RESEARCH-ARTICLE

# The HitchHiker's Guide to High-Assurance System Observability Protection with Efficient Permission Switches

**CHUQI ZHANG**, National University of Singapore, Singapore City, Singapore

**JUN ZENG**

**YIMING ZHANG**, Southern University of Science and Technology, Shenzhen, Guangdong, China

**ADIL AHMAD**, School of Computing and Augmented Intelligence, Tempe, AZ, United States

**FENGWEI ZHANG**, Southern University of Science and Technology, Shenzhen, Guangdong, China

**HAI JIN**, Huazhong University of Science and Technology, Wuhan, Hubei, China

**View all**



**Published:** 02 December 2024



CCS '24: ACM SIGSAC Conference on Computer and Communications Security
*October 14 - 18, 2024*
*UT, Salt Lake City, USA*

**Conference Sponsors:**
SIGSAC

CCS '24: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (December 2024)
https://doi.org/10.1145/3658644.3690188
ISBN: 9798400706363

# The HitchHiker's Guide to High-Assurance System Observability Protection with Efficient Permission Switches

Chuqi Zhang*
School of Computing
National University of Singapore
Singapore, Singapore

Jun Zeng
Independent Researcher
Hangzhou, China

Yiming Zhang
Southern University of Science and Technology
Shenzhen, China
The Hong Kong Polytechnic University
Hong Kong, China

Adil Ahmad†
School of Computing and Augmented Intelligence
Arizona State University
Tempe, USA

Fengwei Zhang
Department of Computer Science and Engineering
Southern University of Science and Technology
Shenzhen, China

Hai Jin
Huazhong University of Science and Technology
Wuhan, China

Zhenkai Liang
School of Computing
National University of Singapore
Singapore, Singapore

## Abstract

Protecting system observability records (*logs*) from compromised OSs has gained significant traction in recent times, with several note-worthy approaches proposed. Unfortunately, none of the proposed approaches achieve high performance with tiny log protection delays. They also leverage risky environments for protection (*e.g.*, many use general-purpose hypervisors or TrustZone, which have large TCB and attack surfaces). HitchHiker is an attempt to rectify this problem. The system is designed to ensure (a) in-memory protection of batched logs within a short and configurable real-time deadline by efficient hardware permission switching, and (b) an end-to-end high-assurance environment built upon hardware protection primitives with debloating strategies for secure log protection, persistence, and management. Security evaluations and validations show that HitchHiker reduces log protection delay by 93.3–99.3% compared to the state-of-the-art, while reducing TCB by 9.4–26.9×. Performance evaluations show HitchHiker incurs a geometric mean of less than 6% overhead on diverse real-world programs, improving on the state-of-the-art approach by 61.9–77.5%.

## CCS Concepts

• **Security and privacy → Trusted computing**; **Operating systems security**.

*Work partly completed while being a visiting researcher at ASU with Dr. Adil Ahmad.
†Corresponding author.



*CCS '24, October 14–18, 2024, Salt Lake City, UT, USA.*

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690188

## Keywords

eBPF; System Observability; Trusted Execution Environment

**ACM Reference Format:**
Chuqi Zhang, Jun Zeng, Yiming Zhang, Adil Ahmad, Fengwei Zhang, Hai Jin, and Zhenkai Liang. 2024. The HitchHiker's Guide to High-Assurance System Observability Protection with Efficient Permission Switches. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690188

## 1 Introduction

System observability is the capability achieved by *logging* sensitive activities across different system layers, including the operating system (OS) [75], network [114], and applications [66]. In modern enterprises, logs provide invaluable insights into attacks for post-mortem investigations: a report notes that 75% of analysts find logs to be the most valuable forensic investigation asset [11].

Attackers are well aware of the importance of logs, and thus intentionally destroy or tamper with logs to hide their footprints in victim computers [12]. They typically achieve this after escalating privilege into the OS [12]—a realistic threat given the complex and buggy codebase of commodity kernels [57, 83].

To counter log tampering and ensure attack investigations, several systems have been proposed to protect logs against a compromised OS. They provide *tamper-evident* log hashes [68, 73, 88, 89, 97], or preserve both integrity and availability [39, 40, 56, 62] of the logs *before* system compromise. While these systems take significant strides towards system observability protection, they still have the following limitations (§3).

- *Risky protection environments with large TCB and attack surfaces.* In-memory protection solutions [56, 62, 97] employ general-purpose privileged components (*e.g.*, hypervisors with guest virtual machines, or TrustZone with secure OSs) to isolate host memory and manage logs. These full-fledged components maintain bloated interfaces to support their feature-rich clients (virtual machines or trusted applications), which are not directly required by log protection. As such, their redundant interfaces bring notorious attack surfaces [48, 51, 98].
- *Significant synchronization slowdowns in I/O intensive workloads.* For those in-host-memory protection systems [40, 62, 97], as soon as each log is generated, it is *copied* to the isolated privileged memory to be managed and finally persisted (onto a protected disk). Considering the high log throughput under realistic workloads within different layers of logging sources—applications, system calls, and network, our evaluation shows that this incurs up to a 52.6% slowdown. This overhead is considerable given the tight budget of production systems.
- *Large exposure protection time windows for log tampering attacks.* Alternative solutions [39, 76, 95] batch logs in the untrusted host memory, and periodically protect logs by transferring them into an external local *tamper-proof* device or remote storage. Their inherent I/O latency exposes a large protection window, during which logs remain vulnerable in host memory, with the best case being 15*ms* [39]. Our study shows that proof-of-concept (PoC) kernel exploits (not specifically designed for speed) can tamper with *all* logs within exposed windows.

In this work, we present HitchHiker. HitchHiker is an observability protection system built on two key design principles (§4) that overcomes aforementioned prior work limitations.

To ensure logs are maintained in a high-assurance in-memory environment, HitchHiker takes a first principles approach to environment design (§4.1). In particular, only components directly required for log protection are included, and large software components are *debloated* to minimize TCB and attack surface. Additionally, the task of remote log management is delegated to a *protected process* under the native OS. The combination of these strategies reduces HitchHiker's TCB by 9.4–26.9× compared to prior work, and helps build a significantly less vulnerable system.

To achieve high performance while protecting against tampering attacks, HitchHiker ensures that logs are protected in memory asynchronously but within a short, configurable real-time deadline (§4.2). In-memory protection is further sped up by using hardware memory permission switching primitives, instead of memory copies. HitchHiker achieves a protection deadline between 1.015*ms* and 100.12$\mu s$, 93.3–99.3% lower than I/O-based asynchronous systems. Under such settings, the system reduced performance overhead by 61.9 − 77.5% compared to existing work.

There are three challenges towards HitchHiker's design based on the aforementioned principles (§5). First, it is unclear how to design the high-assurance environment without significant hardware or software porting and deployment effort. Second, it is challenging to *enforce* real-time configurable protection deadlines on a system that produces logs at different layers, especially when an attacker may possess unprivileged access before OS compromise. The third is to natively delegate the log management process execution to an *untrusted* OS, with the notion of assured correctness.

HitchHiker leverages platform-available hardware protection primitives [5, 7, 36] as the building blocks to maintain its secure environment. Automatic debloating techniques [63, 104] are employed to port software components essential for log availability. Additionally, unmodified native processes with attestation philosophies [85] are adopted, minimizing software redesign efforts (§5.1).

To enforce short, configurable protection deadlines, HitchHiker unifies logs produced at different layers into its kernel buffers. There, HitchHiker leverages a precise hardware timer for log protection—using real-time memory permission switches—that is isolated from extra delays that a non-privileged attacker may induce. All protected logs are eventually, yet assuredly, persisted to disk (§5.2).

HitchHiker secures native log management process by memory view and execution context enforcement. The process's memory permissions are restricted from the untrusted OS, while system calls are secured through transparent context switch interposition. The integrity of remote tasks is further ensured by end-to-end secure channels and provisioned cryptographic secrets (§5.3).

We implemented the prototype of HitchHiker on an ARM machine (§6). Both the Stage-2 Page Table (S2PT [7]) and the Granule Protection Table (GPT [17]) are implemented as the memory permission primitive for log protection, respectively. Observability logs are generated using the extended-Aqua Tracee [2] with eBPF [9], a widely-used technique for security observability auditing [87]. We will release the artifact of our implementation prototype [13].

Using our prototype, we analyzed and evaluated HitchHiker's log protection guarantees (§7). Worst-case stress testing shows HitchHiker guarantees a 1.015*ms* to 100.13$\mu s$ protection deadline, which is 93.3% to 99.3% lower than the state-of-the-art asynchronous protection [39]. Under our most powerful tested attack, HitchHiker protected the vast majority of the log traces (≥97%) even configured with its most relaxed deadline. Importantly, all lost logs were for the last step (kernel module loading), while all logs related to exploitation and escalation were always saved.

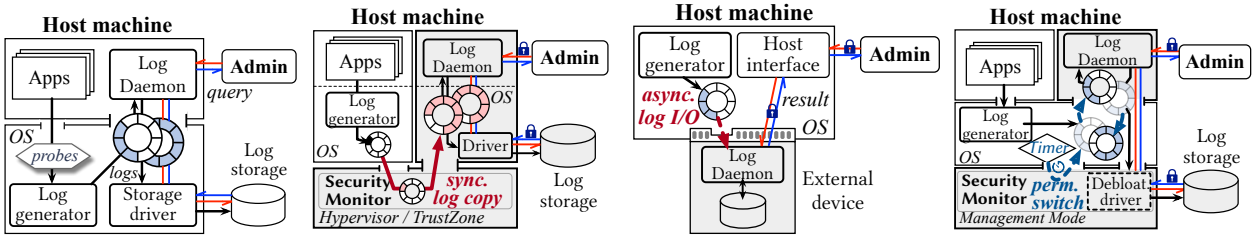
We evaluated HitchHiker's performance (§8) with both microbenchmarks and commonly used real-world programs. Experimental results show that HitchHiker introduces a geometric mean overhead of 1.8% for log-sparse and 9.9% for log-intensive programs (6% cumulative geometric mean). The overhead is up to 77.5% and 61.9% lower than the state-of-the-art protection system under log-sparse and intensive programs, respectively. This shows that HitchHiker can be readily deployed today to achieve high performance and secure log protection in enterprise computers.

## 2 Background

### 2.1 System Observability

Observability logs provide visibility into enterprise computer's *security posture*, and are captured by Security Information and Event Management (SIEM) from the industry [8, 11] and academia [54, 106]. Based on standard security practices [21], we consider three main sources (or layers) of logs: (a) *application logs* for program execution semantics [66], (b) *audit logs* for OS system call events [75], and (c) *network logs* for network traffic surveillance [54].

As common in both widely-deployed available software [2, 9, 30] and the literature [82, 91, 92], there are three main components in a conventional observability log collection stack (Fig. 1a).




(a) Conventional (observability) logging systems [2, 9, 30].
(b) Logging systems with in-memory trusted kernels [62, 97].
(c) Logging systems with external tamper-proof audit devices [39, 95].
(d) HitchHiker (this work).

**Figure 1: Overviews of (a) conventional observability systems and (b-d) observability protection systems (□: trusted components).**

First, *an in-kernel log generator* is to capture system events and generate logs. Given the kernel's privileged ability to oversee and control the entire system [9], events among different layers can be captured by various *probes* (*e.g.*, Linux Kprobe and Tracepoint). Second, *a userspace log daemon* is to manage logs. Once captured, logs are transmitted from the kernel to the user daemon to support log management tasks (*e.g.*, filter, query, and remote retrieval). Last, *a storage device* finally keeps all logs persistent locally.

