Rethinking System Audit Architectures for High Event Coverage and Synchronous Log Availability

Varun Gandhi, *Harvard University;* Sarbartha Banerjee, *University of Texas at Austin;* Aniket Agrawal and Adil Ahmad, *Arizona State University;* Sangho Lee and Marcus Peinado, *Microsoft Research*

https://www.usenix.org/conference/usenixsecurity23/presentation/gandhi

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

Rethinking System Audit Architectures for High Event Coverage and Synchronous Log Availability

Varun Gandhi* *Harvard University*
Sarbartha Banerjee* *University of Texas at Austin*
Aniket Agrawal *Arizona State University*
Adil Ahmad *Arizona State University*
Sangho Lee *Microsoft Research*
Marcus Peinado *Microsoft Research*

Abstract

Once an attacker compromises the operating system, the integrity and availability of unprotected system audit logs still kept on the computer becomes uncertain. In this paper, we ask the question: *can recently proposed audit systems aimed at tackling such an attacker provide enough information for forensic analysis?* Our findings suggest that the answer is no, because the inefficient logging pipelines of existing audit systems prohibit generating log entries for a vast majority of attack events and protecting logs as soon as they are created (i.e., *synchronously*). This leads to a low attack event coverage within generated logs, while allowing attackers to tamper with unprotected logs after a compromise. To counter these limitations, we present OMNILOG, a system audit architecture that composes an end-to-end efficient logging pipeline where logs are rapidly generated and protected using a set of platform-agnostic security abstractions. This allows OMNILOG to enable high attack event coverage and synchronous log availability, while even outperforming the state-of-the-art audit systems that achieve neither property.

1 Introduction

System auditing [66, 85] is a widely-used defense mechanism on enterprise machines. Audit systems record important events (e.g., system calls from unknown binaries) and store them in local or remote storage. If a machine is found to be compromised, logs are used by forensic analysts to determine the attack vector and origin of the compromise.

However, audit systems are not robust against an adversary that attacks the operating system [18, 43, 48, 51, 72, 73, 87]. In particular, an adversary that obtains kernel privilege can tamper with the audit system to prevent it from generating logs for all their post-compromise behavior and tamper with all logs kept in the machine to hide most of their pre-compromise behavior. This is highly-concerning since kernel vulnerabilities continue to be an important threat to computer security with hundreds of CVEs for the Linux and Windows kernels in recent years [27, 28]. Such machines are a preferred target for sophisticated malware such as Advanced Persistent Threats (APTs) [19].

In recent years, significant research [8, 34, 44, 72, 73, 83] has been undertaken to design audit systems that at least protect logs generated *before* compromise so that forensic analysts are able to rely on them to detect or forensically analyze kernel attacks. Our case-study, however, shows that log entries available to analysts are often inadequate to analyze the attacks due to two fundamental limitations of existing audit systems: low event coverage and asynchronous log availability.

First, due to inefficient log generation and storage, existing systems are configured to record a small set of infrequent events (e.g., file system or network access) [30, 61, 68, 70, 75, 90] to avoid prohibitive slowdowns (§3.1). This restricted logging is insufficient to capture kernel exploit behaviors. Our analysis of 164 Proof-of-Concept (PoC) kernel exploits with CVEs finds a considerable number of exploits which leave *no trace* under widely used audit policies. These PoCs had not even been designed to evade the audit system. Sophisticated attacks should be able to develop stealthier exploits than them.

Second, since existing systems rely on local or remote storage for protection, they protect their generated log entries after a noticeable delay (i.e., *asynchronously*) to avoid significant slowdown (§3.2). Even the state-of-the-art, HardLog [8], requires 15 ms to protect logs in the worst-case, whereas even a naive attack took only ∼5 ms to tamper with logs on our machine.

This paper rethinks audit system architectures and presents OMNILOG, an audit system design that is optimized to efficiently generate log entries for frequent events (i.e., all system calls) and rapidly protect these entries at a location inaccessible to the operating system before allowing the events to happen. This allows OMNILOG to satisfy the two key properties—*high event coverage* and *synchronous log availability*—while

*Both authors contributed equally to this work. Part of this work was done while Varun Gandhi was an intern at Microsoft Research.

incurring a geometric mean performance slowdown of less than 4% across diverse real-world evaluation experiments.

The OMNILOG architecture (§4.1) divides the duties of an audit system as follows. First, critical tasks like log protection and persistence are delegated to trusted components running with higher CPU-enforced privileges than the operating system, namely OMNIMONITOR and OMNIKERNEL. Hence, they can quickly protect log entries *in-memory* and ensure eventual log persistence. Second, semi-critical tasks like log retrieval are securely but collaboratively enabled by trusted components and the operating system. Third, the non-critical task of log generation is left to the operating system, which is expected to handle it correctly until compromise.

There are two non-trivial hurdles towards the design of OMNILOG. First, given the wide diversity of CPU platforms, it is critical that our trusted components rely on security abstractions that are easily realizable across platforms. Second, even in-memory logging pipelines can become inefficient especially for a large volume of events.

We define security abstractions required by OMNILOG's trusted components and show how they can be realized using CPU features on two major platforms (i.e., Arm and x86) (§5.1). These abstractions include the ability to enable secure runtime environments for trusted component execution and interpose on power events to persist buffered log entries before they are discarded on system reset. Each of our defined abstractions can be enabled on major CPU platforms using already-available features like the Arm TrustZone [37] and x86 Extended Page Tables (EPT) [38].

We also compose an end-to-end efficient logging pipeline to generate and protect log entries for a high volume of events (§5.2). Inside the operating system, the logging pipeline rapidly generates compact log entries to effectively use the protected memory buffer space and reduce overall storage costs. Then, it leverages per-core data structures to maintain overall system concurrency while synchronously protecting entries (in OMNIMONITOR's memory). Finally, using OMNIKERNEL, the pipeline asynchronously but assuredly persists buffered log entries to protected storage inaccessible to the operating system.

To demonstrate the wide compatibility of OMNILOG, we prototyped it for both Arm and x86-based computers (§6). We perform a detailed security analysis of OMNILOG to show that it enables synchronous log availability (§7). We also evaluate both prototypes on micro-benchmarks and several popular real-world applications: NGINX, Memcached, Redis, Chromium, OpenSSL, 7-Zip, SQLite, and GNU Octave (§8). OMNILOG's geometric mean overhead across the real-world applications is only 3.2% (Arm) and 3.6% (x86). This overhead is 9.1× and 8.0× lower than that of Linux's deployed Auditd [85] on these platforms, respectively, and even outperforms state-of-the-art audit systems [8, 34, 73] that only log a small set (43/341) of system calls. Consequently, OMNILOG is readily-deployable today.

2 Background

2.1 Audit System

System auditing is the process by which the operating system maintains system logs of *security-related events* on a machine. These logs are maintained as a sequence of entries where each entry corresponds to an event (e.g., system call). Audit logs are implemented to support the detection and forensic analysis of anomalies and suspicious activities [75]. Famous examples of audit systems include the Linux Audit system [85] and Microsoft's Event Tracing for Windows (ETW) [66].

At a high-level, an audit system enables four main tasks: log generation, protection, persistence, and retrieval. The log generation task is handled by a kernel component which generates log entries according to a user-supplied audit policy. The remaining tasks (protection, persistence, and retrieval) are delegated to a root-level user-space daemon which receives generated logs, stores them to root-owned file(s), and communicates with remote administrators for log retrieval. Both audit system components communicate with each other through a user-to-kernel interface (e.g., Netlink, shared memory).

2.2 Threat Model

Our threat model is similar to the classical model used in recent studies [8, 34, 73]. In particular, we consider a remote attacker who aims to take control of a corporate machine by compromising its operating system. This machine is running an audit system inside the operating system and our attacker is aware of it. Hence, the attacker wants to hide their activities from the audit system or impair its execution [90].

The attacker must compromise the operating system to tamper with the audit system or its produced logs. Prior to compromising the operating system, we assume that the attacker completes attack preparation steps (i.e., initial reconnaissance, initial compromise, and foothold establishment [62]) using several stealthy techniques [19, 76]. Then, the attacker exploits a kernel vulnerability to compromise the operating system with a sequence of system calls that complete at time t_c.

Starting at t_c, the attacker controls all aspects of the audit system and corrupts it completely to hide their activities. Specifically, they prevent the audit system from generating log entries for their subsequent malicious activities [18, 43, 87]. Hence, all log entries generated after t_c are worthless. Moreover, they tamper with the log entries generated before t_c and still on the computer to hide their attack exploit traces.

2.3 Assumptions

We make the following general assumptions about our hardware, software, and configuration. First, we assume that a corporate machine's built-in hardware components (e.g., CPU, motherboard) are trustworthy. Second, we assume that the

	MITRE [90]	NISPOM [70]	DSS [75]	OSPP [30]	STIG [68]	MSD [61]	Academic [8, 72, 73]
Average	14.7%	12.4%	10.0%	9.8%	9.8%	10.2%	15.4%
Variance	17.1%	15.3%	14.8%	14.2%	13.9%	14.3%	21.6%
0% coverage	60/164	77/164	122/164	77/164	77/164	80/164	11/164

Table 1: Attack event coverage for industry-standard and academic audit rulesets using our attack exploit case-study.

machine is provisioned and managed by trustworthy IT administrators. This management includes machine recovery and forensic data collection through physical access or remote mechanisms [29, 36, 54, 95] after a machine compromise. Third, we assume that the employee who is using the machine is not allowed to arbitrarily reprovision the machine and change its security configuration. Fourth, we assume that the machine's operating system (including the audit subsystem) behaves as designed until attackers exploit and take control of it. Lastly, we assume that there is no significant bug in our codebase and in the system software, except for the operating system.

3 Case-Study on Today's System Auditing

Ideally, an audit system designed for kernel-level exploits leveraged by our attacker should ensure two properties:

P1: High event coverage. An audit system provides *high coverage* if it generates log entries for a large majority of attack-related events on a computer. Given our threat model and that of similar auditing studies [8, 34, 72, 73], we consider these events to be system calls executed by a user process in the course of exploiting a kernel vulnerability. A high event coverage gives defenders (e.g., forensics analysts) a good chance to perform analysis on these attacks.

