Measuring System Auditing Overheads in Real-Time Systems

Anant Kandikuppa
University of Illinois at Urbana-Champaign
{anantk3}@illinois.edu

Abstract

Securing Cyber-Physical and Real-Time systems is a growing concern as these systems interact with the physical environment. Off the shelf system auditing solutions like the Linux Audit framework, introduce significant overheads in terms of latency and storage, which are undesirable in this domain, which has strict task deadlines and resource constraints.

This work presents *Ellipsis*, a solution that reduces costs associated with system auditing in these time-sensitive systems, by exploiting the predictability in their task model to drastically cut down audit log generation without losing information valuable for forensic investigations. We evaluate the effectiveness of our solution using ArduPilot, an autopilot application suite, observing a **91%** reduction in storage costs while meeting temporal and auditing requirements of the application. *Ellipsis* provides a way forward for auditing CPS.

1 Introduction

Cyber-Physical Systems (CPS) are increasingly being used in automobiles, medical devices and industrial equipment. The use of internet connected devices in these domains is only expected to increase in the future with the advent of 5G technology [3]. Interaction of CPS with the physical environment is an important differentiator from run of the mill IoT devices. These interactions often introduce real-time (RT) constraints in the system. For example, one would expect airbags in an automobile to deploy within milliseconds of a collision.

Attacks on industrial CPS are not new. Stuxnet was uncovered in 2010 [24] amidst a lot of media attention and led to the discovery of a wide variety of long-term stealthy Advanced Persistent Threat (APT) attacks against industrial CPS. The US FDA recently notified healthcare providers about 11 security vulnerabilities that were found in actively used medical devices [9]. Over the past couple of years, automobile cyber-security incidents have nearly doubled as well [4].

Such threats have been around in more traditional datacenter settings for a while now. Host Intrusion Detection (HID) systems have been traditionally used to protect computing infrastructure against external attackers. HIDs typically monitor system events to derive a sense of normal system behavior, highlighting any deviation from the normal as attacker behavior. System audit logs are the most common source of event streams that HID systems build upon. The Linux Audit subsystem provides a means to capture auditing information with the Windows Event Viewer providing similar functionality in Windows.

But these auditing techniques cannot be directly applied to CPS due to its peculiar resource and timing constraints. Auditing introduces a significant temporal latency and storage costs which might adversely impact CPS performance [15]. Our novel audit reduction technique *Ellipsis*, aims to bring Linux Audit to real-time CPS while keeping up with its unique constraints. By showing a means to reduce audit log generation by leveraging predictable CPS behavior, we provide a way forward for enabling auditing in real-time cyber-physical systems.

The **contributions** of this paper can be summarized as :

- We present *Ellipsis*, a novel audit log reduction scheme that is designed for real-time CPS applications.
- We evaluate *Ellipsis* against a real world autopilot application and find that *Ellipsis* retains all audit information while meeting temporal requirements.

2 Background

2.1 Cyber Physical and Real-Time Systems

Cyber-Physical Systems (CPS) and Real-Time Systems (RTS) are digital systems that interact with the physical world. Examples of CPS include medical devices and systems, aerospace systems, autonomous vehicles, process control, and factory automation, building and environmental control and smart homes. These systems must operate dependably, safely, securely all in real-time. [19].

Real-Time systems are designed to fulfill a specific set of functions and have tasks that come with varying degrees of strictness in terms of deadlines [14], from hard real-time systems like anti-lock braking systems on automobiles to soft real-time systems like an audio-video streaming system. These tasks generally have periodic repetitive loops, for example, a sensor periodically collects data from its surroundings, the system decides upon an action based on these inputs and sends appropriate commands to actuators. Hard real-time systems are designed to operate predictably and meet deadlines even under worst-case loads. The applications that run on these systems are known beforehand and are not modified online. This allows them to be profiled, tuned and certified before they are deployed.

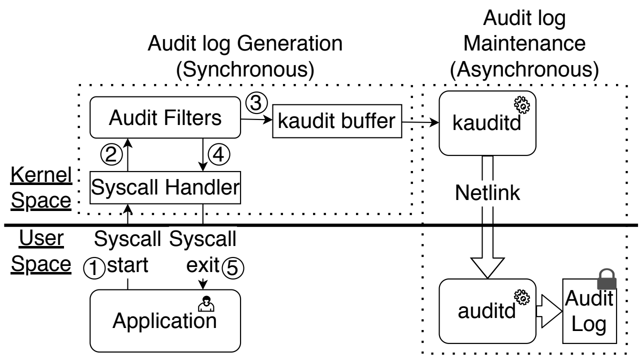

Figure 1. Architecture of Linux Audit Framework. Audit logs are generated using auditing hooks in kernel's system call (Syscall) handler and temporarily stored in the kaudit buffer. Log maintenance is handled by two background daemons, kauditd and auditd.