## 2.2 Threat Model and Assumptions

Like recent work in log protection [39, 62, 68, 88, 89], we assume the enterprise host system is *initially benign*, but a non-privileged remote adversary starts to attack it at time $t_s$. The adversary exploits software and kernel vulnerabilities in the host system to (a) circumvent the common defenses (*e.g.*, ASLR, DEP, and stack canaries) and (b) compromise the full system at time $t_c$. Being aware of mechanisms of the system observability, the adversary attempts to tamper with the logs in kernel memory or on storage, striving to hide the attack footprints (*i.e.*, logs between $t_s$ and $t_c$) [15].

Before $t_c$, the observability log generator is honestly deployed inside the OS to capture logs. Thus, all logs are initially generated and maintained correctly. For instance, the attacker cannot trick the OS into corrupting logging mechanisms in applications before $t_c$. Post OS compromise (after $t_c$), adversaries gain full system control. Hence, logs after $t_c$ are *worthless* for forensic analysis [39, 62].

***Assumptions.*** We make standard assumptions that enterprise parties (OS vendors and administrators), computer hardware, and firmware (*e.g.*, UEFI) are trustworthy. In addition, cryptographic protocols and key management schemes work correctly. Moreover, we assume the secure boot protocol is trustworthy for authenticating system configurations, ensuring the system is always benign initially. We finally assume only *forensics-critical* logs (in §2.1) should be protected, as they are the primary targets of the adversaries. System display messages, such as *dmesg* [14], are out of scope. Regardless, the administrator can configure the observability generator to specify what logs to be protected (§6).

# 3 Motivation

## 3.1 Observability Protection

An observability protection system is deployed to shield both log integrity and availability. In general, there are two main aspects to the observability log (from any sources) protection.

**Table 1: The percentage of discovered vulnerabilities in general-purpose system components and interfaces [47, 48, 51, 98] used by prior in-memory log protection systems.**

| Hypervisor (virtual machine manager) | | | | TrustZone | | |
|---|---|---|---|---|---|---|
| VM memory manage. | CPU virtualiz. | Code emulation | Device I/O | Trusted Apps. | TZ kernel | TZ memory manage. |
| 25.7% | 21.5% | 13.2% | 9% | 33.2% | 27.8% | 17.1% |

First, a *log protection scheme* determines when to shield logs from the untrusted OS after their generation. This can be *synchronous* or *asynchronous*. In the former case, each log is secured immediately after generation (*i.e.*, blocking a syscall until its corresponding log entry is protected). In the latter case, logs are batched in the OS memory after generation and periodically protected.

Second, a *secure log environment* is required to safely hold and manage logs in a persistent manner. This environment is either a combination of a secure in-memory environment and a protected disk, or only a secure disk (*e.g.*, remote storage).

**Existing solutions.** In terms of synchronous log protection, OmniLog [62] is the state-of-the-art system (Fig. 1b). It blocks the system and synchronizes the log by copying it to a privileged host system layer, a so-called *security monitor*. The high I/O latency makes it impractical to directly store each log entry synchronously to disk. Hence, OmniLog constructs an in-memory secure environment based on its security monitor (*i.e.*, the hypervisor and ARM TrustZone). Logs are kept in this environment before being eventually written to a protected disk. Other solutions [56, 97] follow a similar approach to building a secure environment.

Alternatively, solutions [39, 95] asynchronously protect logs by using *write-once-read-many* (WORM) or custom device. Due to inherent I/O latencies, such systems are forced to use asynchronous protection. In this regard, HardLog [39] (Fig. 1c) is the state-of-the-art. It batches logs in OS memory and periodically sends them to a custom audit device, which stores logs from the host. Such audit devices also support secure log management tasks, like remote retrieval. Besides that, some asynchronous systems also batch and transmit logs to a network storage server [76, 102].

## 3.2 Limitations of Current Protection Solutions

**L1: Risky environments with large TCB and attack surface.** Prior work [56, 62, 97] adopts the hypervisor or TrustZone (TZ) to

**Table 2: Multi-layer observability log throughput, log capture overhead, and synchronous log protection [62] overhead. Audit Log: system call logs [30]; App. Log: application logs [66]; Net. Log: network event logs [101].**

| Program | #Multi-layer Observability Throughput[1] | | | | Overhead (%) | |
|---|---|---|---|---|---|---|
| | Audit Log | App. Log | Net. Log | Total[2] | Native-OBS | OmniLog |
| **Nginx** | 53,038 | 25,144 | 594 | 78,776 | 4.2% | **15.4%** |
| **Redis** | 40,857 | 152 | 37,528 | 78,537 | 6.4% | **52.6%** |
| **MySQL** | 76,228 | 5,151 | 20,583 | 101,962 | 13.8% | **27%** |

[1] Throughput (logs/second) is counted by the conventional log generator (**Native-OBS**; §6) under the real-world workload detailed in Tab. 6 (§8.2).

[2] Each log entry is around 256 bytes on average size.

protect logs in memory. However, their bloated interfaces enlarge the vulnerabilities within the secure environment.

Logs are kept and managed within the trusted kernels, such as feature-rich virtual machines or the TZ secure kernels (*e.g.*, OP-TEE [22]). Supporting such environments involves complex interfaces, such as CPU virtualization, code emulation, and TZ management. Unfortunately, those full-fledged general-purpose interfaces enlarge the vulnerabilities [45, 47, 98] of the secure environment. For instance, a single vulnerability in the TZ driver interface results in the entire secure world compromise [48].

Tab. 1 then concretely concludes the discovered vulnerabilities within general hypervisor and TZ interfaces. Considering the severe vulnerabilities in the hypervisor, 72.07% and 80.81% of the common vulnerabilities and exposures (CVEs) in KVM [93] and Xen [33] can lead to host exploitations, respectively [51]. Meanwhile, more than 65% of CVEs in TZ kernel-supported systems [48] have severe ($\geq 7$) common vulnerability scoring system (CVSS [32]) scores.

**L2: Slow synchronous protection in log intensive scenarios.** Modern computers can produce lots of observability logs when running realistic I/O-intensive real-world programs. Under such scenarios, synchronously protecting every produced log, even in memory, imposes a notable slowdown.

To concretely show the slowdown, we synchronously protected logs produced by three high-performance programs: web server (Nginx), in-memory key-value store (Redis), and on-disk relational database (MySQL). Programs are tested with default configurations and benchmark settings of *apachebench* [1], *memtier_bench* [20], and *sysbench* [27], respectively. We employed the synchronous protection of OmniLog [62] (detailed setup can be found in §8).

Tab. 2 illustrates the performance impact incurred by OmniLog and the conventional observability generator (Native-OBS), which stores logs within the kernel's memory, against native execution. We observe that OmniLog's synchronous protection incurs 0.9 – 7.2× more overhead than Native-OBS. Due to the high throughput of observability logs, synchronous protection for each log requires frequent system interruption, involving context switches to the privileged layer (*e.g.*, changing the protection ring in x86 or the exception level in ARM), and copying the log to the protected memory inside a privileged layer, which is time-consuming.

**L3: Insecure I/O-based asynchronous log protection window.** Current asynchronous protection systems [39, 88, 89, 95] expose a long time window before logs are protected to remote or local *tamper-proof* storage through slow I/O operations. For instance, the state-of-the-art, HardLog [39], provides a lengthy 15*ms* delay. Unfortunately, adversaries can exploit kernel vulnerabilities and obtain root access within very few syscalls [23], each of which only requires thousands of cycles, and tamper with in-kernel logs.

To understand the tampering problem, consider the DirtyPipe [74] attack due to improper preservation of permissions. Given a flaw of a simple improper `PIPE_BUF_FLAG_CAN_MERGE` flag in kernel, the vulnerability offers adversaries the capability to quickly rewrite arbitrary files (*e.g.*, `/etc/passwd`) to escalate privileges. We reproduced its PoC exploit to load a kernel module to delete the logs. We found that it took within around 12*ms* (detailed case study in §7.3), resulting in scenarios where *all* attack traces were lost in 15*ms* windows. This exploit arises from a semantic bug. Unlike common memory corruption vulnerabilities, exploiting such bugs does not require slow object manipulation or heap spraying [83], making them brutal and "efficient" for adversaries.

Please note that this simple example does not show that a 15*ms* protection delay is insufficient to prevent logs from being tampered with in the majority of kernel attacks. However, it does indicate that PoC exploits (which are not designed to be fast) for common attacks are able to allow log tampering within a 15*ms* protection window. This underscores the necessity of reducing the protection window to substantially increase the difficulty of such attacks.

## 4 HitchHiker Overview

HitchHiker is an efficient and high-assurance observability protection system. This section describes HitchHiker's two key design principles (§4.1-§4.2) and its system deployment model (§4.3). Given the rising use of ARM-based computers in both enterprise machines and cloud computing infrastructures, we use ARM as the reference architecture of this paper. We further discuss the efforts of porting HitchHiker to support other architectures in Appendix §D [110].

### 4.1 High Assurance Log Environment Design

To overcome **L1**, HitchHiker protects logs in a trusted in-memory environment that is redesiged from the ground-up by leveraging three high-assurance strategies discussed in this section.

Fig. 1d shows HitchHiker's trusted environment. It contains a security monitor (**HitchHiker Monitor or HkM**) and a protected log daemon (**HitchHiker Daemon or HkD**). The monitor is required to partition and configure system resources to achieve isolation. It can be implemented in any higher privileged layer than the OS, *i.e.*, both the hypervisor mode (EL2 in ARM) and system management mode (EL3 in ARM). However, HitchHiker implements the monitor as an EL3 runtime service [28, 37]. This is because EL3 can be kept small (and privileged) even in enterprise computers that want to run virtual machines [3], since it would not have to include the hypervisor's TCB.

The first strategy is to implement HkM with only *directly required* components for log protection, namely, control over memory and device protection primitives. Memory protection is required to isolate trusted components (HkM and HkD) from malicious CPU access by the OS. Device protection is required to (a) reserve a storage disk for log availability and (b) prevent malicious attacks from untrusted device access (i.e., DMA attacks). HkM controls EL3-supported memory protection primitives for memory protection, while it leverages the System Memory Management Unit

**Table 3: Cost of different memory protection mechanisms.**

| Primitive | #CPU cycle[1] cost per log buffer size (less means better) | | | | | | |
|---|---|---|---|---|---|---|---|
| | 64B | 256B | 512B | 4KB | 16KB | 32KB | 64KB |
| EL3-memcpy | 1,785 | 2,783 | 4,680 | 24,672 | 98,978 | 198,776 | 400,903 |
| S2PT | 4,482[2] | 4,482[2] | 4,482[2] | 4,482 | 4,496 | 4,516 | 4,636 |
| GPT | 4,383[2] | 4,383[2] | 4,383[2] | 4,383 | 4,410 | 4,398 | 4,490 |

[1] CPU runs at 1.2GHz. The cost of the Secure Monitor Call (SMC) context switch round-trip is included. Detailed system configuration is illustrated in §6.

[2] S2PT and GPT's protection granularities are at the page level. Protecting small-sized buffers will remain at the same cost as protecting a page (4KB).

(SMMU) [62, 100] for device protection. The options of memory protection primitives are explained in §4.2, and we describe all implementation details regarding these features in §6.

The second strategy is to avoid large software components within the security monitor. Specifically, a storage device driver is required to persist logs. However, porting a commodity driver software into HkM is undesirable—a simple Linux SATA driver has more than 10k lines of code and complex kernel dependencies [63]. Thus, HitchHiker implements a secure, minimized version (§5.1.2).

The third strategy is to ensure log management tasks (e.g., remote retrieval) through a *protected daemon process* in the untrusted OS. This avoids including trusted kernels like Linux and OP-TEE [22] used by prior work [62, 97] into the system's TCB. While a compromised OS may prevent the scheduling of the log daemon, it is unable to alter the integrity or availability of logs, nor the integrity of log management tasks such as retrieval (§5.1.3).