P2: Synchronous log availability. An audit system provides *synchronous availability* if it guarantees that every generated log entry is stored beyond the reach of our attacker before its corresponding event occurs. Kernel exploits can complete within a very short interval, especially once attack preparation is complete. Synchronous availability ensures that every generated log entry will be eventually available to a defender for analysis, no matter the kernel exploit speed.

Through case-studies on attack event coverage (§3.1) and log availability (§3.2), this section analyzes whether the configuration and design of existing audit systems [8, 34, 72, 73, 85] is suitable to achieve an ideal scenario for auditing.

3.1 Event Coverage

We first determine whether widely-used audit rulesets can log all attack-related events on known real-world kernel exploits (hence, achieve **P1**). For this study, we collected 164 PoC exploits with publicly disclosed CVEs from several open-source GitHub repositories, which include diverse kernel exploits (with payloads) from the last 15 years affecting various subsystems (e.g., memory, scheduling) of Linux and exploits that leverage popular kernel vulnerabilities like dirty COW [1]. While our collected exploit database is statistically small given that hundreds of kernel vulnerabilities are discovered each year, it serves as a good starting point for discussion in terms of auditing under kernel exploits.

To understand how kernel exploits are logged today, we studied the logging specifications of several influential information security standards, a real-world system, and academic papers. In particular, we studied five standards, including the MITRE ATT&CK framework [90] and the United States Department of Defense National Industrial Security Program Operation Manual (NISPOM) [70], and collected their audit rulesets [58]. We also analyzed the audit ruleset of Microsoft Defender ATP for Linux [61] and prior academic papers that attempt to secure logs against kernel exploits [8, 72, 73].

Using the collected attack exploits and rulesets, we performed an automated offline analysis by extracting all the system calls contained in the kernel exploits and referencing them with the system calls logged by the audit rulesets. This allowed us to determine the attack event coverage on each exploit for these rulesets. We will release links to the GitHub repositories for the PoC exploits and rulesets, and our analysis scripts for future research. We believe this release has no ethical issue because these exploits are with publicly disclosed CVEs and governed by GitHub policies [33].

Our finding (Table 1) was that existing rulesets do not log an overwhelming majority of attack-related events—the highest average attack event coverage on our database was only 15.4%. In fact, five out of seven rulesets did not log even a single event for more than 40% of the kernel exploits. This was a highly surprising result given that these exploits are PoCs, hence, not even designed to evade auditing.

While it might seem trivial to address the problem of low event coverage using a more comprehensive audit ruleset, the diversity of exploited system calls (146 unique system calls) suggest that audit systems must log all system calls to maximize coverage. However, the runtime cost of typical audit systems likely prohibits such an expansive configuration.

To estimate the runtime cost of a comprehensive audit policy, we ran several real-world workloads with Linux Auditd [85], the defacto Linux audit system, while configuring it to log all system calls on our x86 computer (§6.2). We observed that workloads like memcached show up to 261% overhead (Figure 1). Since some systems show that Auditd's performance can be improved by optimizing its kernel to user log transmission pipeline [8, 59], we also ran Auditd entirely *in-memory* where each entry was discarded right after generation instead of sending to a user daemon (shown as Auditd-IM). This is a setting that assumes logs can be protected instantaneously after generation, which is not practical (§3.2), but serves as a good basis for comparison. To our

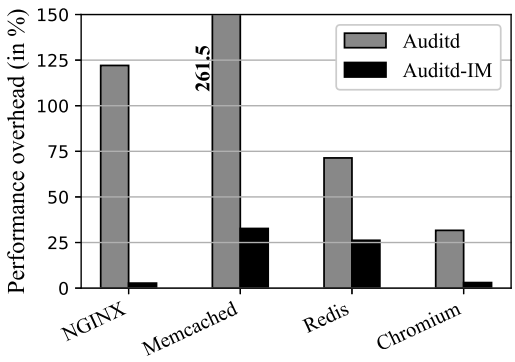


Figure 1: Performance overhead incurred by Auditd and Auditd-IM using a few real-world programs on our x86 computer. The platform and experiment settings are noted in §6.2 and §8.

surprise, Auditd-IM's reported throughput for memcached (using the comprehensive ruleset) was still 33% lower than native, a significant cost on enterprise computers.

A closer examination of Auditd's logging pipeline revealed several inefficiencies. In particular, Auditd produces string-formatted log entries which are very large (200–1024 B) and incur a significant time (∼12k cycles) to generate. These log entries are enqueued in a single buffer shared across all cores through serialization operations. Since the log entries are large and sent through a local network interface to Auditd's user daemon, the bandwidth that can be supported through this interface also becomes a bottleneck. For instance, we noticed a ping latency of 0.031 ms on our x86 computer's loopback network interface. This means the computer can support 32k packets of 64 B per second. In contrast, Memcached invoked more than 66k system calls per second per core, with an average log entry size of more than 300 B. Finally, the 60-second benchmark also produced 2.2 GiB of logs, which also creates a storage problem.

> **Findings:** *Comprehensive policies for logging all system calls are needed to ensure a high event coverage (**P1**), but inefficiencies in the logging pipeline of state-of-the-art audit systems prohibit such configurations.*

3.2 Log Availability

Next, we determine whether the design of existing audit systems is sufficient to synchronously protect every generated log (hence, achieve **P2**). Remote storage has been the *de facto* log protection mechanism for decades. However, given high network latency, remote storage mechanisms are too slow to ensure synchronous log availability; hence, logs are protected asynchronously after long periods. Many systems [34, 72, 73] attempt to secure such asynchronously-protected log entries using cryptographic tamper-evident hashes, which inherently cannot provide availability and suffer from false alerts [8].

HardLog [8] synchronously stores log entries to protected local storage but can do so only *infrequently* for a small set of 11 system calls without incurring excessive overhead. For all remaining system calls, HardLog asynchronously stores log entries within a small bounded delay. While this bounded delay is a noteworthy step towards log availability, HardLog's smallest achieved bound of 15 ms is still a long time on modern computers for attackers to tamper with logs and harm availability. For example, on our x86 machine (§6.2), it takes only ∼5 ms to load and execute an attack kernel module (4.1 kB) which overwrites a known buffer address. Note that this is a naive attack. In principle, attackers can complete this address overwrite entirely in-memory at a system call.

We surmise that audit systems must synchronously protect logs in main memory for efficiency. While modern storage and networking hardware can reach very high bandwidths, storing or sending data invariably incurs a fixed setup cost [8] which is too high for synchronous use-cases. However, since memory is finite and volatile, the logs must be eventually stored (in a guaranteed manner) to persistent drives.

> **Findings:** *In-memory log protection with eventual but assured persistence is needed to ensure efficient synchronous log availability (**P2**), but existing audit systems rely on slow network or storage protection.*

4 OmniLog Overview

OmniLog is an audit system architecture that efficiently achieves high event coverage and synchronous log protection (§3). This section provides an overview of OmniLog's architecture and expands on two design challenges.

4.1 Audit System Architecture

Following from an audit system's four main tasks (§2.1), OmniLog delegates log protection and persistence to trusted components, OmniMonitor and OmniKernel, respectively, while leaving log generation to the untrusted operating system's logging subsystem. Log retrieval is handled securely but collaboratively by the trusted and untrusted components. Figure 2 gives an overview of the main architectural components. We explain each component in the next paragraphs.

OmniMonitor. It is a trusted *security monitor* executing on the enterprise machine as the main security enforcement component of OmniLog. It executes at a CPU-enforced layer more privileged than the operating system like virtualization-based monitors (e.g., Microsoft's VBS [65]) and TrustZone-based monitors (e.g., Samsung's TZ-RKP [12]). This monitor isolates a region of the system's memory to rapidly protect log entries generated by the logging subsystem, while facilitating their persistence and retrieval through OmniKernel.

OmniKernel. It is a trusted but separate environment spun by OmniMonitor, with two main components: the *log keeper* and *log manager*. The log keeper is responsible

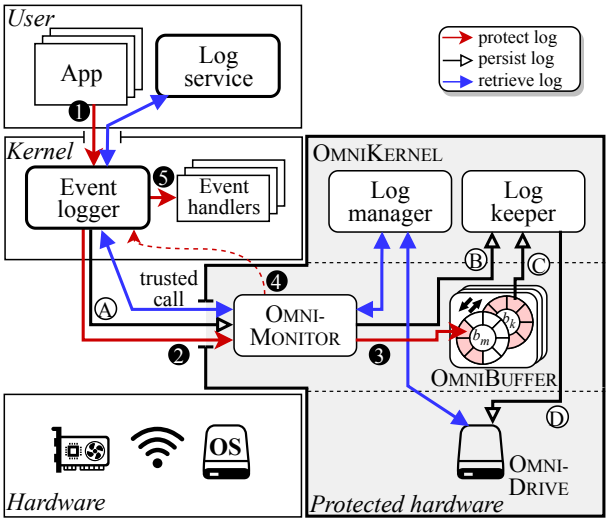


Figure 2: Overview of OmniLog.

for reliably storing protected log entries in persistent storage, while the log manager addresses queries by remote administrators (e.g., for log retrieval). Both components can access a protected persistent storage device (OmniDrive).

Although both OmniMonitor and OmniKernel are trusted, we separate them primarily from a Trusted Computing Base (TCB) and implementation standpoint. In particular, a separate trusted environment limits the attack surface since all communication between the untrusted logging subsystem and OmniKernel goes through OmniMonitor. Moreover, frameworks with isolated monitors and kernels can be widely-implemented in today's computers (§6).

Unlike storage access, OmniKernel does not have direct network access and all network communication from the system administrator is received through the operating system but through an authenticated channel. Like prior audit work [8] and several trusted execution environments without direct network access [40], OmniLog uses the operating system as an untrusted network transport mechanism, while relying on an end-to-end TLS-based secure communication channel between OmniKernel and the remote administrator.

Finally, the log manager supports all typical audit system commands from the remote administrator, including log retrieval, filtration, and other processing commands [8].

Logging subsystem. This subsystem is shipped as part of the operating system. Like typical audit systems, it contains a kernel-level *event logger* and user-level *log service*.

