apache Software Foundation, "ab - Apache HTTP server benchmark tool," https://httpd.apache.org/docs/2.4/programs/ab.html.

[103] Redis Ltd., "Redis benchmark," https://redis.io/docs/reference/optimization/benchmarks/.

[104] M. Zhuang and B. Aker, "memaslap - load testing and benchmarking a server," http://docs.libmemcached.org/bin/memaslap.html.

[105] Phoronix Media, "Phoronix test suite." [Online]. Available: https://www.phoronix-test-suite.com/

[106] X. Dong, Z. Shen, J. Criswell, A. L. Cox, and S. Dwarkadas, "Shielding software from privileged side-channel attacks," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug 2018.

[107] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[108] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Aug. 2021.

[109] M. Xu, M. Huber, Z. Sun, P. England, M. Peinado, S. Lee, A. Marochko, D. Matoon, R. Spiger, and S. Thom, "Dominance as a New Trusted Computing Primitive for the Internet of Things," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[110] J. Frazelle, "Opening Up the Baseboard Management Controller," *Communications of the ACM*, vol. 63, no. 2, pp. 38–40, 2020.

[111] O. Levy, A. Kumar, and P. Goel, "Advanced Security Features of Intel vPro Technology," *Intel Technology Journal*, vol. 12, no. 4, 2008.

[112] AMD, "AMD PRO Technologies," https://www.amd.com/en/technologies/pro-technologies.

[113] M. Abubakar, A. Ahmad, P. Fonseca, and D. Xu, "Shard: Fine-grained kernel specialization with context-aware hardening," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Aug. 2021.

[114] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, "Practical Whole-System Provenance Capture," in *Proceedings of the 2017 Symposium on Cloud Computing (SoCC)*, 2017, pp. 405–418.

[115] T. Pasquier, X. Han, T. Moyer, A. Bates, O. Hermant, D. Eyers, J. Bacon, and M. Seltzer, "Runtime Analysis of Whole-System Provenance," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[116] K. H. Lee, X. Zhang, and D. Xu, "LogGC: Garbage Collecting Audit Log," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Oct. 2013.

[117] Z. Xu, Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang, "High Fidelity Data Reduction for Big Data Security Dependency Analyses," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[118] M. N. Hossain, J. Wang, R. Sekar, and S. D. Stoller, "Dependence-Preserving Data Compaction for Scalable Forensic Analysis," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[119] Y. Tang, D. Li, Z. Li, M. Zhang, K. Jee, X. Xiao, Z. Wu, J. Rhee, F. Xu, and Q. Li, "NodeMerge: Template Based Efficient Data Reduction For Big-Data Causality Analysis," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[120] N. Michael, J. Mink, J. Liu, S. Gaur, W. U. Hassan, and A. Bates, "On the Forensic Validity of Approximated Audit Logs," in *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC)*, 2020, pp. 189–202.

[121] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen, "Eidetic Systems," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, CO, Oct. 2014.

[122] A. Quinn, D. Devecsery, P. M. Chen, and J. Flinn, "JetStream: Cluster-Scale Parallelization of Information Flow Queries," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.

[123] K. Rodrigues, Y. Luo, and D. Yuan, "CLP: Efficient and Scalable Search on Compressed Text Logs," in *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Jul. 2021.

[124] P. Fei, Z. Li, Z. Wang, X. Yu, D. Li, and K. Jee, "SEAL: Storage-efficient Causality Analysis on Enterprise Logs with Query-friendly Compression," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Aug. 2021.

[125] H. Ding, S. Yan, J. Zhai, and S. Ma, "ELISE: A Storage Efficient Logging System Powered by Redundancy Reduction and Representation Learning," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Aug. 2021.

[126] H. Oh, A. Ahmad, S. Park, B. Lee, and Y. Paek, "TRUSTORE: Side-Channel Resistant Storage for SGX using Intel Hybrid CPU-FPGA," in *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2020.

[127] J. Ahn, J. Lee, Y. Ko, D. Min, J. Park, S. Park, and Y. Kim, "DiskShield: A Data Tamper-Resistant Storage for Intel SGX," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS)*, 2020, pp. 799–812.

[128] L. Zhao and M. Mannan, "TEE-aided Write Protection Against Privileged Data Tampering," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[129] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An Execution Infrastructure for TCB Minimization," in *Proceedings of the 3rd European Conference on Computer Systems (EuroSys)*, Glasgow, Scotland, Mar. 2008.

[130] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, CO, Oct. 2014.

[131] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using Arm TrustZone to Build a Trusted Language Runtime for Mobile Applications," in *Proceedings of the 19th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Salt Lake City, UT, Mar. 2014.

[132] D. Hein, J. Winter, and A. Fitzek, "Secure Block Device – Secure, Flexible, and Efficient Data Storage for Arm TrustZone Systems," in *Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA*, 2015.

[133] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "TrustShadow: Secure Execution of Unmodified Applications with Arm TrustZone," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2017.

[134] L. Guo, Y. Zhang, and F. X. Lin, "Let the Cloud Watch Over Your IoT File Systems," *arXiv preprint arXiv:1902.06327*, 2019.