## 4.2 Deadline-Enforced Log Permission Switch

To overcome **L2-L3**, HitchHiker protects logs *in-memory* within short real-time deadlines set by administrators by using efficient hardware permission switches.

In-memory protection inherently provides tighter deadlines than I/O-based protection [39], which suffers from large device latencies and interference from other devices. Naively, at configured deadlines, logs may be copied from the OS to the protected memory [62]. However, short deadlines required for security result in frequent copying. This causes non-negligible protection overheads in scenarios where logs are frequently produced (*e.g.*, up to 32% runtime performance overhead as we show in §8.2).

*Switching* log buffer permissions using hardware permission primitives can mitigate large copy overheads. To illustrate this, Tab. 3 compares copying different sizes of log buffers to the privileged monitor layer (ARM's management mode or EL3 [3] as used by prior work [62, 97]), against protecting them by (one of) hardware permission primitives[1] supported by HitchHiker:

- Stage-2 Page Table (S2PT), enabled by the virtualization extension, offers a second level of MMU translation which is transparent to the OS. It supports page-level memory access control by configuring the related table entry's permission bits [7].
- Granule Protection Table (GPT) is an in-memory table structure from Real Management Extension (RME) of Confidential Computing Architecture (CCA). It defines page-level physical memory's accessible CPU states during MMU translation [36].

[1]TrustZone Address Space Controller (TZASC [4]) is an alternative primitive. However, its bus-transaction-level protection granularity lacks flexibility. See Appendix §D [110] for TZASC discussion and features on other platforms.

We find that switching permissions on log buffers (over 4KB) proves more efficient than log copying. Copying is bound by the size of costly memory-intensive operations that are only efficient for smaller buffers (less than 512B). Hardware permission switching maintains stable costs by (a) modifying permission control registers or data structures to alter hardware permissions and (b) invalidating stale entries in the translation-lookaside buffer (TLB). This requires significantly less CPU and memory operations.

## 4.3 System Deployment Model

We consider the same enterprise deployment model for HitchHiker as existing research [62]. Specifically, we consider that the enterprise OS vendor (*e.g.*, Microsoft, Red Hat Enterprise Linux) will instantiate the monitor (HkM) during system boot. There is a clear example of this in the past. Microsoft Windows 11 instantiates a hypervisor-level monitor during boot-up (for kernel memory and integrity protection), and also enforces firmware-level (UEFI) extensions to support this monitor [31]. We also consider that the OS vendor will modify the logging facility to invoke HkM for log protection (§5.2) and management. The latter include minor changes to retrieve logs remotely using the protected log daemon (HkD) and locally from audit system tools. Finally, for remote retrieval (§5.3), we assume that the enterprise system administrator will provision cryptographic keys, as is common practice for other IT administration standards like Intel's Active Management Technology (AMT) [10].

# 5 HitchHiker Design

This section describes the main challenges towards HitchHiker's design, and the following sections describe how we address them.

**C1: Reducing software porting and deployment efforts.** Given strict requirements to ensure high assurance, it is challenging to design an infrastructure where these strategies are satisfied without significant deployment effort (*e.g.*, completely rebuilt storage drivers for persistence or adapting secure log daemon).

**C2: Enforcing short multi-layer log protection deadlines.** Unlike prior work that only deals with OS logs, protecting logs from different layers, including applications and network logs, and under the (at least unprivileged) access of an adversary, makes it challenging to enforce uniform and very short deadlines.

**C3: Delegating log management to untrusted OS securely.** Any task naively delegated to the OS becomes untrustworthy after its compromise. Since it is agnostic to administrators when the OS is compromised, the OS can provide completely wrong responses after compromise even without tampering with logs.

## 5.1 Secure Environment Software Stack

HitchHiker's security monitor (HkM) configures protection primitives to create isolated protection domains (§5.1.1). Within the isolated protection domain, HitchHiker synthesizes a debloated log storage device driver into HkM during offline stages (§5.1.2). Besides the debloated log driver, the monitor maintains a *process-based* log daemon enclave in the isolated domain by execution state and context switch protection (§5.1.3).

*5.1.1* ***Protection domain bootstrap and enforcement.*** During initialization, HkM is loaded using secure boot (§6). It establishes

two protection domains. For each domain, HKM defines specific memory view permissions (Fig. 2-left).

- *OS domain.* Untrusted components (i.e., OS and applications) are executing within this domain. They can only access their own memory, as well as shared memory and unprotected log buffers (which are dynamically protected, explained in next paragraphs).
- *Protected domain.* Trusted components (i.e., HKD and HKM) are executing within this domain. HKD contains two threads (whose role is explained in §5.2) and it is allowed to access all log buffers. HKM manages its own code and data, as well as the device driver interface (§5.1.2) and HKD's protected execution states and page table (§5.1.3) within the protected domain as well.

Depending on the hardware protection primitive, each domain's memory view is enforced by *maintaining separate S2PTs or GPTs*. In particular, GPT configuration is as follows. In the OS domain's GPT, only untrusted components are marked as *normal-world accessible* [36]. HKD memory region is marked as *none-accessible*, while HKM is marked as *root-world-assessible* (EL3-exclusive). When switching to the protected domain's GPT, HKD memory is marked as *normal-world accessible*. Such a design is agnostic to the Realm world [17] and, therefore, does not interfere with GPT's original functionality (i.e., supporting confidential VMs in Realm world [35]). Configurations of S2PT are similar to GPTs—the OS domain's S2PT only grants access to the untrusted components, while the other can access trusted components [69].

With enforced memory views, HKM dynamically protects log buffers by removing the OS domain GPT or S2PT's access permission to buffers. CPU cores that execute in the OS domain share GPT (or S2PT) entries in TLB [36]. As such, a synchronized memory view is maintained for the OS domain, even though different cores may concurrently protect log buffers (e.g., by requesting HKM to change the OS domain's GPT entries) at runtime (§5.2).

To switch between domains (e.g., to execute HKD or protect log buffers), the OS executes a Secure Monitor Call (SMC [25]) to HKM. At such calls, HKM validates the request and switches the GPT/S2PT to perform the requested functionality (e.g., log protection). A detailed list of SMC interfaces implemented by HitchHiker and their validations can be found in Tab. 11 under Appendix §B in [110].

Note that, in addition to GPT/S2PT, HKM leverages System Memory Management Unit (SMMU) to prevent DMA attacks against the protected domain. The SMMU limits the addressable physical memory ranges of a device [100]. SMMU configuration interfaces (memory-mapped I/O interfaces and stream tables) are only accessible by HKM within the protected domain (by aforementioned memory permission restrictions).

*5.1.2* ***Debloated driver synthesis and protection.*** HitchHiker synthesizes a debloated driver prior to system bootstrap, which is later instantiated into HKM during initialization. Driver debloating is achieved by record-and-replay mechanisms [63, 104] for storage drives (*e.g.*, SATA, USB, etc), as a one-time effort.

Considering a block storage device as a finite state machine, driver operations are distilled into a sequence of *driver-device* interactions, each of which is a state transition. These interactions remain consistent irrespective of variations in I/O workloads [63]. During recording, sample I/O jobs are issued to log the storage to templatize the following interactions (detailed in §6): (1) the sequences and content of Memory-mapped I/O (MMIO) write operations to the log storage, (2) the allocation of direct-memory-access (DMA) buffers for the log storage, and (3) interrupt request (IRQ) numbers invoked by the log storage, along with the completion status. The template is then installed into HKM.

HKM sets up and protects the driver from the template during initialization to enable communication with the log storage disk. In particular, the driver's MMIO regions (which are fixed addresses) are only mapped to the protected domain. Similarly, DMA buffers to hold I/O payloads are pre-allocated in the protected domain.

Thus, the untrusted OS cannot manipulate the driver interface. To ensure interrupts from the log storage disk are routed to the protected driver only, HKM updates the IRQ table in the EL3 interrupt management framework [16]. At runtime, whenever a log persistence or deletion command is issued, HKM replays the template with dynamic parameters (*e.g.*, MMIO control sequences with payloads filled in the DMA buffers), to directly read or write the corresponding log disk sectors.

*5.1.3* ***Protected userspace daemon execution.*** HitchHiker enables loading an unmodified program into a protected userspace process (or enclave) as HKD. This allows administrators to build log management functionality easily. This section describes how HKD's memory and execution state are protected in the protected domain with secure context switches. §5.3 explains how log management tasks are achieved using HKD.

**Memory and execution state protection.** HKD (including its memory and execution stack) is loaded into the protected domain, in a manner inspired by SGX [85] and recent research [41, 69, 70, 112]. Specifically, during machine provisioning, the binary's SHA-256 integrity hash is pre-installed onto protected storage. During system boot, the OS allocates its process context, loads the binary within a reserved region, and requests HKM to attest to its initial integrity (*i.e.*, via the hash). To prevent *time-of-check-to-time-of-use* attacks, the monitor protects the process region (by the aforementioned protection domain) before attestation.

After attestation, HKD's pages are *pinned* to memory (*i.e.*, no swapping). The monitor then copies the page tables of the process into its own memory region. As HKD's page table is maintained inside the monitor's memory, the OS cannot make any changes to verified mappings, thereby preventing page remapping attacks [70]. Pinning is acceptable since HKD would typically require a small amount of memory (less than 8MB in our implementation).

HKD is initialized with two threads: a consumer thread and a manager thread. The former thread consumes protected logs, while the latter issues log persistence operations (§5.2) and securely interacts with remote tasks (§5.3).

**Secure context switch handling.** Like any user process or SGX enclaves [85], the task scheduling of HKD is controlled by the untrusted OS. However, the OS cannot directly transition into HKD's context, as it is in the protected domain. Therefore, whenever the OS wants to schedule the threads of HKD, it calls the monitor using SMC (§5.1.1). In such calls, HKM restores HKD's context state and memory permission view.

Once scheduled, HKD cannot exit directly to the OS for system calls or interrupts due to protection domain enforcement. To *securely and transparently* support HKD exits, HKM enables a custom

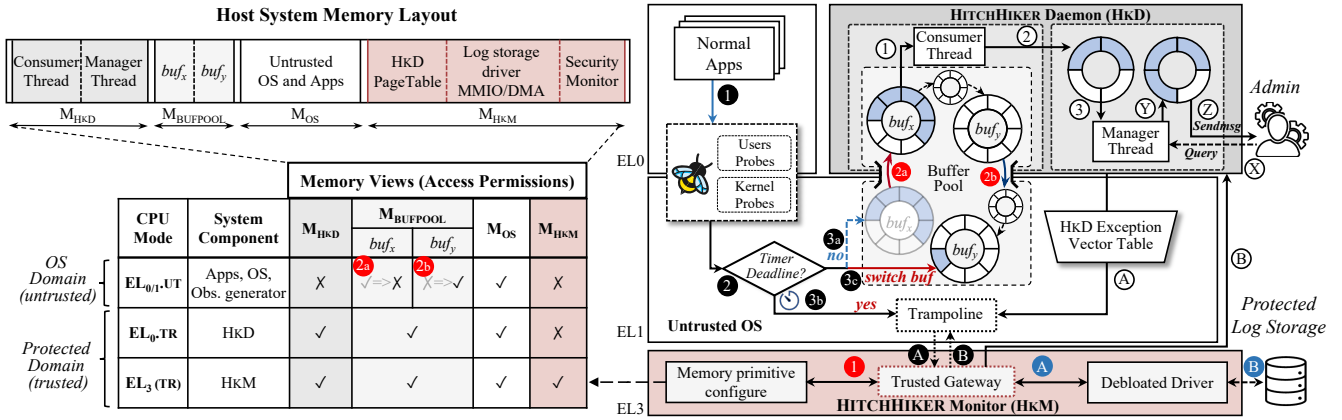



**Figure 2: HitchHiker memory layout and permission views (left) and workflow (right). The OS and applications (□) are inside the OS domain (UT). HitchHiker *Monitor* (□ HkM) and *Daemon* (□ HkD) are inside the protected domain (TR). Memory views of two domains are maintained by their separated hardware primitive configurations, respectively (i.e., two S2PTs/GPTs, §5.1).**

exception vector table when executing HkD, in which all exceptions are interposed by a trampoline SMC (Fig. 2, Ⓐ–Ⓐ). Thus, at HkD exits, the monitor saves HkD's context state before exiting to the OS (Ⓑ). Whenever the OS calls the monitor to reschedule HkD, the context is restored (Ⓑ). HkM ensures that system call results are sanitized to prevent IAGO attacks [49] (*e.g.*, OS returns a malicious pointer). We explain how network system calls are sanitized in §5.3. In the future, if other system calls, such as file system support, are needed by the program, a TCB-conscious library operating system (LibOS) [99] can be integrated with HkD.