The event logger is responsible for (a) intercepting security-related events and generating log entries and (b) initiating blocking requests to trusted components of OmniLog when log entries must be protected and persisted. For instance, the logger intercepts all system call events and generates log entries corresponding to them. Then, it submits a request to OmniMonitor for log protection. This request is a blocking call, where the logger stops executing until the request completes. Then, it resumes and allows the event to happen.

The log service receives signed queries (e.g., for log retrieval) from a remote system administrator and sends them to the trusted kernel. This service also relays the (encrypted and signed) responses for these queries to the admin. The remote administrator would frequently retrieve logs to detect attacks, conduct forensics analysis, and vacate the log storage.

4.2 Challenges

This section presents two significant challenges towards the design of the OmniLog architecture.

C1: Formulating platform-agnostic security abstractions. Given the wide diversity of CPU platforms (e.g., Arm, x86), it is critical to show that OmniMonitor's CPU-enabled security abstractions can be realized across major existing platforms to clear hurdles towards wide adoption.

C2: Composing an end-to-end efficient logging pipeline. Given the inefficiency of existing logging pipelines, even when configured for unprotected in-memory logging, on comprehensive auditing policies (§3.1), it is critical to compose a new pipeline that can rapidly generate and protect logs.

5 OmniLog Design

5.1 Platform-Agnostic Security Abstractions

This section outlines OmniMonitor's required security abstractions and discusses how they can be fulfilled using popular CPU-enforced security monitor designs [37, 65, 80] on major CPU platforms (i.e., x86 and Arm). In particular, we consider the hypervisor enforcement layer (e.g., Intel VMX) and architectural system management mode (i.e., Arm's EL3).

S1: Secure key provision and load. OmniMonitor must be securely provisioned on the computer during initial setup and executed before the operating system starts at each subsequent computer reset to ensure it can uphold security properties (e.g., protect OmniDrive).

During initial setup, OmniMonitor is installed on a machine by a trusted administrator. At this time, they also provision a pair of public and private cryptographic keys to enable an authenticated secure communication channel with the monitor at a later time (e.g., for log retrieval).

On each subsequent boot, OmniMonitor must execute before the operating system so that it can implement protections. This execute-before property can be achieved using a *verification* or *protection* approach. In particular, many systems leverage secure signature verification [9] during boot to halt execution on tampering of the boot configuration like UEFI Secure Boot [94]. Other systems [52, 95] write-protect boot configurations—kept in block storage or non-volatile RAM (NVRAM)—at runtime, and load trusted binaries from

isolated storage drives. OMNILOG can implement this property using the write-protection and device isolation primitives (**S2** and **S3**).

S2: Secure runtime environment. OMNILOG must have the ability to create a secure runtime environment (protected from the operating system) for its own context.

OMNILOG's secure runtime environment requires (a) complete memory protection (i.e., read, write, and execute protection) and (b) trusted call gates. Read protections prevent the attacker from stealing provisioned secrets (**S1**). Write protections prevent the attacker from overwriting the code or data of trusted components, protected log buffers (OMNIBUFFER), and system boot configurations. Execute protections prevent the attacker from initiating arbitrary control-flow transfers to the execution of trusted components and attempt to sabotage the log protection or persistence process. Moreover, since a compromised operating system might launch these attacks using Direct Memory Access (DMA)-capable devices, memory protection is needed from devices too. Finally, trusted call gates control the execution of trusted components and verify parameters passed by untrusted callers (e.g., through registers or shared memory) before execution.

All security monitor designs we consider can create secure runtime environments. In particular, to restrict CPU memory accesses, the x86 hypervisor layer and Arm EL3 can leverage EPT [38] and TrustZone Address Space Controller (TZASC) [37] features, respectively. Also, the Input-Output Memory Management Unit (IOMMU) on x86 [38] and System Memory Management Unit (SMMU) [17] on Arm can protect against malicious devices. Finally, they have trusted call gate mechanisms: hypercall (e.g., VMCALL, HVC) for the hypervisor and Secure Monitor Call (SMC) for Arm EL3.

S3: Protected device interface. OMNILOG requires exclusive OMNIDRIVE access to securely (a) store logs and (b) keep the monitor and trusted kernel images. To achieve this, OMNIMONITOR must be able to configure the computer's device configuration and interfaces to isolate OMNIDRIVE from the rest of the machine and maintain this configuration even when the operating system is compromised.

Device isolation is well-studied. In x86, hypervisor-level monitors can use the EPT and IOMMU to enable protected device access [98]. Also, on Arm EL3, the TrustZone Protection Controller (TZPC) and SMMU (Arm's IOMMU) can isolate devices and I/O memory ranges [37].

S4: Controlled power management. A compromised operating system should not be allowed to reset or shut down the computer while logs are still transient (in OMNIBUFFER). Hence, OMNIMONITOR must interpose on all power management interfaces traditionally available to the operating system to ensure that log entries in OMNIBUFFER are stored to OMNIDRIVE before power cycle or hibernation events.

Traditionally, computers have two categories of power management interfaces: (a) internal interfaces for CPUs like Advanced Configuration and Power Interface (ACPI) and (b) power-related external device interfaces (e.g., IPMI and BMC). The latter class of interfaces are generally controlled by system administrators using out-of-band channels. Hence, OMNILOG isolates these devices from the operating system (using **S3**). Some systems also have a watchdog timer for periodic reset. OMNILOG also isolates this timer in such systems.

Security monitor designs can also interpose on motherboard-related power events. In particular, in modern x86 machines, power reset and shutdown events are sent as ACPI function calls to UEFI [80]. Monitors can prevent such calls by (a) write-protecting UEFI runtime communication regions (using **S2**) and (b) exposing an alternate emulated power management interface [4]. Similarly, all power management events are delivered to Arm EL3 through the Power State Coordination Interface (PSCI) [10]. Thus, OMNIMONITOR can intercept power cycling requests and persist log entries with our custom secure monitor before power cycling the machines.

5.2 Efficient Protected Logging Pipeline

OMNILOG implements a new logging pipeline that avoids lengthy stalls and reduces storage I/O cost. This is critical since each generated log entry must be protected before its corresponding event happens. In particular, OMNILOG ensures that the event logger speedily generates log entries in a compact format, generated logs from concurrent processors are sent to OMNIMONITOR, and hence, protected, with a small latency, and the transient in-memory logs are always flushed to storage but in an asynchronous fashion.

Fast compacted log generation. Many real-world audit systems like the Linux Audit system [3] generate logs in human-readable format using raw data, but this conversion is slow and the format requires extra space [97]. Instead, OMNILOG's event logger generates log entries in raw format while further compacting them by removing static content which can be regenerated, implementing variable-length encoding, and using protected hash tables for lengthy strings. All these result in fast and storage-efficient log generation. During log compaction, OMNILOG maintains all information produced by the native auditing system for regeneration. For instance, all attributes recorded by Auditd (also called Linux Audit Subsystem or LAuS) are kept in our implementation (§6).

Each compacted log entry is generated with *metadata* to reconstruct its human-readable counterpart during future forensic analysis. This metadata structure is primarily needed because different system calls produce different log entries. For instance, network-related system calls have an additional socket address. Such conditional fields can be encoded in the metadata as `1` or `0` to denote their presence or absence.

Human-readable log entries typically contain static information [97]. For example, Auditd's system-call log entry always starts with `type=syscall, msg=audit(timestamp)`. Our

event logger removes all such information because it can be automatically generated during post-processing.

Although fields in log entries can take up a different number of memory bytes, a naive raw encoding would allocate a fixed size based on type. For instance, system call arguments would be encoded as 8-byte integers, even if they only occupy a single byte. To avoid such waste, OMNILOG implements variable-length encoding to pack each value of an entry to its minimum number of required bytes. The required number of bytes are transposed into the metadata of each log entry field. We chose to keep this packing at a byte-level instead of a bit-level since the latter resulted in a very large metadata (e.g., 64 bits instead of 8 bits for an 8-byte object).

Log entries also contain various lengthy strings that cannot be efficiently packed. For instance, entries can contain names and directory paths of the application being audited and special keystrings used to describe certain log rules. OMNILOG replaces such values with unique small counter-based integer identifiers. OMNILOG tracks the unique identifiers and their corresponding full string inside chained hash tables.

The string replacement hash tables are kept (a) in protected memory and (b) at a per-core granularity. The hash tables must be kept in protected memory, otherwise the attacker can tamper with them to avoid decompaction of log entries. Naively, it would result in a context switch to the OMNIMONITOR each time to retrieve the unique identifier. To avoid such extra context switches, OMNILOG ensures that the hash table is mapped *read-only* to the event logger's address space (using memory protection primitives discussed in **S2**). Hence, the event logger can directly read the hash table to retrieve integers but must go through OMNIMONITOR to update the hash table. Moreover, the tables are kept individually for each core to avoid serialization operations on their access. The processor core number is appended to the log entry to track which core's hash table the unique identifier belongs to.

Concurrent log protection. Once the event logger generates the compact log entry for an event, it protects the log entry before allowing the event's execution. The event logger submits the log entry to OMNIMONITOR for protection through shared memory (Figure 2 ❶–❷). Then, the logger context switches to the monitor, which copies the log entry to OMNIBUFFER (❸) and returns (❹). At this point, the logger passes the event to its corresponding event handler (❺).

Log protection is within the critical path of an event's execution; hence, it must be optimized to avoid system slowdown. There are two challenges we must overcome to ensure efficient log protection. First, multiple threads can concurrently execute a system call. To maintain their concurrency, the log protection path must minimize synchronization operations while ensuring no race conditions. Second, OMNIBUFFER is finite and volatile, so we must flush it to OMNIDRIVE before it fills up. If a thread executes a system call when OMNIBUFFER is filled, the logger blocks the thread until the log keeper flushes OMNIBUFFER and the event's entry is protected. This blocking degrades performance; thus, the system must ensure high availability of OMNIBUFFER.