## 5.2 Deadline-Enforced Unified Log Protection

HitchHiker *enforces* a short log protection deadline by employing a periodic hardware timer by the OS. The administrator configures a real-time protection delay of $T_p + \epsilon$, where $T_p$ is the timer interrupt frequency and $\epsilon$ is the latency incurred while protecting logs in memory. We found that modern hardware timers have a high resolution [104]. Therefore, $T_p$ can be set as a tight log protection time interval (*e.g.*, $100\mu s$–$1ms$).

This section first explains how different logs are collected in a unified location to standardize the protection window. Then, it explains how the timer ensures logs are protected in memory within a set deadline. Finally, it describes how protected logs are securely persisted in the background to guarantee eventual availability.

*5.2.1* ***Unified log collection in kernel buffer pools.*** Capturing logs at different locations (*e.g.*, network drivers, applications) in an ad-hoc manner to protect them can increase protection delay. HitchHiker avoids this by unified capturing logs within the kernel before protection (❶–❷ in Fig. 2). It requires two aspects for unification: (a) a central region to hold all logs and (b) a mechanism to generate the captured events as logs to that region.

To create a central region, HitchHiker allocates a *pool of per-core* buffers within the kernel observability generator's memory. For each kernel thread, logs are always written to the *current* per-core buffer within the pool, thereby avoiding cross-core contention as multiple kernel threads can write logs simultaneously.

A per-core pool of (at least two) buffers is required, because HitchHiker dynamically switches the *current* buffer's permission under the OS's domain for protection (explained in §5.2.2). Hence, there should always be an available buffer as the *current* to the OS. The size of the buffers is administrator-configured, and the buffers are pre-allocated during system initialization. If the buffer becomes full, very rare in honest scenarios based on proper configuration in our evaluation (§8) and prior work [62], the generator waits for a buffer to be consumed and returned. This prevents attackers from overwriting their attack trace even if they generate many spurious logs from userspace [89] before full system compromise.

HitchHiker captures the logs (❶) by extended Berkeley Packet Filter (eBPF [9]), a widely-used generator [87? ] in prior work (see implementation details in §6). A challenge in capturing *application logs* is locating and automatically intercepting the routines where logs are generated. HitchHiker addresses this by using contextual tracking of file system calls. In particular, applications leverage the OS file system to write logs. Thus, the generator records *write* syscalls and extracts file paths in kernel structures (*e.g.*, `dentry` in `vfsmount`) to match them with administrator-defined log file paths (*e.g.*, `/var/log`) [2]. For application log file writes, HitchHiker extracts the log string using an eBPF helper `bpf_probe_read_user`. For logs from network and system calls, HitchHiker injects BPF programs into kernel tracepoints to attach all system calls, task scheduling, and socket-related events. HitchHiker follows Linux Auditd's format [30] to generate system call log entries, which include parameters, return values, and other fields (*e.g.*, process identifiers and timestamps). For network traffic logging, HitchHiker logs source and destination IPs, ports, and other packet data from the kernel socket structure of network traffic.

*5.2.2* ***Controlled hardware timer-based buffer protection.*** The OS deploys a hardware timer based on configured frequency $T_p$ to direct a kernel thread to initiate log protection. As the OS is initially benign, it will correctly deploy the timer before compromise. To ensure the short protection window, HitchHiker *controls* the delay after the timer expires (each $T_p$) to ensure that protection

is achieved in an isolated small period ($\epsilon$). All these aspects are explained in the following paragraphs.

Once logs are captured, HitchHiker writes them to the *current* buffer (Fig. 2, ❸ₐ). When a timer deadline is reached, a kernel thread calls HKM (❸ᵦ–Ⓐ) to temporarily switch the OS access permission to the current buffer, marking the buffer as inaccessible to the OS domain (❶–❷ₐ). After protection is complete and execution returns to the OS, a new buffer is fetched from the pool to be the *current* buffer (Ⓑ–❸꜀). The previous (now protected) buffer will be re-inserted into the pool after consumption (§5.2.3).

To correctly capture logs and protect them by constant timer preemptions, both log buffer writing and protection are executed in *atomic contexts*. Otherwise, the preemptions between log writings and protections will not only disrupt the deadline but cause faults (*e.g.*, by writing logs to a just-protected buffer before fetching a new current buffer). To this end, HitchHiker enforces each core to acquire a lock before every log buffer write. On timer interruption, the timer thread acquires those locks before protection. Since the lock is released after every log buffer write, this timer thread acquires it within a tiny delay (less than a hundred CPU cycles on average in our experiments).

HitchHiker forcibly isolates $\epsilon$ (the time delay between the timer thread invocation and log protection finished by HKM) to a small value. Otherwise, adversaries may significantly extend $\epsilon$ to corrupt the protection deadline. This property is regulated by (a) prioritizing the timer interrupt kernel thread and (b) enforcing the aforementioned atomic execution of the log protection procedure. The former (interrupt prioritization) prevents the timer thread invocation from being delayed by other interrupts (*e.g.*, by raising spurious interrupts [77]). The latter (atomic permission switching execution) ensures the permission switch procedure execution cannot be extended by other task preemptions or IRQs.

*5.2.3* ***Fast buffer consumption with eventual persistence.*** HKD consumes the protected buffer and then returns it into the kernel buffer pool. It consumes buffers efficiently using multi-threading and *real-time* techniques [39]. Since HKD might not be scheduled after OS compromise, HKM ensures all remnant buffers (in the protected domain) are persisted before shutdown events.

HKD separates its threads (§5.1.3) to the consumer thread for log consumption and the manager thread for log persistence and admin task response. By doing so, heavy I/O and network jobs are dispatched to the manager thread. In addition, the consumer thread is prioritized by (an honest) OS over other threads. Moreover, all hardware interrupts, except for the protection scheme timer, are disabled by the OS during the thread's execution.

HKD requests the monitor to persist the consumed logs through the debloated driver. Since HKD is at userspace, it cannot directly issue I/O operations. Hence, the manager thread executes an *ioctl* system call as a proxy, which then invokes SMC to request HKM. This is just a notification mechanism, all message passing between HKM and HKD is done using a reserved *communication buffer* within HKD's memory. When there are no logs, HKD goes to sleep.

The monitor intercepts power management operations to ensure all protected logs are persisted. In particular, it intercepts the Power State Coordination Interface (PSCI) and issues an I/O command to persist all protected logs forcibly before power events [39, 62].

Fig. 2 illustrates the procedure. The consumer thread consumes logs in the protected buffer (①–②), and asks the monitor (Ⓐ–Ⓐ) to edit the OS domain's permission to return the buffer (❶–❷ᵦ). Meanwhile, the manager thread requests HKM to persist them into secure storage via the debloated driver interface (Ⓐ–Ⓑ). Finally, before power events, HKM issues an I/O command via its debloated driver to force all protected logs to be persisted to the storage.

## 5.3 Secure Delegated Remote Log Management

HitchHiker empowers HKD to securely support remote log management tasks, such as retrieval, prior to OS compromise. After the OS compromise, the integrity of any completed management task is ensured, and the OS can only prevent new task completion—until administrators recover it—by not scheduling HKD. To achieve such guarantees, HKD sets up a secure channel to a remote administrator, using provisioned cryptographic secrets, over a sanitized OS-delegated network transport.

*5.3.1* ***Daemon secret establishment.*** At machine provisioning (§4.3), the IT administrator installs the log daemon HKD, and sets up cryptographic secrets for runtime secure communication.

To ensure secure communication, the administrator provisions a key pair. The secret portion of the key is installed on the protected disk, while the public portion is kept by administrators for later authentications. Besides that, admins also install their own public key into the protected disk to ensure a *two-way* authenticated and secure channel can be created. Since the protected storage disk can only be accessed by the monitor, during HKD loading, HKM securely installs the provisioned keys within a reserved region of HKD. Thus, the key remains protected from the untrusted OS.

*5.3.2* ***Secure channel on sanitized delegated network.*** While HitchHiker sends and receives packets through OS network interfaces, it ensures packets are secured end-to-end between HKD and the administrator. This is ensured by the Transport Layer Security (TLS) protocol and the key pair provisioned. Therefore, adversaries cannot impersonate the remote admin (*e.g.*, forge and issue log deletion requests and responses). The remaining paragraphs illustrate how HKD network system calls are supported and sanitized.

HKM interposes HKD's network service system call invocations and returns (*i.e.*, context switches) to perform secure domain interpositions. The interposition is enforced through HKD's specialized exception vector table, which is transparent to the OS (§5.1.3). As such, HKD's execution context is securely saved and managed by HKM. At syscall invocations, before restricting the memory permission view to enter the OS, syscall pointer parameters are redirected to a shared buffer that is accessible in the OS. Otherwise, the OS (within in restricted permission view) cannot access HKD's memory buffer pointed by the syscall parameters.

Once the syscall returns, the monitor sanitizes the return values before resuming HKD. In particular, to prevent pointer-based IAGO attacks [49], HKM ensures that no pointer is returned for a region that belongs to HKD's domain. Moreover, HKM copies the packet data from the shared buffer into HKD's buffer. Then, HKM transits to the protected domain (enables HKD's memory permission view), restores the context, and enforces its maintained HKD page and vector tables. Finally, HKM transits the CPU execution to HKD.

**Table 4: HitchHiker component implementation breaks down in source lines of code (LoC).**

| Component | Base | SLoC |
|---|---|---|
| **HitchHiker Monitor (HKM)** | | **2.7K in total** |
| Hardware permission - GPT configuration | TF-A [37] | 399 |
| Hardware permission - S2PT configuration | TF-A [37] | 738 |
| Debloated driver | Sata_sil24 [24] | 566 |
| HKD context interposition | TF-A [37] | 1,106 |
| **Operating System** | | **3K in total** |
| Observability generator | Tracee [2], Libbpf [18] | 860 |
| Timer and buffer pool management | Linux kernel module | 1,618 |
| HKD binary loader | Linux kernel module | 584 |
| **HitchHiker Daemon (HKD)** | | **0.8K in total** |
| Consumer thread | – | 208 |
| Manager thread | – | 622 |

Fig. 2 (Ⓧ–Ⓩ, Ⓐ–Ⓑ, ❶–❷) illustrates the interposed context switch. Whenever HKD requests network services (Ⓧ, Ⓩ), it traps to HKM (Ⓐ–❶) by its customized exception vector table. HKM then saves its context, redirects pointer parameters, transits to the OS domain (❶), and then exits to the OS (❷). Whenever the OS finishes the service, HKM interposes the return (❶) and performs the context switch on the OS' behalf. It restores the saved contexts, enables HKD's domain (❶), enforces its page and vector table, and directly returns to HKD's execution context (Ⓑ).

## 6 Implementation

We prototyped HitchHiker for an ARM-Juno R2 board featuring six CPU cores (a dual-core Cortex-A72 and a quad-core Cortex-A53) alongside 8GB SDRAM. A 256GB SATA WD BLUE SSD was connected to the board for secure log storage, and a 1TB Sandisk Extreme Pro SSD was used by the OS for normal filesystem storage. The board runs Linux 5.3 as the OS, using Trusted-Firmware-A (TF-A) arm_cca_v0.3 [37] at EL3 as the base of HKM. For the GPT implementation, there is no commercial hardware that supports Realm Management Extension (RME). To validate the functionalities of HitchHiker, we developed a functional prototype on ARM Fixed Virtual Platform (FVP) with RME support. To further emulate the performance of GPT on our board, we followed common practices (described in the following paragraphs). Tab. 4 offers a breakdown of source code lines modified or introduced.

**HKM and protection domain enforcement.** HKM's bootloader images are burnt into the *secure boot region* (trusted boot ROM and SRAM), which is not accessible to the normal OS once boot. During boot, the host memory is partitioned, specified by TF-A (`bl_regions`) and device tree. Contiguous physical memory regions are reserved for HKD and the log buffer pool by specifying the Linux contiguous memory allocator [79]. All the partitions (their ranges) can be easily adjusted and configured by administrators.

We reserve memory in TF-A to maintain two GPT/S2PT table instances. For S2PT implementation, we set up `HCR_EL2.VM` during boot to enable the second-level address translation. All S2PT control interfaces (`VTCR_`, `VTTBR_EL2`) are directly configured by TF-A.

We emulate GPT's performance by following common practice [81, 100, 112]. First, we emulate the GPT control interface cost by substituting all control registers (`GPCCR_`, `GPTBR_EL3`) with idle EL3 ones. Second, we program GPT table structures per GPT specifications [36] by using TF-A official code. Third, we flush the entire TLB entries after every GPT configuration change. Even if the cost of hardware GPT checks (expected to exhibit good caching behavior) is not included, this *will not affect the relative overhead* since GPT checks are applied to the full system [81].

*Limitation.* Our prototype currently lacks support for SMMU interface protection against untrusted hypervisors; however, it does not impact our evaluation. Implementing it would involve extending two SMC interfaces within HKM, and modifying the SMMU driver with approximately 0.6k LoC as indicated by prior work [100].

**Debloated driver synthesis.** To synthesize a debloated driver offline, we used the default SATA driver (`sata_sil24`) [24] on our system for recording. In principle, any driver can be used. The recording is performed by sample I/O that writes directly to the log device (*e.g.*, `/dev/sda`), with flags `O_DIRECT | O_SYNC` to bypass the file system (*e.g.*, journaling) and block abstractions. To record MMIO and DMA operations, our recorder instruments the lowest kernel functions (`readl`, `writel`, and `dma_alloc`) before I/O. During debloated driver setup in HKM, its device IRQ is isolated from the OS (`plat_ic_set_interrupt_type`) and routed to HKM by setting the `SCR_EL3.FIQ` bit in TF-A (§5.1.2).

**HKD and timer interrupt prioritization.** HKD's initialization is supported by a customized binary loader (in a kernel module). Its thread execution stacks and specialized exception vector table are reserved during HKM setup (§5.1.3). To prioritize the protection timer interrupt over others, we utilized TF-A's interrupt management framework (the function `plat_ic_set_interrupt_priority`). In particular, we programmed the Interrupt Priority GIC MMIO registers (`GICD_IPRIORITYR`) to grant the timer interrupt with the highest priority (GIC_HIGHEST_NS_PRIORITY) in the normal world. Note that the kernel (at EL1) can also access this register. Currently, we do not protect attackers from tricking the vulnerable kernel into changing the timer configuration. We discuss such attacks and techniques to enable full timer protection in Appendix §D [110].

**Observability generator.** The generator is built and extended upon Aqua Tracee [2], a widely deployed eBPF-based forensics tool [87]. We extended the Linux BPF helpers with a new one named `bpf_write_current_buf` and integrated it into libbpf. The helper's underlying kernel function unifies log generation into our *current* kernel buffers (§5.2.1). Deploying and verifying such an extended BPF helper is straightforward and aligns with common practices [82], requiring ~50 lines of change for the BPF verifier.

## 7 Security Evaluation

### 7.1 Security Claims and Analysis

We analyze HitchHiker's log integrity and availability guarantees through a sequence of security claims (**S1**–**S5**).

**S1.** *All observability logs before OS compromise are captured.*

Prior to system compromise, the log generator operating within the OS faithfully captures all (OS, application, and network) logs within the *current* kernel log buffer. The administrator configures the buffer pool to allocate enough space (§8) for logging. If buffers are exhausted within the pool, the generator will pause the event and wait for a buffer to be consumed by HKD (§5.2.1). Hence, the attacker cannot prevent events from being logged, and neither can they flood the buffers with numerous spurious event logs to force the system into dropping previously-logged entries [89].

**S2.** *All collected logs will be protected in memory within the configured, short real-time deadline ($T_p + \epsilon$).*

The OS, which is initially benign, faithfully configures a hardware timer based on the frequency ($T_p$) specified by the administrator. This cannot be modified by the attacker until they compromise the OS. Once a timer interrupt is fired, a kernel thread calls HKM (using an SMC) to protect logs. The delay between when the interrupt is fired and logs are protected ($\epsilon$) is kept small and regulated because (a) the buffer permission switch procedure being executed in an atomic context with preemption and IRQ disabled, (b) the timer interrupt handling is always prioritized (§5.2.2), and (c) buffer permission switch is fast. As such, adversaries cannot prolong either $T_p$ or $\epsilon$ via launching malicious interrupts [77] or flooding the buffer with numerate events [89] (further validated in §7.2).

**S3.** *All protected logs in memory cannot be accessed by the OS.*

Once the log protection SMC is handled by HKM (*i.e.*, at $T_p + \epsilon$), the log buffer's memory access permission under the OS domain is removed. The log buffer is also protected by SMMU; hence, device-based DMA attacks are prevented. HKM uses the same mechanisms—memory permission and SMMU—to protect itself and the data structures of SMMU from the OS (§6). Hence, any malicious access to the protected domain is prevented.

**S4.** *All protected logs in memory will be persisted in the secure log storage disk before system shutdown.*

HKD's consumer thread, which is securely loaded (§5.1.3), asynchronously issues I/O commands through HKM's debloated driver interface to persist protected logs (§5.2.3). Prior to compromise, the OS will schedule the consumer thread to ensure logs are persisted. After compromise, the OS cannot modify HKD's operations due to domain enforcement, and it can only refuse to schedule the thread. If this happens, the logs will remain in protected memory until a system shutdown event is triggered by the OS. Assuming no power or hardware failures (which cannot be controlled by our attacker), HKM will intercept all shutdown events and leverage its protected driver to persist all remnant in-memory logs to disk.

**S5.** *All stored logs will only be deleted (e.g., after retrieval to remote storage) by the system administrator.*

A compromised OS cannot directly send log deletion commands to the protected disk, since the disk is isolated, and can only be accessed by the debloated driver in HKM (§5.1.2). HKM only accepts log deletion requests from HKD, which will only send such a request if it receives a command from the system administrator. In particular, HKD and the system administrator leverage a cryptographic key pair—securely provisioned in the protected disk (§5.3.1) and loaded by HKM during initialization (§5.1.3)—to establish a secure, authenticated secure channel through the OS-controlled network (§5.3.2). Since the OS cannot extract or modify these keys, it cannot compromise the channel (*e.g.*, impersonate the administrator). Even though the request notification between HKD and HKM is relayed by the OS (using an SMC), the request parameters (*e.g.*, which disk sectors to clear) are protected in HKD's memory (§5.2.3).

## 7.2 Protection Deadline Analysis

This section describes the worst-case deadlines ($T_p + \epsilon$) found for in-memory log protection, given different configurations of timer frequency ($T_p$) on our test system.

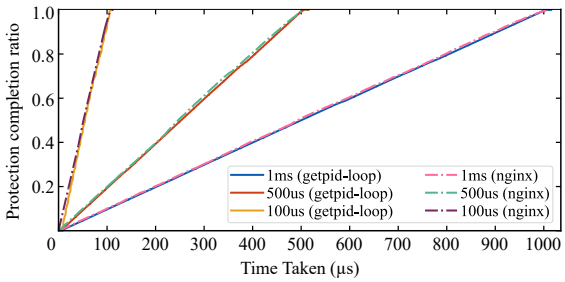



**Figure 3: CDF shows the relation between different protection timer $T_p$ ($1ms$, $500\mu s$, and $100\mu s$) deadline settings and HitchHiker's actual in-memory protection window (delay).**

**Settings.** We ran two workloads. The first workload is from a stressful benchmark, *getpid-flood*, based on prior work [39]. This benchmark executes one million `getpid` system calls in a loop to intensively generate logs. The second workload is from a real-world program, *Nginx*, which produces diverse logs. For Nginx, we used benchmark `ab` [1] under settings in Tab. 6. We evaluated HitchHiker in three configurations of $T_p$: $1ms$, $500\mu s$, and $100\mu s$. Both S2PT and GPT are employed as the HitchHiker's permission primitive, respectively, and we report the worst-case experimental results. The experiment was repeated 100 times.

**Results.** Fig. 3 shows the cumulative distribution function (CDF) of the time taken to protect logs with different $T_p$. Under the *getpid-flood* benchmark, when $T_p$ is assigned values of $1ms$, $500\mu s$, and $100\mu s$, 99.6%, 99.1%, and 92.8% of logs, respectively, were protected in a window less than the assigned $T_p$. Each configuration had an additional delay ($\epsilon$) in the worst case, which ranged between $12.12\mu s$ and $14.96\mu s$. In the case of *nginx*, 99.5%, 99.3%, and 95.5% of logs were protected within $T_p$ (ranging from $1ms$ to $100\mu s$, respectively). The worst-case values of $\epsilon$ ranged between $11.52\mu s$ and $13.73\mu s$. As Nginx's workloads also generate considerable logs (around 79k per second), its CDF is similar to getpid-flood (which generates logs at around 178k per second) but slightly better. Under both settings, the delay $\epsilon$ was negligible and stable, due to HitchHiker's enforcement mechanisms (§5.2.2) and fast permission switches (§4.2).

Recall that the state-of-the-art asynchronous protection system [39] incurs a best-case deadline of $15ms$. Hence, with configurable deadlines between roughly $1.012ms$ and $115\mu s$, HitchHiker exposes 93.3% – 99.3% shorter attack windows. The next section shows how such short protection windows further prevent log tampering attacks when compared to prior work.

## 7.3 Log Tampering Case Study

To evaluate whether HitchHiker's short deadlines provide a significant hurdle for adversaries to tamper with logs, we simulated a *powerful local attacker* to delete logs in memory as fast as possible.

**Attack setup.** We consider powerful local attackers who know the log buffers' physical addresses. Attackers also *already held* the foothold to access the system (at time $t_s$). Such settings avoid time to establish footholds. To begin with, attackers exploit kernel vulnerabilities to escalate privileges to root at $t_c$. Upon system corruption, attackers halt the log generator and delete logs in memory.