We address these issues by constructing OMNIBUFFER as *per-core double buffers*. We assign local buffers to each core to transfer and save log entries while avoiding any serialization operations on the critical path. This leads to a reordering of log entries, but it can always be corrected as each log entry contains a unified timestamp (obtained using a CPU instruction like RDTSCP which is constant for all cores in the same socket [38] or can be calibrated across multiple sockets [47]).

To minimize contention between log protection and persistence (explained in the next heading), OMNILOG augments per-core buffering with two demarcated buffers: b_m for OMNIMONITOR and b_k for the log keeper. Initially, both buffers are empty and the monitor adds new entries to b_m. When b_m becomes full, it swaps b_m for b_k (i.e., swaps their pointer values), allowing the log keeper to asynchronously flush b_k (which was b_m) to OMNIDRIVE while adding new entries to b_m (which was b_k).

If both b_m and b_k are full (which should only happen in an exceptional situation), OMNIMONITOR explicitly calls the log keeper to flush b_k, waits until b_k becomes empty, and then swaps b_m and b_k. Since a buffer swap operation could be requested while log keeper is flushing b_k, it is serialized with the log keeper's buffer flushing operations to avoid a race condition. This buffer flushing does not add significant synchronization delay because it occurs infrequently and swapping is fast. (i.e., it only swaps pointer values.)

Asynchronous persistence with fallback. Log entries transient in OMNIBUFFER must be flushed to OMNIDRIVE to avoid delays if the buffer is full and losing logs if there is a power event. At the same time, the buffer should not be flushed frequently because this buffer flushing affects the system performance—i.e., it blocks buffer swap operations and thus event execution.

To satisfy both requirements, the event logger asynchronously runs the log keeper if b_k becomes full. (i.e., the monitor swaps b_m and b_k.) In particular, after protecting each log entry on a buffer, OMNIMONITOR returns a buffer status to the event logger (Figure 2 ❹). If the status indicates that b_k is full, the event logger unblocks a background thread which context switches to OMNIMONITOR (Ⓐ) and asks it to execute the log keeper (Ⓑ). Then, the log keeper flushes the filled b_k to OMNIDRIVE (Ⓒ–Ⓓ). It flushes other b_k's (belonging to different cores) as well if they are full.

OMNILOG also implements a *fallback* mechanism to prevent transient log entries from being discarded if a compromised operating system does not notify log keeper and resets or turns off the machine. Specifically, every power event that the operating system can trigger is intercepted by OMNIMONITOR (§5.1) which then executes the log keeper to flush all log entries in both b_m and b_k. Hence, all log entries generated before the compromise are persisted.

	OmniLog-Arm			OmniLog-x86		
Component	**Base**	**Ver.**	**SLoC**	**Base**	**Ver.**	**SLoC**
OmniMonitor	TF-A [11]	2.6	551	Linux/KVM	5.16.9	573
OmniKernel	OP-TEE [92]	3.10.0	158	Linux kernel	5.16.9	274
Logging subsystem	Linux kernel	5.15.32	2551	Linux kernel	5.16.9	1863

Table 2: Base software framework and the source lines of C code we added or changed for our two prototypes.

The audited system's performance under OmniLog is bounded by OmniDrive throughput. In particular, if a thread running on a processor core begins to execute a system call while the core's b_m and b_k are both full, the event logger blocks the system call's execution until the log keeper flushes b_k. Our evaluation will show this aspect (§8). Moreover, it is the administrator's duty to prevent OmniDrive from filling up, otherwise, the computer will halt until the stored log entries are retrieved, much like the policy of other systems [8].

6 Implementation

We demonstrate the wide applicability of OmniLog by prototyping it for Arm and x86. We summarize the source lines of C code we added to or changed in individual software components in Table 2. We will open-source them to help foster research and development.

6.1 OmniLog-Arm

We implemented OmniLog for an Arm machine which supports TrustZone and other required security features.

Hardware specification. We used an NXP IMX8MQ-EVK board featuring an i.MX 8MQ application processor with four Cortex A53 cores running at 1.5 GHz, 3 GiB of RAM, and 16 GB eMMC. The device ran Debian 9 with Linux kernel 5.15.32 as its main operating system on the eMMC module. We attached a 64 GB SD Card to the device to use it as OmniDrive. This board provides required hardware-based security features: TZASC to isolate DRAM, Central Security Unit (CSU) to isolate device memory (like TZPC), Resource Domain Controller (RDC) to restrict the device's DMA (like SMMU), and PSCI to manage power events [69].

OmniMonitor: TF-A runtime firmware at EL3. We implemented the monitor by modifying TrustedFirmware-A (TF-A) [11]. TF-A consists of early boot code and runtime firmware which executes at EL3 as the secure monitor. The rest of this section describes how we enabled all critical OmniLog security invariants (§5.1: **S1**–**S4**).

We configured our board to load OmniMonitor on power-on or reset (**S1**). In particular, we set its DIP switch to always boot from OmniDrive (i.e., the SD Card) containing the boot code (TF-A and U-Boot SPL 2020.04 [91]) and the OmniLog code. By default, the boot code is configured to load the monitor prior to any other code. At runtime, the monitor prevents the untrusted operating system and devices from corrupting OmniDrive (**S3**). It configures the CSU to make the SD Card controller for OmniDrive a secure-world device and the RDC to prevent other devices from accessing the controller. That is, it sets the `CSU_CSL_USDHC2` register to make the controller only accessible to TrustZone. Also, it sets the `RDC_MDA_A53` and other `RDC_MDA_*` registers to assign the A53 cores to a specific domain while assigning other DMA masters to other domains, and sets the `RDC_PDAP_USDHC2` register to make the SD Card controller only accessible to the A53 domain [11, 69]. These settings remain locked until the next device reset.

OmniMonitor establishes the secure runtime environment once it is loaded (**S2**). It configures the TZASC to restrict access to secure-world memory regions from non-secure exception levels (for the main operating system and applications) and configures the RDC to prevent arbitrary devices from accessing the secure-world memory regions. Then, it locks these settings such that they cannot be changed until the next device reset. The monitor also specifies an SMC handler to protect log entries on a secure-world memory buffer OmniBuffer which is accessible to OmniKernel.

For controlled power management (**S4**), OmniMonitor (a) configures PSCI to intercept software-driven power events and (b) isolates the board's watchdog timer from the operating system using the RDC and CSU. This allows the monitor to flush all buffered logs to OmniDrive before power events.

OmniKernel: OP-TEE at S-EL1. We implemented OmniKernel using Open Portable TEE (OP-TEE) [92] which runs at Secure EL1. OmniMonitor loads it before the Linux operating system. We implemented its log keeper as a Pseudo Trusted Application (PTA). For ease of implementation, we did not port a storage device driver to OP-TEE. Instead, we enabled TF-A's MMC driver (847 source lines of C code in total) and exposed SMC handlers to allow OmniKernel to access it. This simple driver's performance is limited (§8). In the future, a custom storage driver can be implemented for OmniKernel using automated tools [32].

Logging subsystem. Our event logger is based on LAuS [3]. In particular, we replace its formatted-string-based log generation with our compact log generation (§5.2). Also, we access the Arm processor's `CNTVCT_EL0` register to obtain a timestamp value. To share log entries with OmniMonitor once they are generated, the event logger maintains a tiny buffer dedicated to store a single log entry for each processor core which is accessible to OmniMonitor. The event logger asynchronously instructs OmniKernel to store a part of OmniBuffer (i.e., b_k) to OmniDrive if the part is full.

6.2 OmniLog-x86

We implemented OmniLog for an x86 machine using virtualization. Although we think that a lightweight hypervisor [42, 65, 80, 84] is ideal, there is no publicly available

and mature lightweight x86 hypervisor. Hence, our prototype used a full-featured hypervisor (i.e., Linux KVM [50]) while replicating a lightweight hypervisor's setup (e.g., device passthrough). This approach has TCB and deployment challenges in certain scenarios, but both can be overcome as we explain in the next paragraphs.

In x86 enterprise servers with multiple virtual machines, OMNIMONITOR would execute alongside a hypervisor, increasing its TCB. This TCB problem can be mitigated using a compartmentalization approach [81] to isolate a security monitor (including OMNIMONITOR) from the remaining hypervisor. Also, we can use memory lockdown for code integrity and compiler instrumentation for control-flow integrity [93].

The main deployment hurdle of a virtualization design is the overhead of running laptop or desktop systems inside full VMs. However, even though our current implementation considers this design for prototyping ease, a *host-only virtualization* approach is ideal. This approach implements hardware-assisted checks (capable of supporting all primitives in §5.1) on the system's execution with passthrough access to all permitted devices. Host-only virtualization provides near-native performance since hardware checks are lightweight [2]. Moreover, it is enabled by default in Windows 11 [67].

Hardware specifications. Our x86 machine featured an Intel Core i5-12600K CPU with 10 physical cores (16 logical threads) at up to 4.9 GHz, 20 MiB of L3 cache, and 32 GiB of DDR5 RAM. We attached two 1 TB PCIe SSDs (Teamgroup MP33) to the machine to use as OMNIDRIVE and operating system storage, respectively. The machine ran Ubuntu 22.04 with Linux kernel 5.16.9 as its KVM (hypervisor) host.

OMNIMONITOR: KVM hypervisor at VMX root mode. We implemented OMNIMONITOR by modifying KVM. To securely boot the machine into OMNIMONITOR when it is powered on or reset (**S1**), we changed UEFI settings to make OMNIDRIVE the primary boot device. In addition to the monitor, the drive also contained the system bootloader (GRUB v2.06) and OMNIKERNEL. Once loaded, the monitor prevents untrusted code and devices from corrupting the UEFI NVRAM (where boot configurations are stored) and OMNIDRIVE (using **S2** and **S3** primitives below).

OMNIMONITOR creates a secure runtime environment and protected storage using VMX features. The operating system (Ubuntu 22.04) runs inside a VM and its access is restricted within the VM using the EPT and IOMMU (**S2**). The VM is provided passthrough access to all non-critical devices (e.g., untrusted storage) using Virtual Function I/O (VFIO) [88], but the IOMMU configuration prevents it from using these devices to access outside regions. Also, to prevent DMA redirection attacks [77] (**S3**), the monitor assigns it an isolated IOMMU group with PCIe Access Control Services (ACS) [7].