We reproduced ten kernel CVEs with PoC exploits to escalate privileges to root. CVEs were selected from well-known security

**Table 5: Local attack time and log event statistics. #Total Logs: the number of log events captured during the attack. #Lost Logs: the number of events that are cleared by the attacker. $T_p$ 15$ms$: state-of-the-art asynchronous protection [39].**

| Attack Time Statistics (ms) | | | | Attack Trace Event Statistics (#) | | | |
|---|---|---|---|---|---|---|---|
| Exploit | Escalate | Load | Clear | #Total | #Lost Logs (Min/Avg/Max) | | |
| DirtyPipe | privilege | .ko | Buffer | Logs | $T_p$: 100µs | 1ms | 15ms [39] |
| 3.73 | 3.76 | 4.89 | 0.02 | 1,407 | 0/2/**7** | 3/16/**35** | 7/618/**all** |

assessment forums (*e.g.*, *Seclist, Openwall*) with high Common Vulnerability Scoring System (CVSS) [32] scores. Also, they were collected from two categories [50]: (i) semantic bugs like improper permissions or security checks, and (ii) memory corruptions (*e.g.*, stack or heap-related temporal and spatial corruptions).

To simulate a powerful attacker, we chose the fastest out of the ten exploits on our host machine (§6): DirtyPipe [74]. The full list of reproduced CVEs and their execution times are provided in Appendix §A [110]. After gaining root, the attacker launches a malicious kernel module (`.ko`) to *clear logs residing in the unprotected buffer*, erasing the attack evidence. During the attack, HitchHiker was fully deployed with different protection timer configurations $T_p$ of 15$ms$, 1$\mu s$, and 100$\mu s$. We ran this attack 50 times by activating the attack script at random times (same as the §3.2).

**Results.** Tab. 5 shows the statistics of the attack, indicating an average total attack time of 12.4$ms$ with 1,407 attack-related logs generated. Threats from semantic bugs (*e.g.*, DirtyPipe) can be efficiently exploited by attackers (as stated in §3.2 and in Tab. 5), giving them a high chance of removing considerable attack traces before a single large protection window (*e.g.*, removed 618 out of 1,407 logs on average given a 15$ms$ protection window), or compromising full attack traces. Reducing the protection window significantly, by 93.3% to 1$ms$ or by 99.3% to 100$\mu s$, proves effective in hindering log tampering. In these scenarios, the attacker can delete only 3% (35 logs) or 0.2% (7 logs) of total logs at most, respectively.

Notably, as the last step for log deletion, installing a malicious kernel module requires around 4.9$ms$. This time span significantly exceeds given a $T_p$ of 1$ms$. As a result, all logs during the exploit of DirtyPipe and escalating privilege were logged and protected, even with a 1$ms$ $T_p$. In fact, only the "final millisecond" logs during module installation were lost. Hence, this deadline (of 1$ms$) setting also *preserved the availability of the full attack trace.*

**Memory corruption attack discussion.** Admittedly, a pure memory corruption attack like in-kernel ROP can *avoid kernel module loading* after privilege escalation and tamper with logs faster, by memory write gadgets. However, unlike userspace exploits, chaining gadgets in-kernel for arbitrary code execution may be challenging and time-consuming. Our reproduced real-world memory corruption exploits (see Appendix §A in [110]) also demonstrate a lengthy attack preparation of more than 1$s$, making it impossible to use such attacks to compromise logs within 1$ms$ windows. This is due to the time-consuming nature of manipulating memory layout (*e.g.*, heap spray), lack of fault tolerance for attack debugging, and defenses like data execution prevention (DEP) and guarded control stacks (GCS) [26]. So far, loading a malicious kernel module remains a popular rootkit [29, 34] to disable logging.

**Table 6: Real-world application and workload description.**

| Application | Workload Description |
|---|---|
| Nginx | Default 4 worker threads; tested with the `ApacheBench` of 10K requests for a default file and 32 client concurrency. |
| Apache | Default settings; tested with the `ApacheBench` of 10K requests for a default file and 32 client concurrency. |
| Redis | Default 16 databases; tested by `memtier_benchmark` of 1M sets and 1M gets for 32 bytes data with 50 clients. |
| Memcached | Default settings; tested by `memtier_benchmark` of 1M set and 1M get for 32 bytes data with 50 clients. |
| MySQL | Default 10 tables with table size 10,000; tested by `sysbench oltp_read_write` for total 10K transactions (≥ 200K queries). |
| 7zip | Phoronix benchmark `pts/compress-7zip`. |
| OpenSSL | Phoronix benchmark `pts/openssl`. |
| Firefox | `Speedometer 2.0` benchmark. |
| GNU Octave | Phoronix benchmark `system/octave-benchmark`. |
| Wget | No-cache and quiet mode; fetching a default file with 10K runs. |

Regardless, when shrinking the protection window to a real-time value of around 1$ms$, HitchHiker significantly raises the difficulty for adversaries to alter the vast majority of their attack traces.

## 7.4 TCB and Vulnerability Discussion

HitchHiker's TCB is comprised of HkM and HkD. HkM requires an additional 2.2k LoC based on TF-A (29k LoC[2]), while the HkD requires 0.8k LoC. By debloating the driver via record-and-replay, it avoids the adoption of a commodity Linux driver that contains more than 10k LoC and complex kernel dependencies [63].

In total, HitchHiker requires a TCB of around 32k LoC. This is significantly smaller than typical virtualization software (*e.g.*, larger than 862k LoC with KVM [69]), or the TZ components (*e.g.*, 300k LoC of OP-TEE) used by existing solutions [62, 97]. Moreover, unlike prior work, HitchHiker does not rely on full-fledged components (both hypervisor and TrustZone), thereby avoiding sharing with their interfaces and attack surfaces (Tab. 1). We further analyzed the security properties of HitchHiker in terms of attack surface reduction (compared to existing solution [62]) in Appendix §B [110]. In conclusion, HitchHiker reduced their TCB by 9.4 – 26.9×, and avoided the general-purpose attack surfaces.

# 8 Performance Evaluation

All experiments were performed on our JUNO R2 computer (§6). For client-server programs, the client workload operated on a desktop with Intel(R) Core(TM) i7-10700 CPU, 16GB RAM, and a 512GB hard disk, connected to the JUNO server via Gigabit Ethernet.

Unless otherwise specified, we measured HitchHiker's performance with the $T_p$ of 1$ms$, since it can prevent attack trace tampering even in strong attacks (§7.3). We configured HitchHiker's kernel buffer pool with 16 buffers (a total of 1MB size). This setting very rarely experienced buffer fill-up delays (§5.2).

**Comparison settings.** We used four comparison settings to evaluate our system, which we describe in the following paragraphs.

**Native** indicates system performance without any kind of logging. **Native-OBS** is the performance when logs are captured and then in memory by the native observability generator (extended

[2]The LoC of TF-A is calculated by counting the compiled sources. All uncompiled other platform or SoC-related driver and library sources are excluded.

**Table 7: Throughput under the *getpid-flood* (§7.2) benchmark.**

| Approach | Throughput (logs/sec) | Relative Percent (%) |
|---|---|---|
| **Native-OBS** | 201,038 | 100% |
| **HitchHiker-IM** $T_p$: 1*ms* / 500*µs* / 100*µs* | 178,571 / 172,053 / 166,500 | 88.8% / 85.5% / 81.3% |
| **HitchHiker-GPT** $T_p$: 1*ms* / 500*µs* / 100*µs* | 177,399 / 168,711 / 154,966 | 88.2% / 83.9% / 77.1% |
| **HitchHiker-S2PT** $T_p$: 1*ms* / 500*µs* / 100*µs* | 173,094 / 165,253 / 155,603 | 86.1% / 82.2% / 77.4% |
| **HitchHiker-MCPY** $T_p$: 1*ms* / 500*µs* / 100*µs* | 150,466 / 137,757 / 120,376 | 74.8% / 68.5% / 59.9% |
| **OmniLog** | 99,143 | 49.5% |

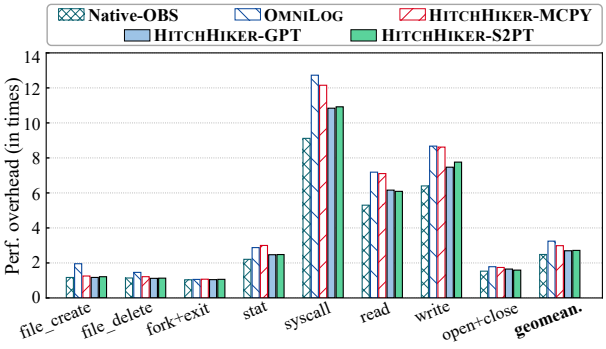



**Figure 4: Overhead of HitchHiker on LMBench [86].**

Tracee [2]). Native-OBS does not persist or protect logs, thus serving as the ideal (maximum) performance in our system with logging.

To show the advantage of hardware permission-based protection, we also evaluated a HitchHiker variant employing *EL3-memcpy*-based (§4.2) protection (labeled **HitchHiker-MCPY**), which copies logs from the OS domain to the protected. Furthermore, to show the cost of storage operations within our system, we also evaluated a HitchHiker variant without any such operations, called **HitchHiker-IM**. We evaluate **HitchHiker-S2PT** and **HitchHiker-GPT**, by using S2PT and GPT, respectively.

Finally, we compared HitchHiker with the state-of-the-art synchronous protection approach, OmniLog [62] (referred to as **OmniLog**). Note that OmniLog is designed for audit (system) log protection and integrated with Linux Auditd [30]. For a fair comparison, we made our best efforts to reproduce OmniLog and incorporated it with our observability generator, focusing only on in-memory protection (*i.e.*, logs are discarded after synchronously being copied into EL3 memory by OmniLog). This avoids storage overhead to optimistically approximate its performance.

## 8.1 Micro-benchmarks

This section describes HitchHiker's throughput breakdown and its system-event-level performance.

**Log throughput breakdown with *getpid-flood*** . To break down the impact of different aspects of HitchHiker (*e.g.*, protection, persistence), we leveraged the *getpid-flood* micro-benchmark (§7.2) to generate logs intensively evaluate the performance.

*Results.* Tab. 7 illustrates the results. The throughput of Native-OBS (201,038 logs/second) acts as the reference point, representing the theoretical maximum observability log throughput of our system. HitchHiker-GPT reaches a throughput of 177,399 entries per second (88.2% of Native-OBS) under the 1*ms* deadline setting. Similarly, at such settings, HitchHiker-S2PT has 86.1% of the Native-OBS throughput. Furthermore, as the timer frequency $T_p$ shortens from 1*ms* to 100*µs*, there is a corresponding decline in the relative throughput of HitchHiker (from 88.2% to 77.1%). This is expected, as more frequent protection enforcement introduces more system overheads for HitchHiker to respond.

HitchHiker-MCPY has a throughput varies between 74.8% and 59.9%, according to the $T_p$ setting ranging from 1*ms* to 100*µs*. This further shows the advantage of the hardware permission switch over memory copy. OmniLog displays a relative throughput of 49.5%, underscoring its significant slowdown due to frequent synchronous system blocks and log protections.

Compared to HitchHiker-IM, which solely employs memory protection, HitchHiker (S2PT and GPT) experiences an additional 0.6% to 4.2% reduction in throughput depending on the deadline settings. This overhead is attributed to the activation of HkD's manager thread, which causes extra system burdens (*e.g.*, task synchronization and I/O). Nevertheless, the performance impact of the background log persistence remains limited, primarily due to the utilization of high-bandwidth SATA SSD.

**System event latency.** We evaluated HitchHiker's slowdown upon kernel operations by using the widely adopted [39, 88] *lmbench* [86]. We executed latency benchmarks for (a) file system creation/deletion, (b) process creation/exit, and (c) syscall.

*Results.* Fig. 4 shows the evaluation results. HitchHiker-GPT and HitchHiker-S2PT imposes a geometric mean of 1.69× and 1.71× overhead more than Native, while adding only 8.4% and 9.3% overhead to Native-OBS (1.48× more than native), respectively. The overheads for the memory copy-based approaches, HitchHiker-MCPY and OmniLog, are respectively 1.99× (20.6%) and 2.25× (31%) higher than Native-OBS. As the lmbench only generates kernel events, the large overheads for HitchHiker mainly come from the Native-OBS (1.48× more than Native). This overhead is amortized during real-world workload execution (next section). In the future, observability generation overhead can be optimized using eBPF compile-optimization [84] and efficient log collection [82, 95].

## 8.2 Real-world Programs

This section describes HitchHiker's performance on real-world workloads using common benchmarks. The overhead is compared with different system settings mentioned in §8.

**Settings.** We chose five common client-side applications (7zip, OpenSSL, Firefox, GNU Octave, and Wget) and server-side applications (Nginx, Apache, Redis, Memcached, and MySQL). We divide applications by client and server workloads, because the latter typically produce significantly higher amounts of logs. These applications have also been evaluated under similar workloads by prior work [39, 62, 89]. Detailed workloads are described in Tab. 6.

**Results.** Fig. 5 illustrates the performance overhead incurred by real-world applications. HitchHiker-GPT exhibits a geometric mean overhead of 1.8% and 9.9% over Native in client-side and server-side programs, respectively. Similarly, HitchHiker-S2PT's

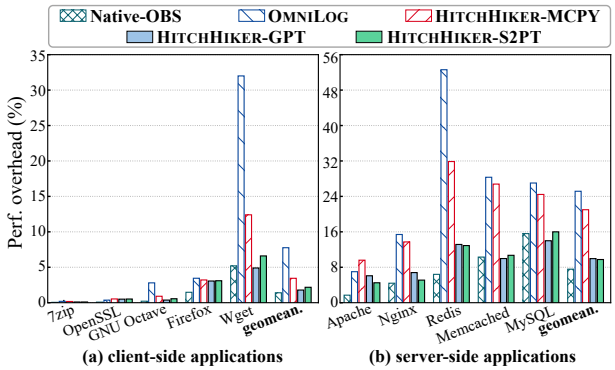



**Figure 5: Real-world workload overheads. From left to right, the log throughput for each program is: 1907, 589, 32358, 7234, 31434; 10999, 78776, 78537, 93928, 101962 logs/sec.**

geometric mean overheads are 2.2% and 9.7%. In contrast, HitchHiker-MCPY's geometric mean overheads are 3.4% and 20.9% over Native, on client-side and server-side programs, respectively. In the worst-case, HitchHiker-MCPY's overhead is up to 31.9%, while HitchHiker's worst remains at only 13.9%. Recall the protection cost described in Tab. 3; *el3-memcpy* is expensive due to its CPU-intensive nature and its lack of optimizations in EL3 software [19]. Thus, HitchHiker-MCPY imposes an overhead of $0.5 - 1.2\times$ more than HitchHiker-GPT and HitchHiker-S2PT, which leverage efficient hardware permission switch (§5.2).

OmniLog also adopts *el3-memcpy* for in-memory log protection, but also *synchronously* protects every log. It results in a geometric mean of 8% client-side and 26% server-side overheads over Native. Client programs typically focus on computational tasks (*e.g.*, compression, encryption, and calculation) and thus are "*log-sparse*" that generate fewer logs. However, Wget (a web downloader) is an exceptional client program. Due to its frequent system calls and network transitions, OmniLog imposes a 30% overhead over Native on it. Conversely, server-side applications are characterized as "*log-intensive*", owing to their frequent I/O tasks for invoking system calls, recording application logs into local file systems, and communicating with clients through the network. Thus, given the high observability log throughput, OmniLog results in a large overhead (*e.g.*, up to 52% over Native in Redis) due to intensively blocking the system execution to wait for log protection. In all scenarios, OmniLog's synchronous protection incurs $0.12 - 0.65\times$ overheads over asynchronous memory copying (HitchHiker-MCPY), or $0.2 - 3\times$ over hardware-based permission switching (HitchHiker).

**Takeaway.** Despite enforcing short real-time deadlines (around 1*ms*), HitchHiker provides high performance. On *log-sparse* workloads, all solutions provide decent performance (less than 8% overhead), yet HitchHiker provides near-native performance (2% geometric mean overhead only). On *log-intensive* applications, synchronous or periodic copying of logs to privileged memory becomes inefficient (up to 52.6% overheads). Here, HitchHiker with permission switch incurs a geometric mean of only 9.9% (up to 15.9% only), which is up to 77.5% and 61.9% lower than OmniLog on client and server-side programs, respectively, showing its efficiency.

## 9 Timer Protection Discussion

A limitation of our current implementation (§6) is that attackers may use kernel vulnerabilities in the kernel, even before it is compromised, to trick it into manipulating the GIC register; thereby disabling timer interrupt prioritization. For instance, attackers may leverage code reuse (e.g., ROP) attacks. We found that the preparation steps for such attacks typically take a long time ($\geq 100ms$, Appendix §A [110]); thus, the initial logs (before GIC manipulation) will still be protected to reveal attack preparations. Nevertheless, such an attack would give attackers an extended window to tamper with the remaining logs during full system compromise.

There are two potential solutions to prevent the aforementioned attack. The first solution is to write-protect the MMIO interface of GIC's interrupt priority register (e.g., using S2PT or GPT to remove the permissions for accessing the MMIO page in the OS domain). By doing so, only the security monitor controls the interface of GIC interrupt priority registers; thus, any attempts to manipulate them are interposed by the monitor. The second solution is to isolate the entire timer interrupt handling into the protected domain. In particular, the monitor (HkM) may (a) initialize the hardware timer interrupt and lock it down to a secure interrupt (in GICv2 [104]), or (b) set the interrupt route model to route it exclusively to EL3 (in GICv3 [16]). We leave exploring and implementing the aforementioned two solutions for future work.

## 10 Related Work

**System Observability.** Observability is crucial for system dissection and analysis with a broad spectrum of applications [78, 87]. Researchers utilize different levels of observability by capturing program semantics via application logs [67] for anomaly detection [55, 111] and failure diagnosis [53, 107]; monitoring syscall-level audit logs [72] for attack detection [65, 108] and investigation [75, 105]; and utilizing network logs [59] for malware detection [60, 61], forensics and traffic classification [101, 114]. Fusing multi-layer observability integrates a holistic security profile. This is achieved by incorporating application logs [66, 106], network logs [43, 54], library function logs [103], and instruction traces [109] with audit logs, strengthens system forensics and supports fine-grained provenance reconstruction. HitchHiker is the first system for cross-layer system observability protection and maintenance.

**Trusted execution environment.** Enclaves [6] enable applications like stream processing [90], mobile app protection [94], control flow attestation [38], security policy enforcement [46], and serverless computing [80, 113]. Researchers explored building enclaves with different abstraction levels, including userspace process [52, 70], secure containers [42, 96], and confidential VMs [71, 79]. Works adopt hardware protections like TZASC [45], S2PT [64, 69], GPT [100, 112], or even hardware-software co-design [58] to support domain enforcement for general-purpose enclaves. Inspired by those works, HitchHiker employs hardware features to tailor the secure environment exclusively for its secure log management.

**Tamper-evident logging.** Instead of providing log integrity and availability against tampering attacks, *tamper-evident* schemes detect tampering using cryptographic integrity proofs. Logs are either asynchronously [44, 73, 88, 97] or synchronously [68, 89] signed by Message Authentication Code (MAC). After creating signatures,

logs as well as their signatures (as the proof-of-integrity) are securely sealed (encrypted) on local storage and later sent to the central servers for further authentication. In the future, combining tamper-evident logging and HitchHiker can lead to the best of both worlds, namely the real-time availability and tamper evidence.

## 11 Conclusion

HitchHiker is an efficient and high-assurance observability protection system. It leverages efficient memory permission switches and hardware timers to enforce real-time short and configurable log protection deadlines. By its first principles approach of secure environment design, it debloats the secure environment and avoids sharing attack surfaces with other system components. Compared to prior work, HitchHiker achieves 93.3 – 99.3% shorter log protection deadlines (protecting the vast majority of logs), 9.4 – 26.9× smaller TCB, while reducing performance overheads by 61.9–77.5%.

## Acknowledgments

We thank the shepherd and anonymous reviewers for their constructive feedback in finalizing this paper. We thank Michael Swift for his insightful comments on the manuscript and Jinting Wu for his assistance with the experiments. We also thank all the members from the ASTERISC, COMPASS, and CURIOSITY labs for their insightful comments and feedback. This research/project is supported by the National Research Foundation, Singapore under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative, the US Air Force Office of Scientific Research (AFOSR) under award number FA9550-24-1-0204, and the National Natural Science Foundation of China (NSFC) under Grant No.62372218. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of National Research Foundation, Singapore, AFOSR, or NSFC.

## References

[1] The apache software foundation, "ab - apache http server benchmark tool". https://httpd.apache.org/docs/2.4/programs/ab.html.
[2] Aqua tracee, "tracee: Linux runtime security and forensics using ebpf". https://github.com/aquasecurity/tracee.
[3] Arm architecture reference manual for a-profile architecture. https://developer.arm.com/documentation/ddi0487/latest/.
[4] Arm corelink tzc-400 trustzone address space controller. https://developer.arm.com/documentation/ddi0504/c.
[5] Arm system memory management unit architecture specification, smmu architecture version 3. https://developer.arm.com/documentation/ihi0070/latest/.
[6] Confidential computing: Hardware-based trusted execution for applications and data. https://confidentialcomputing.io/wp-content/uploads/sites/10/2023/03/CCC_outreach_whitepaper_updated_November_2022.pdf.
[7] "d5.3 vmsav8-64 translation table format descriptors" – arm architecture reference manual for a-profile architecture. https://developer.arm.com/documentation/ddi0487/latest/.
[8] Different types of logs in siem and their log formats. https://www.manageengine.com/log-management/siem/collecting-and-analysing-different-log-types.html.
[9] Dynamically program the kernel for efficient networking, observability, tracing, and security. https://ebpf.io/.
[10] Getting started with intel® active management technology. https://www.intel.com/content/www/us/en/developer/articles/guide/getting-started-with-active-management-technology.html.
[11] Global incident response threat report. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/docs/vmwcb-report-the-ominous-rise-of-island-hopping-and-counter-incident-response-continues.pdf.
[12] Hackers are increasingly destroying logs to hide attacks. https://www.zdnet.com/article/hackers-are-increasingly-destroying-logs-to-hide-attacks/.
[13] HitchHiker artifact. https://github.com/ASTERISC-Release/HitchHiker.
[14] How to view kernel messages in linux | dmesg command. https://www.geeksforgeeks.org/how-to-use-the-dmesg-command-on-linux/.
[15] Indicator removal: Clear linux or mac system logs. https://attack.mitre.org/techniques/T1070/002/.
[16] Interrupt management framework. https://trustedfirmware-a.readthedocs.io/en/latest/design/interrupt-framework-design.html.
[17] Learn the architecture - realm management extension, "impact on translation lookaside buffers and caches". https://developer.arm.com/documentation/den0126/0100/Physical-Addresses.
[18] libbpf. https://docs.kernel.org/bpf/libbpf/index.html.
[19] memcpy_s.c, "arm-trusted-firmware". https://github.com/ARM-software/arm-trusted-firmware/blob/master/lib/libc/memcpy_s.c.
[20] memtier-benchmark. https://github.com/RedisLabs/memtier_benchmark.
[21] Observability in security. https://techblog.cisco.com/blog/observability-in-security.
[22] Op-tee. https://www.op-tee.org/.
[23] Overlayfs — privilege escalation. https://systemweakness.com/cve-2021-3493-overlayfs-privilege-escalation-51ba49c3255c.
[24] Sata sil24. https://archive.kernel.org/oldwiki/ata.wiki.kernel.org/index.php/Sata_sil24.html.
[25] Secure monitor calling convention. https://developer.arm.com/Architectures/SMCCC.
[26] Shadow stacks for 64-bit arm systems. https://lwn.net/SubscriberLink/940403/c4561635ec6d8881/.
[27] sysbench, "scriptable database and system performance benchmark". https://github.com/akopytov/sysbench.
[28] Uefi specification – "runtime services". https://uefi.org/specs/UEFI/2.10/08_Services_Runtime_Services.html.
[29] Understanding, detecting, & preventing modern linux rootkits. https://blog.securityinnovation.com/modern-linux-rootkits.
[30] Understanding linux audit. https://documentation.suse.com/sles/12-SP4/html/SLES-all/cha-audit-comp.html.
[31] Virtualization-based security (vbs). https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs.
[32] What is common vulnerability scoring system (cvss). https://www.sans.org/blog/what-is-cvss/.
[33] Xen project. https://xenproject.org/.
[34] "pulling back the curtain" – rootkit detection and removal. https://securenetworkers.com/2019/07/31/pulling-back-the-curtain-rootkit-detection-and-removal/.
[35] Arm. arm confidential compute architecture. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture, 2021.
[36] The realm management extension (rme), for armv9-a. https://developer.arm.com/documentation/ddi0615/latest, 2021.
[37] Trusted-firmware-a. https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git, 2022.
[38] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-flat: Control-flow attestation for embedded systems software. In *CCS*, 2016.
[39] Adil Ahmad, Sangho Lee, and Marcus Peinado. Hardlog: Practical tamper-proof system auditing using a novel audit device. In *S&P*, 2022.
[40] Adil Ahmad, Botong Ou, and Congyu Liu et al. Veil: A protected services framework for confidential virtual machines. In *ASPLOS*, 2024.
[41] Adil Ahmad, Alex Schultz, Byoungyoung Lee, and Pedro Fonseca. An extensible orchestration and protection framework for confidential cloud computing. In *OSDI*, 2023.
[42] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with intel SGX. In *OSDI*, 2016.
[43] Adam Bates, Wajih Ul Hassan, Kevin Butler, Alin Dobra, Bradley Reaves, Patrick Cable, Thomas Moyer, and Nabil Schear. Transparent web service auditing via network provenance functions. In *WWW*, 2017.
[44] Kevin D. Bowers, Catherine Hart, Ari Juels, and Nikos Triandopoulos. Pillarbox: Combating next-generation malware with fast forward-secure logging. In *RAID*, 2014.
[45] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *NDSS*, 2019.
[46] Ferdinand Brasser, Daeyoung Kim, Christopher Liebchen, Vinod Ganapathy, Liviu Iftode, and Ahmad-Reza Sadeghi. Regulating arm trustzone devices in restricted spaces. In *MobiSys*, 2016.
[47] David Cerdeira, José Martins, Nuno Santos, and Sandro Pinto. ReZone: Disarming TrustZone with TEE privilege reduction. In *USENIX Security*, 2022.
[48] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *S&P*, 2020.
[49] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *ASPLOS*, 2013.