The monitor specifies two hypercall handlers to (a) protect log entries to OMNIBUFFER and (b) persist log entries by calling OMNIKERNEL. Both handlers check the caller's Current Privilege Level (CPL) (which is stored inside the machine control structure on a hypercall [38]) and only interact with operating system-level code (e.g., event logger). Finally, to ensure controlled power management (**S4**), we modified KVM's ACPI emulation code to intercept the operating system's power event requests and flush buffered logs.

OMNIKERNEL: Linux at VMX non-root mode. We implemented OMNIKERNEL based on Ubuntu 22.04 (kernel 5.16.9) as a separate single-core VM and the log keeper as its kernel module. The log keeper had direct access to OMNIBUFFER, which was implemented using KVM's inter-VM shared memory (ivshmem). Using a separate ivhsmem location, it waited for log storage notifications and handled them. To avoid external attacks, the VM was not assigned any other device (e.g., network-related) apart from the storage device.

Logging subsystem. OMNILOG-x86's event logger is similar to OMNILOG-Arm's (§6.1) except for two aspects. First, it uses the `RDTSCP` instruction to obtain a timestamp value. Second, it communicates with OMNIMONITOR using hypercalls.

Limitations: Both prototypes currently do not support log retrieval. It does not affect our evaluation (§8). Implementing log retrieval is a matter of engineering as both Linux and OP-TEE provide feature-rich drivers and libraries [5, 89].

7 Security Claims and Analysis

This section shows that OMNILOG achieves synchronous log protection (§3: **P2**)—all log entries whose events executed before kernel compromise (time t_c) are available and unchanged—using five claims. Here, "an event executes at time t" means that the event begins to be processed by a corresponding event handler inside the kernel at time t.

Claim 1. *All log entries for events executed before t_c will be stored in* OMNIBUFFER.

The operating system's event logger behaves correctly until t_c (§2.3)—it will faithfully execute all logging subsystem tasks delegated to it by OMNILOG until this point (§4.1). In particular, before t_c, the event logger will allow an event (which is a system call based on our policy) triggered by a process to occur only after the logger (a) creates a log entry for the event and (b) calls OMNIMONITOR to synchronously copy the log entry to OMNIBUFFER. Once the entry is copied into OMNIBUFFER (before the event's execution), it cannot be modified by the attacker, no matter how fast our attacker can compromise the operating system after the event (§5.1). Thus, for any event that executes before t_c, the corresponding log entry will also have been saved before t_c.

Claim 2. *All log entries that arrive at* OMNIBUFFER *will be saved to* OMNIDRIVE.

All log entries stored in OMNIBUFFER are handled by our system's trusted components, OMNIMONITOR and OMNIKERNEL. These components (and the buffer) are protected from the attacker by OMNILOG's secure runtime environment (§5.1: **S2**), and their correct behavior is ensured

by our assumption that both the hardware and the OMNILOG software running in it are free of bugs (§2.3).

The log keeper component of OMNIKERNEL (a) periodically and (b) before the system turns off persists buffered log entries to OMNIDRIVE (§5.2). In particular, the keeper will be notified periodically by the event logger to send logs to disk. If OMNIBUFFER is full and a new log entry arrives, OMNIMONITOR will invoke the log keeper to ensure buffered logs are persisted before appending new log entries to the buffer. Finally, OMNIMONITOR interposes on the execution of all operating system-driven power events on the machine, and at such an event will invoke the log keeper to ensure entries are persisted before the power event occurs (**S4**).

Claim 3. *The operating system can never directly or indirectly (e.g., through devices) access* OMNIDRIVE.

At the first system boot, OMNIMONITOR will load and run before the operating system. This is ensured by the provisioning of an initial trusted boot configuration by our system administrator (§5.1: **S1**). Once booted, the monitor configures the hardware (**S2**–**S3**) to protect its own context, OMNIKERNEL, OMNIDRIVE (where trusted boot images and logs are kept), and boot configurations from the operating system. These hardware configurations prevent the operating system from corrupting trusted computations and OMNIDRIVE directly or indirectly (via controlled DMA master devices). OMNIMONITOR will allow the operating system to boot only after these protections are in place. Finally, boot configuration protections ensure that, on each subsequent boot, OMNIMONITOR will run before the operating system and repeat these configuration steps.

Claim 4. *Only the system administrator can remove log entries from* OMNIDRIVE.

All log retrieval and subsequent removal commands are guaranteed to come to the log manager (in OMNIKERNEL) from the system administrator. In particular, the commands are accepted only after a remote administrator has initiated and successfully completed an authenticated TLS connection with the manager. The connection's authenticity is ensured by the administrator's public key and OMNILOG's private keys, both of which are securely provisioned in the boot image by the administrator (**S1**), kept in protected memory at runtime (**S2**), and securely persisted in OMNIDRIVE (**S3**).

Claim 5. OMNILOG *ensures the integrity and availability of all log entries whose events executed before* t_c.

The integrity and availability guarantee for OMNILOG follows by combining the previous four claims.

8 Evaluation

8.1 Experimental Setup

For each of our two implementations, the experimental setup consists of the target device and a workload generator. We ensure that the workload generator (a) is separated from the target so that it does not interfere with the measurements and (b) has sufficiently high bandwidth to saturate the target. The radically different computational power of our two hardware devices leads us to different experimental setups.

OMNILOG-Arm. We run the workload generator on a separate PC (with an Intel Core i5-8250U CPU featuring eight logical threads and 16 GiB of RAM) connected to the target Arm device over 1 Gb Ethernet. This ensures isolation and saturation of all four i.MX 8MQ cores. On the target device, we allocate 32 MiB of RAM (out of 3 GiB) to the secure world (i.e., TF-A, OP-TEE). Out of these 32 MiB, we allocate 512 KiB for OMNIBUFFER (i.e., two 64 KiB buffers per core).

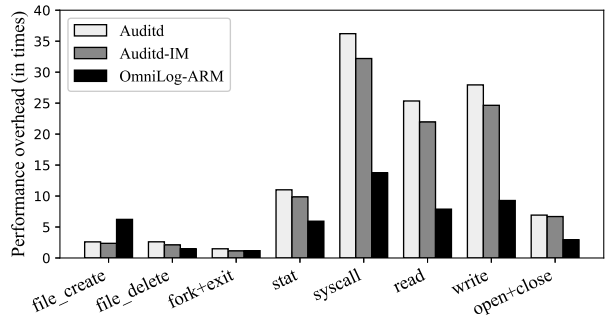
OMNILOG-x86. We partition our physical machine between the workload generator and the target because saturating our machine over a network connection from a separate computer would have required high-speed networking equipment and a very fast second computer (neither of which we had readily available). We configure our target to be a VM with 8 GiB of RAM and four virtual processors backed by four pinned logical threads. OMNIKERNEL runs in a separate VM with 1 GiB of RAM and one virtual processor. Out of the 1 GiB, we allocate 128 MiB for OMNIBUFFER (i.e., two 16 MiB buffers per core). Most of the remaining resources (i.e., 23 GiB of RAM and 11 logical threads) are available to the workload generator which we run on the host system.

Other configurations. We compare OMNILOG against the **Auditd** and **Auditd-IM** configurations defined in §3.1. In all three cases, logging is enabled for all system calls. We present the results normalized over a baseline in which logging is turned off. **Auditd-IM** also serves as a proxy to compare against several other audit systems [8, 72, 73] because it is faster than the current state-of-the-art HardLog [8], which incurs additional performance cost (over **Auditd-IM**) due to log buffering and criticality-aware protection.

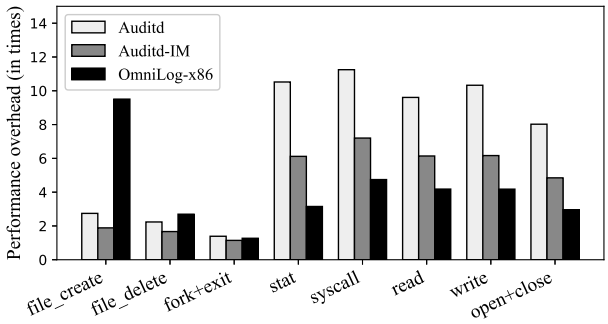
Auditd (and **Auditd-IM**) typically does not run under virtualization in non-server machines, but we ran them in the same VM environment as OMNILOG-x86 for fair comparison. In particular, our VM-based OMNILOG implementation adds extra performance overhead to experiments (over bare-metal) because several hardware devices (e.g., GPU) and features (e.g., ACPI) are virtualized. This overhead is not incurred in an ideal implementation (§6.2). Hence, to avoid its impact on our experiments, we run all configurations inside the VM.

8.2 Micro-benchmarks

Log generation latency. OMNILOG and **Auditd-IM** generate log entries synchronously for each event, adding latency. To estimate it, we make one million system calls to `getpid` in a tight loop and measure the total number of cycles for a baseline in which auditing is turned off, and **Auditd-IM** and a



(a) Arm platform



(b) x86 platform

Figure 3: LMbench overhead.

version of OMNILOG in which only log generation is enabled. We subtract the baseline cycle count from the cycle counts for the other two configurations to obtain the net overhead.

Results: The log generation latencies of OMNILOG are 3,856 cycles (Arm) and 3,407 cycles (x86). The corresponding latencies of **Auditd-IM** are 22,747 cycles and 8,001 cycles. These numbers are averages over ten runs. The standard deviation was less than 2% in all cases. In summary, OMNILOG has a significantly lower log generation latency.

Log protection latency. OMNILOG's synchronous log protection adds further latency in addition to log generation. To measure it, we repeat the previous experiment with a different variant of OMNILOG which enables both log generation and synchronous copying to OMNIBUFFER.

Results: The combined log generation and protection latencies are 4,795 cycles (Arm) and 5,324 cycles (x86). That is, the additional latencies to copy a log entry (∼68 bytes for `getpid` §8.4) to a secure region are 939 cycles (Arm) and 1,468 cycles (x86). The x86 version takes longer due to the expensive hypercall and second-level address translation. The combined latencies for OMNILOG are still noticeably lower than those of **Auditd-IM**.

Amortized log storage latency. Although OMNILOG log storage is asynchronous, it may affect the system call latency because OMNIBUFFER is finite. Especially, if logs are frequently generated like in our `getpid` micro-benchmark, all buffers can fill up and OMNILOG will block system calls until it flushes a full buffer to storage. We measure these effects on our experiment with fully enabled OMNILOG.

Application	Parameters
NGINX	Four worker threads in a local network; tested with Apachebench for 10k requests and 32 concurrency
Memcached	Default settings; benchmarked with default memaslap for 120 s with a concurrency level of 16
Redis	Ran with 16 databases; benchmarked using a mix of 1M set and 1M get commands with 50 clients and a pipeline of 32
SQLite	Phoronix benchmark pts/sqlite-speedtest
Chromium	Ran Speedometer 2.0 in a chromium browser GUI
OpenSSL	Phoronix benchmark pts/openssl
7zip	Phoronix benchmark pts/compress-7zip
GNU Octave	Phoronix benchmark system/octave-benchmark

Table 3: Description of real world workloads.

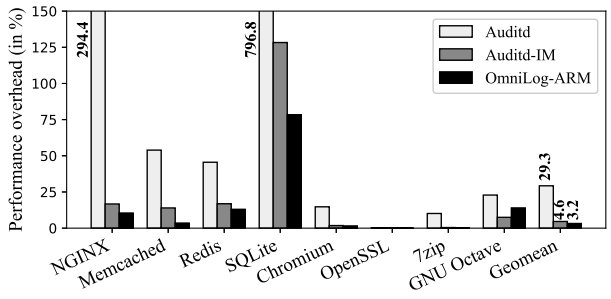
Results: The average latencies for full OMNILOG are 8,913 cycles (Arm) and 6,286 cycles (x86). That is, the additional latencies mainly due to occasional stalls for buffer flushing are 4,118 cycles (Arm) and 962 cycles (x86). The impact of asynchronous storage on x86 is moderate thanks to high-bandwidth PCIe SSD storage. In contrast, on Arm, the average overhead per system call almost doubles. This is the result of our simple storage driver which can operate the SD Card only up to a bandwidth of 10 MiB/s (as opposed to up to 104 MB/s supported by the i.MX 8MQ SD Card controller [69]).

System call latency (LMBench). LMBench [64] is a micro-benchmark suite focusing on low-level system functions. We ran a collection of LMBench micro-benchmarks for all three logging configurations and the no-audit baseline. The concrete list of benchmarks is displayed on the x-axis of Figure 3.

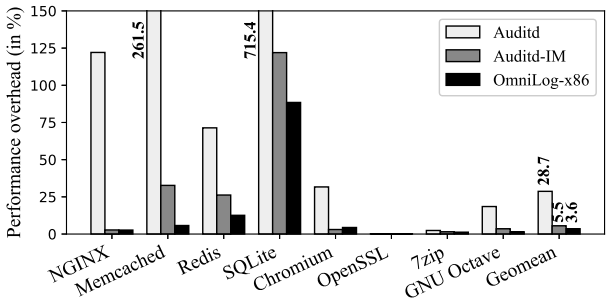
Results: Figure 3a and Figure 3b show the overhead for Arm and x86. Due to the micro-benchmark nature of these workloads which involve unusually high rates of system calls per second, the overheads are significant for all configurations. OMNILOG is generally significantly faster than **Auditd** and noticeably faster than **Auditd-IM**. The main exception is `file_create` which creates a large number of files with different (random) names. The corresponding large number of unique filename strings causes the hash table heuristic we use to generate compressed logs (§5.2) to become very slow. It appears unlikely that the problem will arise with real applications. If necessary, OMNILOG could also stop using the hash table heuristic if it observes hit ratios to be low.

8.3 Real-world Programs

Common settings. We evaluate OMNILOG on eight real-world programs, selected based on prior work [8] to facilitate comparison. Our test applications are OpenSSL, Chromium, 7-zip, GNU Octave, Memcached, Redis, SQLite, and NGINX. The configuration details and workload settings are in Table 3.



(a) Arm platform



(b) x86 platform

Figure 4: Performance overhead of the real world workloads.

Arm platform			x86 platform		
Buffer	**Overhead**	**Fill-ups**	**Buffer**	**Overhead**	**Fill-ups**
512 KiB	0.0%	0	128 MiB	0.0%	0
256 KiB	2.0%	1	16 MiB	0.0%	0
128 KiB	7.3%	10	2 MiB	0.0%	0
64 KiB	27.8%	1354	128 KiB	1.3%	11

Table 4: OMNILOG's performance scalability on different buffer sizes while running SQLite. Shaded row is for default settings. The reported performance overhead is relative to OMNILOG's baseline performance with default buffer settings.

Workload	Arm platform		x86 platform	
	Auditd-IM	**OMNILOG**	**Auditd-IM**	**OMNILOG**
NGINX	5270	5624	23471	23724
Memcached	2043	2211	48290	63499
Redis	1755	1946	21526	23162
Chromium	1849	1987	11616	12129
OpenSSL	480	477	112	83
7zip	371	387	473	481
SQLite	27904	35717	245968	294907
GNU Octave	3182	3182	25967	31412

Table 5: System calls per second per core of the real world workloads.

We averaged results over five runs. The standard deviation was less than 4%.

Results. Figure 4a and Figure 4b show the overhead. The geometric mean overhead of OMNILOG is 3.2% on Arm and 3.6% on x86. It is quite low considering that all system calls are being logged. The geometric mean overhead of **Auditd-IM** is slightly higher than OMNILOG's on both Arm and x86 in spite of not storing logs. This overhead again demonstrates the impact of OMNILOG's significantly faster log generation. The geometric mean overhead for **Auditd** is noticeably larger on Arm (29.3%) and x86 (28.7%), 9.1× and 8.0× higher than OMNILOG's on these platforms, respectively.

Across applications, the overhead of OMNILOG is generally low except for SQLite. SQLite performs system calls at a significantly higher rate than the other applications. We have measured more than 550k system calls per second per core for native SQLite (i.e., no logging) on our x86 machine.

Buffer scalability analysis. We investigated whether both buffers (b_k and b_m) could fill up and cause delays for our most intensive benchmark, SQLite (Table 4). Under default buffer sizes, we observed no such stall since the storage throughput on each platform was significantly higher than the log generation throughput (§8.2). It is worth noting that even under default buffers, OMNILOG was using significantly less memory than **Auditd**, which we configured to use ∼300 MiB in our experiments like prior work [8]. However, to observe the impact of buffer sizes and fill-up delays, we experimented with small buffers on each platform. We observed delays for buffer sizes at and below 128 KiB on both platforms, but the additional overhead compared to the default buffer settings was small. These buffer sizes are miniscule and unrealistic. Nevertheless, they show that OMNILOG's end-to-end logging pipeline remains highly efficient even under tiny buffers.

Context switch statistics. In OMNILOG, each system call results in a switch to OMNIMONITOR to synchronously protect a corresponding log entry. To identify its statistics, we measured each program's per-core system call rate under OMNILOG and **Auditd-IM** (Table 5). On Arm, system calls per second per core range from several hundreds to ∼36k (SQLite). For x86, the per-core call rates range up to ∼295k (SQLite). OMNILOG supported more system calls per second per core (with lower overhead) than **Auditd-IM**.

Key performance takeaways. Across the real-world programs, OMNILOG performs better than **Auditd** and even an unrealistic completely in-memory **Auditd-IM**. There are a few aspects of its compaction (e.g., hashing) that can result in higher overhead, and they can be automatically adjusted with profiling. Moreover, like *all* general-purpose audit systems, OMNILOG has a limit as to how frequently it can produce logs; hence, it can incur high overheads for workloads which frequently make system calls (e.g., SQLite). Nevertheless, OMNILOG's log production limit is significantly higher than that of other systems, allowing it to synchronously log all system calls with a low (3%–4%) typical overhead. Therefore, it is suitable for the vast majority of workloads today.

8.4 Storage

Log entry sizes. We computed the average log entry size for the logs generated by OMNILOG and **Auditd** in the `getpid`, NGINX and Memcached runs of the previous subsections by dividing the total log size by the number of log entries. For `getpid`, the average **Auditd** log entry was 348 bytes vs. 68 bytes for OMNILOG. For NGINX, the results were 320 bytes for **Auditd** vs. 55 bytes for OMNILOG. For Memcached, the sizes are 335 bytes for **Auditd** vs. 56 bytes for OMNILOG. In each case, OMNILOG reduced the log entry size by 5–6×.

Storage requirements. We evaluate how OMNILOG's high event coverage affects storage requirements. We compare the log sizes produced by OMNILOG with those of an established audit policy used by many previous audit systems [8, 72, 73] for three applications: Redis, Memcached, and SQLite. For Memcached, we configure the memaslap benchmark to execute one million operations (get and set). We run Redis with 1000 parallel connections/clients, sending a total of 50 million requests. For SQLite, we run the SQLite-speed test in the Phoronix test suite under the default setting. We run each application with the specified configuration twice: with OMNILOG and with **Auditd** under the academic rule set.

For Memcached, the log sizes are 778 MiB for OMNILOG and 3.3 GiB for the academic rule set. The significantly larger size of the latter indicates that it contains a large fraction of the log entries of the OMNILOG log. Every Memcached request involves network connections that the academic rule set logs. The rest of the size difference is explained by OMNILOG's compact log entry size. For Redis, similarly, the log sizes are 314 MiB for OMNILOG and 2.1 GiB for the academic rule set. Again, it appears that the bulk of the log entries comes from network-related system calls. The situation is different for SQLite. The log sizes are 5.9 GiB for OMNILOG and a mere 11 MiB for the academic rule set. This is because SQLite frequently uses `pread64` and `pwrite64` to access files, but the academic rule set does not log them.

In summary, relative log sizes are highly workload-dependent. Much of the variability arises from the system calls traditional audit policies monitor. There are widely used applications for which OMNILOG, in spite of its high event coverage, does not increase storage requirements.