[50] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *APSys*, 2011.

[51] Jiahao Chen, Dingji Li, Zeyu Mi, et al. Security and performance in the delegated user-level virtualization. In *OSDI*, 2023.

[52] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS*, 2008.

[53] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, Roberto Natella, and Nematollah Bidokhti. How bad can a bug get? an empirical analysis of software failures in the openstack cloud computing platform. In *ESEC/FSE*, 2019.

[54] Pubali Datta, Isaac Polinsky, Muhammad Adil Inam, Adam Bates, and William Enck. ALASTOR: Reconstructing the provenance of serverless intrusions. In *Security*, 2022.

[55] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *CCS*, 2017.

[56] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through Virtual-Machine logging and replay. In *OSDI*, 2002.

[57] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *MICRO*, 2016.

[58] Erhu Feng, Xu Lu, Dong Du, et al. Scalable memory protection in the PENGLAI enclave. In *OSDI*, 2021.

[59] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *NSDI*, 2007.

[60] Chuanpu Fu, Qi Li, Meng Shen, and Ke Xu. Realtime robust malicious traffic detection via frequency domain analysis. In *CCS*, 2021.

[61] Zhuoqun Fu, Mingxuan Liu, Yue Qin, Jia Zhang, Yuan Zou, Qilei Yin, Qi Li, and Haixin Duan. Encrypted malware traffic detection via graph-based network analysis. In *RAID*, 2022.

[62] Varun Gandhi, Sarbartha Banerjee, Aniket Agarwal, Adil Ahmed, Sangho Lee, and Marcus Peinado. Rethinking system audit architectures for high event coverage and synchronous log availability. In *Security*, 2023.

[63] Liwei Guo and Felix Xiaozhu Lin. Minimum viable device drivers for arm trustzone. In *EuroSys*, 2022.

[64] Seung-Kyun Han and Jinsoo Jang. Mytee: Own the trusted execution environment on embedded devices. In *NDSS*, 2023.

[65] Xueyuan Han, Thomas F. J.-M. Pasquier, Adam Bates, James Mickens, and Margo I. Seltzer. Unicorn: Runtime provenance-based detector for advanced persistent threats. In *NDSS*, 2020.

[66] Wajih Ul Hassan, Mohammad A Noureddine, Pubali Datta, and Adam Bates. Omegalog: High-fidelity attack investigation via transparent multi-layer log analysis. In *NDSS*, 2020.

[67] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R. Lyu. A survey on automated log analysis for reliability engineering. *ACM Comput. Surv.*, 2021.

[68] Viet Tung Hoang, Cong Wu, and Xin Yuan. Faster yet safer: Logging system via Fixed-Key blockcipher. In *Security*, 2022.

[69] Alexander Van't Hof and Jason Nieh. BlackBox: A container security monitor for protecting containers on untrusted operating systems. In *OSDI*, 2022.

[70] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, et al. Inktag: Secure applications on an untrusted operating system. In *ASPLOS*, 2013.

[71] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. vTZ: Virtualizing ARM TrustZone. In *USENIX Security*, 2017.

[72] M. Inam, Y. Chen, A. Goyal, J. Liu, J. Mink, N. Michael, S. Gaur, A. Bates, and W. Ul Hassan. Sok: History is a vast early warning system: Auditing the provenance of system intrusions. In *S&P*, 2023.

[73] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. Sgx-log: Securing system logs with sgx. In *AsiaCCS*, 2017.

[74] Max Kellermann. The dirty pipe vulnerability. https://dirtypipe.cm4all.com/.

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

[76] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. Pesos: Policy enhanced secure object store. In *EuroSys*, 2018.

[77] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. ExpRace: Exploiting kernel races through raising interrupts. In *USENIX Security*, 2021.

[78] Bowen Li, Xin Peng, Qilin Xiang, Hanzhang Wang, Tao Xie, Jun Sun, and Xuanzhe Liu. Enjoy your observability: An industrial survey of microservice tracing and analysis. *ESE*, 2022.

[79] Dingji Li, Zeyu Mi, Yubin Xia, et al. Twinvisor: Hardware-isolated confidential virtual machines for arm. In *SOSP*, 2021.

[80] Mingyu Li, Yubin Xia, and Haibo Chen. Confidential serverless made efficient with plug-in enclaves. In *ISCA*, 2021.

[81] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and verification of the arm confidential compute architecture. In *OSDI*, 2022.

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

[83] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. Dirtycred: Escalating privilege in linux kernel. In *CCS*, 2022.

[84] Jinsong Mao, Hailun Ding, Juan Zhai, and Shiqing Ma. Merlin: Multi-tier optimization of ebpf code for performance and compactness. In *ASPLOS*, 2024.

[85] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP*, 2013.

[86] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *ATC*, 1996.

[87] Francesco Minna and Fabio Massacci. Sok: Run-time security for cloud microservices. are we there yet? *Computers & Security*, 2023.

[88] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher W. Fletcher, Andrew Miller, and Dave Tian. Custos: Practical tamper-evident auditing of operating systems using trusted execution. In *NDSS*, 2020.

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

[90] Heejin Park, Shuang Zhai, Long Lu, and Felix Xiaozhu Lin. StreamBox-TZ: Secure stream analytics at the edge with TrustZone. In *ATC*, 2019.

[91] Thomas Pasquier, Xueyuan Han, Mark Goldstein, et al. Practical whole-system provenance capture. In *SoCC*, 2017.

[92] Thomas Pasquier, Xueyuan Han, Thomas Moyer, et al. Runtime analysis of whole-system provenance. In *CCS*, 2018.

[93] Avi Qumranet, Yaniv Qumranet, Dor Qumranet, Uri Qumranet, and Anthony Liguori. Kvm: The linux virtual machine monitor. *Linux Symposium*, 2007.

[94] Nuno Santos, Himanshu Raj, Stefan Saroiu, et al. Using arm trustzone to build a trusted language runtime for mobile applications. In *ASPLOS*, 2014.

[95] R. Sekar, H. Kimm, and R. Aich. eaudit: A fast, scalable and deployable audit data collection system. In *S&P*, 2024.

[96] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, et al. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *ASPLOS*, 2019.

[97] Carlton Shepherd, Raja Naeem Akram, and Konstantinos Markantonakis. EmLog: Tamper-Resistant System Logging for Constrained Devices with TEEs. In *WISTP*, 2017.

[98] Le Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. Deconstructing xen. In *NDSS*, 2017.

[99] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with SGX enclaves. In *NDSS*, 2017.

[100] Supraja Sridhara, Andrin Bertschi, Benedict Schlüter, Mark Kuhne, Fabio Aliberti, and Shweta Shinde. Acai: Extending arm confidential computing architecture protection from cpus to accelerators, 2023.

[101] Benjamin E. Ujcich, Samuel Jero, Richard Skowyra, Adam Bates, William H. Sanders, and Hamed Okhravi. Causal analysis for Software-Defined networking attacks. In *Security*, 2021.

[102] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Aastha Mehta, Deepak Garg, Peter Druschel, Rodrigo Rodrigues, Johannes Gehrke, and Ansley Post. Guardat: Enforcing data policies at the storage layer. In *EuroSys*, 2015.

[103] Fei Wang, Yonghwi Kwon, Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Lprov: Practical library-aware provenance tracing. In *ACSAC*, 2018.

[104] Jinwen Wang, Ao Li, Haoran Li, and et al. Rt-tee: Real-time system availability for cyber-physical systems using arm trustzone. In *S&P*, 2022.

[105] Zhiqiang Xu, Pengcheng Fang, Changlin Liu, et al. Depcomm: Graph summarization on system audit logs for attack investigation. In *S&P*, 2022.

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

[107] Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. Comprehending performance from real-world execution traces: A device-driver case. In *ASPLOS*, 2014.

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

[109] Jun Zeng, Chuqi Zhang, and Zhenkai Liang. Palantir: Optimizing attack provenance with hardware-enhanced system observability. In *CCS*, 2022.

[110] Chuqi Zhang, Jun Zeng, Yiming Zhang, Adil Ahmad, Fengwei Zhang, Hai Jin, and Zhenkai Liang. The hitchhiker's guide to high-assurance system observability protection with efficient permission switches (extended). https://doi.org/10.48550/arXiv.2409.04484, 2024.

[111] Xu Zhang, Yong Xu, Qingwei Lin, et al. Robust log-based anomaly detection on unstable log data. In *ESEC/FSE*, 2019.

[112] Yiming Zhang, Yuxin Hu, Zhenyu Ning, Fengwei Zhang, et al. SHELTER: Extending arm CCA with isolation in user space. In *USENIX Security 23*, 2023.

[113] Shixuan Zhao, Pinshen Xu, Guoxing Chen, et al. Reusable enclaves for confidential serverless computing. In *USENIX Security*, 2023.

[114] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. Secure network provenance. In *SOSP*, 2011.