9 Discussion

General limitations. OMNILOG has a few limitations in terms of deployment compared to software-only secure logging mechanisms like cryptography-based tamper-evident logging [34, 73]. Specifically, it requires hardware supporting the security features mentioned in §5.1 and an ISA-specific security monitor (§6). Both are reasonable because such security features are widely available nowadays, and the ISA-specific modification is small (less than 600 SLoC Table 2).

Intra-kernel privilege separation. Existing systems [20, 21] create security monitors at the same privilege level as the operating system (e.g., ring-0), minimizing hardware dependencies. In particular, Nested Kernel [21] relies on the x86 write-protection feature to implement a security monitor, as well as protected memory to secure critical kernel data structures and access logs. SVA [20] relies on compiler-based Software Fault Isolation (SFI) [23] to restrict the kernel's memory accesses and implement a security monitor. Both systems can be potentially extended to support OMNILOG's security abstractions. However, it would require non-trivial code changes [14, 16] to use an IOMMU for device protection.

HardLog comparison. OMNILOG is inspired by HardLog [8]'s secure logging requirements. However, OMNILOG ensures high event coverage and synchronous log availability that HardLog's inefficient audit device and conventional logging pipeline cannot solve. OMNILOG realizes a protected runtime environment by carefully leveraging widely-available security abstractions (§5.1), formulating new abstractions like a host-based device isolation interface (**S3**) and rethinking system power management aspects (**S4**). Further, it points out the logging pitfalls that all existing audit systems (including HardLog) suffer from (§3) and efficiently overcomes them by redesigning the end-to-end logging pipeline (§5.2).

x86-based Trusted Execution Environments (TEEs). The commercial x86 TEEs, AMD SEV [22], Intel SGX [63], and Intel TDX [39], are mainly used today to prevent cloud operators from leaking sensitive user code and data. They focus on confidentiality and integrity but not availability. Untrusted system code (i.e., the OS and/or hypervisor) retains control of the hardware resources including storage devices and power management to allow flexible resource management by cloud providers. These aspects fundamentally prevent x86 TEEs from satisfying OMNILOG's critical requirements: protected device interface (**S3**) and controlled power management (**S4**). Thus, we leveraged virtualization for our x86 implementation.

Audit security history. The observation that attackers can use software vulnerabilities to tamper with audit data goes back to at least Karger and Schell's 1974 Multics security evaluation [46, 3.4.4]. However, the threat received little direct attention. TCSEC [71], the influential security standard of the time, put the focus of auditing on assuring *individual accountability* (i.e., logging the actions of registered users) and aimed for medium to high-assurance systems in which kernel-level vulnerabilities would be rare or not exist. Several influential systems follow this general approach [25, 31]. In contrast, the VAX security kernel [45], a high-assurance VMM for the lower-assurance VMS and Ultrix guest operating systems, considered the guests as *untrusted subjects* and implemented all relevant security functionality separately. OMNILOG also uses a protected layer to implement important security functionality. However, OMNILOG leaves the bulk of the security functionality in the operating system and lets

it cooperate with the protected layer to maintain both security and performance.

Cross-boundary calls. With OMNILOG, the event logger's calls into OMNIMONITOR (§4.1) have the character of a microkernel Remote Procedure Call (RPC) [6, 57]. Notably, OMNILOG shares the main techniques of the highly optimized lightweight RPC [15] (i.e., simple control and data transfer and design for concurrency).

RISC-V port. OMNILOG can be extended to another popular architecture, RISC-V, as it has hardware features satisfying the four security abstractions (§5.1). RISC-V has an architectural system management mode known as Machine-mode (M-mode), Physical Memory Protection (PMP) [55] to restrict CPU memory accesses, and an IOMMU [82]. Their combination can satisfy **S1**–**S3**. Also, all power management events are delivered to M-mode through the Supervisor Binary Interface (SBI) [78], satisfying **S4**.

10 Related Work

Efficient logging. Numerous researchers have proposed efficient logging schemes by optimizing log generation and management or reducing log size. Logging schemes typically generate human-readable log entries which is slow due to format string handling. Instead, efficient logging schemes [66, 79, 97] generate encoded log entries and construct human-readable ones later. Also, logging schemes need to exchange log entries between the producer (e.g., a kernel component) and consumers (e.g., user services). Conventional schemes use a socket-based channel for log transfers which is portable but slow. Instead, efficient logging schemes [8, 59, 60, 74] use shared memory to reduce such communication overhead. In addition, some logging schemes compress log entries to reduce log storage overhead [24, 26, 35, 56, 86, 96]. OMNILOG also uses encoded log entries and the shared memory channel to improve its logging performance as well as a loseless compression method to reduce its overall log size before writing logs to storage.

Privileged monitoring. Several recent privileged monitoring schemes trap and log all important activities of an operating system in an environment that runs at higher privilege (e.g., hypervisor or TrustZone secure world) [12, 13, 49, 53]. However, these approaches suffer from the semantic gap problem [13, 41] because monitoring and logging are performed outside of the target context. Nested Kernel [21] does not have the semantic gap problem, but it requires nontrivial kernel code changes. Further, the extra code for monitoring and logging bloats the privileged environment. Unlike the these systems, OMNILOG generates log entries inside the operating system and leverages the privileged environment to construct secure and efficient log storage only. Thus, it neither suffers from the semantic gap problem nor heavily bloats the privileged environment.

11 Conclusion

Inefficiencies in the logging pipelines of current audit systems result in only a small selection of events being logged, while log entries remain subject to tampering. OMNILOG addresses both problems by monitoring all system calls and ensuring synchronous availability for all logs. It does so with low overhead by carefully redesigning key parts of the log generation and secure storage pipeline. We prototype OMNILOG on Arm and x86. Our evaluation shows that the overheads are low.

Acknowledgment

We would like to thank the anonymous reviewers and our shepherd for their helpful feedback.

References

[1] Dirty COW (CVE-2016-5195). https://dirtycow.ninja.

[2] Intel(r) clear containers 1: The container landscape. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-clear-containers-1-the-container-landscape.html.

[3] Linux kernel audit subsystem. https://github.com/linux-audit/audit-kernel.

[4] QEMU and ACPI BIOS generic event device interface. https://www.qemu.org/docs/master/specs/acpi_hw_reduced_hotplug.html.

[5] vsock - Linux VSOCK address family. https://man7.org/linux/man-pages/man7/vsock.7.html.

[6] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proc. of the USENIX 1986 Summer Technical Conference*, pages 93–113, 1986.

[7] Advanced Micro Devices and Hewlett-Packard. PCI Express Access Control Services (ACS), 2006. PCI-SIG Engineering Change Notice.

[8] Adil Ahmad, Sangho Lee, and Marcus Peinado. HardLog: Practical tamper-proof system auditing using a novel audit device. In *Proc. of the 43rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2022.

[9] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. In *Proc. of the 18th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 1997.

[10] Arm. Arm power state coordination interface: Platform design document, 2022.

[11] Arm Limited and Contributors. Trusted Firmware-A documentation, 2023. https://trustedfirmware-a.readthedocs.io/en/latest/.

[12] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: Real-time kernel protection from the Arm TrustZone secure world. In *Proc. of the 21st ACM Conference on Computer and Communications Security (CCS)*, Scottsdale, Arizona, November 2014.

[13] Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. A survey on hypervisor-based monitoring: approaches, applications, and evolutions. *ACM Computing Surveys (CSUR)*, 48(1):1–33, 2015.

[14] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Proc. of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, October 2012.

[15] Brian Bershad, Thomas Anderson, Edward Lazowska, and Henry Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[16] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in Linux. In *Proc. of the 2010 USENIX Annual Technical Conference (ATC)*, Boston, MA, June 2010.

[17] David Cerdeira, José Martins, Nuno Santos, and Sandro Pinto. ReZone: Disarming TrustZone with TEE privilege reduction. In *Proc. of the 31st USENIX Security Symposium (Security)*, August 2022.

[18] Ryan Cobb. SharpSploit, 2020. https://github.com/cobbr/SharpSploit/blob/master/SharpSploit/Evasion/ETW.cs.

[19] Eric Cole. *Advanced Persistent Threat: Understanding the Danger and How to Protect Your Organization*. Newnes, 2012.

[20] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proc. of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, October 2007.

[21] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proc. of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, March 2015.

[22] Advanced Micro Devices. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf, 2020.

[23] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Safecode: Enforcing alias analysis for weakly typed languages. In *Proc. of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Ottawa, Canada, June 2006.

[24] Hailun Ding, Shenao Yan, Juan Zhai, and Shiqing Ma. ELISE: A storage efficient logging system powered by redundancy reduction and representation learning. In *Proc. of the 30th USENIX Security Symposium (Security)*, August 2021.

[25] Glenn Faden and Christoph Schuba. *Case Study: Solaris Trusted Extensions*, pages 103–119. Springer, 2008.

[26] Peng Fei, Zhou Li, Zhiying Wang, Xiao Yu, Ding Li, and Kangkook Jee. SEAL: Storage-efficient causality analysis on enterprise logs with query-friendly compression. In *Proc. of the 30th USENIX Security Symposium (Security)*, August 2021.

[27] Foundeo Inc. Linux kernel - security vulnerabilities in 2023, 2023. https://stack.watch/product/linux/linux-kernel/.

[28] Foundeo Inc. Microsoft Windows 10 - security vulnerabilities in 2023, 2023. https://stack.watch/product/microsoft/windows-10/.

[29] Jessie Frazelle. Opening up the baseboard management controller. *Communications of the ACM*, 63(2):38–40, 2020.

[30] German Federal Office for Information Security. Operating System Protection Profile, 2010. https://www.commoncriteriaportal.org/files/ppfiles/pp0067b_pdf.pdf.

[31] V.D. Gligor, C.S. Chandersekaran, R.S. Chapman, L.J. Dotterer, M.S. Hetch, Wen-Der Jiang, A. Johri, G.L. Luckenbaugh, and N. Vasudevan. Design and Implementation of Secure Xenix. *IEEE Transactions on Software Engineering*, SE-13(2):208–221, 1987.

[32] Liwei Guo and Felix Xiaozhu Lin. Minimum viable device drivers for ARM TrustZone. In *Proc. of the 17th*

European Conference on Computer Systems (EuroSys), Rennes, France, April 2022.

[33] Mike Hanley. Updates to our policies regarding exploits, malware, and vulnerability research. https://github.blog/2021-06-04-updates-to-our-policies-regarding-exploits-malware-and-vulnerability-research/, 2021.

[34] Viet Tung Hoang, Cong Wu, and Xin Yuan. Faster yet safer: Logging system via fixed-key blockcipher. In *Proc. of the 31st USENIX Security Symposium (Security)*, August 2022.

[35] Md Nahid Hossain, Junao Wang, R. Sekar, and Scott D. Stoller. Dependence-preserving data compaction for scalable forensic analysis. In *Proc. of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[36] HP. HP Sure Recover whitepaper, 2021. https://www8.hp.com/h20195/v2/GetPDF.aspx/4AA7-4556ENW.pdf.

[37] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. vTZ: Virtualizing ARM TrustZone. In *Proc. of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, August 2017.

[38] Intel. Intel 64 and IA-32 architectures software developer's manual combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4, December 2022.

[39] Intel. Intel Trusted Domain Extensions (Intel TDX). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html, 2023.

[40] Intel Corporation. Intel Software Guard Extensions SSL (Intel SGX SSL) library, 2022.

[41] Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E Porter, and Radu Sion. SoK: Introspections on trust and the semantic gap. In *Proc. of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[42] Yuekai Jia, Shuang Liu, Wenhao Wang, Yu Chen, Zhengde Zhai, Shoumeng Yan, and Zhengyu He. HyperEnclave: An open and cross-platform trusted execution environment. In *Proc. of the 2022 USENIX Annual Technical Conference (ATC)*, July 2022.

[43] Ilan Kalendarov. Blindside: A new technique for EDR evasion with hardware breakpoints, 2022. https://cymulate.com/blog/blindside-a-new-technique-for-edr-evasion-with-hardware-breakpoints.

[44] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. SGX-Log: Securing System Logs with SGX. In *Proc. of the 12th ACM Asia Conference on Computer and Communications Security (ASIA CCS)*, 2017.

[45] P.A. Karger, M.E. Zurko, D.W. Bonin, A.H. Mason, and C.E. Kahn. A VMM security kernel for the VAX architecture. In *Proc. of the 1990 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 2–19, 1990.

[46] Paul Karger and Roger Schell. Multics Security Evaluation - Vulnerability Analysis. Technical Report AD/A-001 120, National Technical Information Service, June 1974.

[47] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. A scalable ordering primitive for multicore machines. In *Proc. of the 13th European Conference on Computer Systems (EuroSys)*, Porto, Portugal, April 2018.

[48] Kaspersky. The Slingshot APT, 2018. https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/03/09133534/The-Slingshot-APT_report_ENG_final.pdf.

[49] Samuel T King and Peter M Chen. Backtracking intrusions. In *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, October 2003.

[50] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux virtual machine monitor. In *Proc. of the Linux Symposium*, 2007.

[51] Andreas Klopsch. Remove all the callbacks – BlackByte ransomware disables EDR via rtcore64.sys abuse, 2022. https://news.sophos.com/en-us/2022/10/04/blackbyte-ransomware-returns/.

[52] Xeno Kovah and Corely Kallenberg. Advanced x86: BIOS and System Management Mode internals SPI flash protection mechanisms, 2014. http://opensecuritytraining.info/IntroBIOS.html.

[53] Srinivas Krishnan, Kevin Z Snow, and Fabian Monrose. Trail of bytes: Efficient support for forensic analysis. In *Proc. of the 17th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, November 2010.

[54] Arvind Kumar. *Active Platform Management Demystified: Unleashing the Power of Intel VPro Technology*. Intel Press, 2009.

[55] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proc. of the 15th European Conference on Computer Systems (EuroSys)*, Crete, Greece, April 2020.

[56] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. LogGC: Garbage collecting audit log. In *Proc. of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, October 2013.

[57] Jochen Liedtke. Improving IPC by kernel design. In *Proc. of the 14th ACM Symposium on Operating Systems Principles (SOSP'93)*, pages 175–188, 1993.

[58] Linux-audit. audit-userspace, 2022. https://github.com/linux-audit/audit-userspace/tree/master/rules.

[59] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. Kernel-supported cost-effective audit logging for causality tracking. In *Proc. of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2018.

[60] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. ProTracer: Towards practical provenance tracing by alternating between logging and tainting. In *Proc. of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[61] Thomas Malmberg. Splunk & auditd with Defender ATP and vulnerability scanning, 2022. https://www.mintsecurity.fi/en/splunk-auditd-with-defender-atp-and-vulnerability-scanning/.

[62] Mandiant. Targeted attack lifecycle. https://www.mandiant.com/resources/insights/targeted-attack-lifecycle.

[63] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. Innovative instructions and software model for isoalted execution. In *Proc. of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[64] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proc. of the USENIX 1996 Annual Technical Conference*, 1996.

[65] Microsoft Learn. Virtualization-based security (VBS), 2020. https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs.

[66] Microsoft Learn. Event tracing for Windows (ETW), 2021. https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/event-tracing-for-windows--etw-.

[67] Microsoft Learn. Memory integrity and VBS enablement, 2023. https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-hvci-enablement.

[68] Network Frontiers. Red Hat Enterprise Linux 8 security technical implementation guide, 2022. https://www.stigviewer.com/stig/red_hat_enterprise_linux_8/.

[69] NXP. i.MX 8M Dual/8M QuadLite/8M Quad applications processors reference manual, 2021.

[70] Department of Defense. National Industrial Security Program Operating Manual (NISPOM) – DoD 5220.22-m, 2013. https://www.nispom.org/NISPOMwithISLsMay2014.pdf.

[71] United States Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria (DoD 5200.28-STD)*. 1983.

[72] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher Fletcher, Andrew Miller, and Dave Tian. CUSTOS: Practical tamper-evident auditing of operating systems using trusted execution. In *Proc. of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2020.

[73] Riccardo Paccagnella, Kevin Liao, Dave Tian, and Adam Bates. Logging to the danger zone: Race condition attacks and defenses on system audit frameworks. In *Proc. of the 27th ACM Conference on Computer and Communications Security (CCS)*, November 2020.

[74] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. In *Proc. of the 2017 Symposium on Cloud Computing (SoCC)*, pages 405–418, 2017.

[75] PCI Security Standards Council. Payment Card Industry Data Security Standard version 4.0, March 2022. https://listings.pcisecuritystandards.org/documents/PCI-DSS-v4_0.pdf.

[76] Sean Peisert, Bruce Schneier, Hamed Okhravi, Fabio Massacci, Terry Benzel, Carl Landwehr, Mohammad Mannan, Jelena Mirkovic, Atul Prakash, and James Bret Michael. Perspectives on the SolarWinds incident. *IEEE Security & Privacy*, 19(2):7–13, 2021.

[77] Red Hat Customer Portal. A deep-dive into IOMMU groups, 2023.

[78] RISC-V Platform Specification Task Group. RISC-V Supervisor Binary Interface specification version 1.0.0, 2022.

[79] Kirk Rodrigues, Yu Luo, and Ding Yuan. CLP: Efficient and scalable search on compressed text logs. In *Proc. of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, July 2021.

[80] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proc. of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, October 2007.

[81] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. Deconstructing Xen. In *Proc. of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2017.

[82] SiFive. RISC-V IOMMU architecture overview. https://open-src-soc.org/2022-05/media/slides/RISC-V-International-Day-2022-05-05-14h10-Perinne-Peresse.pdf.

[83] Arunesh Sinha, Limin Jia, Paul England, and Jacob R Lorch. Continuous Tamper-Proof Logging Using TPM 2.0. In *International Conference on Trust and Trustworthy Computing (Trust)*, pages 19–36. Springer, 2014.

[84] Udo Steinberg and Bernhard Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *Proc. of the 5th European Conference on Computer Systems (EuroSys)*, Paris, France, April 2010.

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

[86] Yutao Tang, Ding Li, Zhichun Li, Mu Zhang, Kangkook Jee, Xusheng Xiao, Zhenyu Wu, Junghwan Rhee, Fengyuan Xu, and Qun Li. NodeMerge: Template based efficient data reduction for big-data causality analysis. In *Proc. of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, October 2018.

[87] Claudiu Teodorescu, Igor Korkin, and Andrey Golchikov. Veni, no vidi, no vici: Attacks on ETW blind EDR sensors. In *BlackHat Europe 2021*, 2021. https://i.blackhat.com/EU-21/Wednesday/EU-21-Teodorescu-Veni-No-Vidi-No-Vici-Attacks-On-ETW-Blind-EDRs.pdf.

[88] The kernel development community. VFIO - "Virtual Function I/O", 2023. https://docs.kernel.org/driver-api/vfio.html.

[89] The Mbed TLS Contributors. Mbed TLS documentation hub, 2023. https://mbed-tls.readthedocs.io/en/latest/.

[90] The Mitre Corporation. MITRE ATT@CK, 2022. http://attack.mitre.org/.

[91] The U-Boot development community. The U-Boot documentation, 2023. https://u-boot.readthedocs.io/en/latest/.

[92] TrustedFirmware.org. OP-TEE documentation, 2023. https://optee.readthedocs.io/en/latest/.

[93] Zhi Wang and Xuxian Jiang. Hypersafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proc. of the 31st IEEE Symposium on Security and Privacy (Oakland)*, May 2010.

[94] Richard Wilkins and Brian Richardson. UEFI Secure Boot in modern computer security solutions, 2013.

[95] Meng Xu, Manuel Huber, Zhichuang Sun, Paul England, Marcus Peinado, Sangho Lee, Andrey Marochko, Dennis Matoon, Rob Spiger, and Stefan Thom. Dominance as a new trusted computing primitive for the Internet of Things. In *Proc. of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[96] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. High fidelity data reduction for big data security dependency analyses. In *Proc. of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, October 2016.

[97] Stephen Yang, Seo Jin Park, and John Ousterhout. NanoLog: A nanosecond scale logging system. In *Proc. of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2018.

[98] Zongwei Zhou, Virgil D. Gligor, James Newsome, and Jonathan M. McCune. Building verifiable trusted path on commodity x86 computers. In *Proc. of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.