As these systems are becoming more pervasive these days, security concerns for RTS and CPS are gaining importance given their implications in the physical realm [12, 19].

2.2 System Causality Analysis

A system event trace is a sequence of recorded interactions among multiple system objects (processes, files, network connections, etc.) [22]. These traces capture information flows in the system and can be used to reconstruct causal dependencies between events.

We represent information flows in the form of a dependency or provenance graph. System events are edges in the graph, annotated with metadata such as event timestamps. The nodes in the graph are typically processes and files. This representation is crucial for many forensic applications, such as root cause diagnosis, intrusion recovery, attack impact analysis and forward tracking, which perform causality tracking on the graph [22]. The tracking can be either in the forward or backward direction. Backward tracing involves tracing paths back from a given node to the source of the attack, while forward tracing involves following edges from a given node to identify the impact of an exploit. It is often difficult to perform such tracing in a server setting due to the problem of dependency explosion[13].

The starting point for generating such graphs is auditing system-level events to capture event logs from the system. Linux Audit framework [2] is one of the most widely deployed auditing systems.

2.3 Linux Audit Framework

Linux Audit provides a way to track security-relevant information on a Linux based system. Based on user-configured rules, the audit system captures information about relevant events occurring in a system, most commonly in the form of system call logging. It is intended to be used for forensic analysis of mission-critical environments and determine violators of security policies along with the events which triggered the violations. [2]

The Audit system comprises of 2 sub-systems: the user-space applications and utilities, and the system call processing unit in kernel-space, as shown in Figure **??**. The system administrator specifies the audit rules and configurations using the Audit control utility `auditctl` and enables the system to be audited. The system call processing unit hooks into each system call invocation and filters relevant calls based on the configured rules. On intercepting these events, they are enriched with useful information like the executable name, the effective user id, inode information for files, etc and are queued onto a kernel buffer in a FIFO manner. A kernel thread `kauditd` reads these events and passes them onto the user-space daemon `auditd` over a netlink socket connection. `auditd` further ensures that these events are logged to disk successfully and are available for reporting and analysis via userspace tools like `aureport` and `ausearch`.

Table 1. Evaluation Platform

Platform	Raspberry Pi 4 [1]
Processor	ARM Cortex A-72 4 core [?]
Memory	4 GB
OS	Linux 4.19 [?]
Patches	Preempt RT [?]
Kernel Source	raspberrypi/linux [?]
Application	Ardupilot [21]

Linux Config	CONFIG_PREEMPT_RT_FULL = y
	CONFIG_AUDIT = y
	CONFIG_AUDITSYSCALL = y

Audit Config	backlog limit = 50000 backlog wait time = 0 failure flag = 1 #printk
CPUFreq Governor	Performance [?]

3 Measurement Setup / Design

3.1 System Model

3.2 Real Time Task Model

3.3 Instrumentation

4 Evaluation

Give context about benchmarking applications and why we chose them and the audit rules and stuff

4.1 Buffer Utilization

Experiment:

For measuring the utilization of the kaudit buffer, we periodically sampled the buffer state every 2 seconds using the audit command line utility *auditctl*, while executing the ArduPilot application for 100K iterations.

Observations: From Figure 2, we see that for Linux Audit, the utilization of the kaudit buffer rises quickly in the

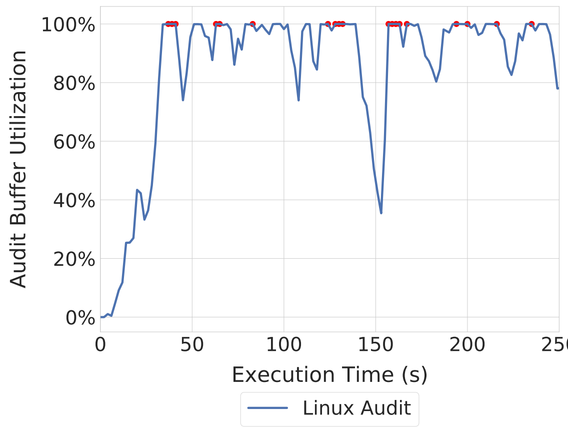

Figure 2. Kaudit buffer utilization over time for Linux Audit. Linux Audit's `backlog` value was measured every 2 seconds during the execution of the application and plotted as a percentage of the max backlog limit. Additional red annotations signify all times where buffer utilization was 100% and hence audit events might be lost. At these points Linux Audit performance is actually worse than depicted, but was constrained by the alotted buffer space.

beginning and remains close to 100% for the majority of the running time, resulting in loss of audit messages.

Discussion: When the kaudit buffer is full, new audit messages are lost; hence, to ensure that suspicious events are recorded, it is essential that *the buffer is never full*. The variations that we see in the plots can be attributed to the scheduling of the non real-time *kauditd* thread that is responsible for sending the outstanding audit messages to user-space for retention on disk. We observe that the backlog builds with time when *kauditd* isn't scheduled and drops sharply when *kauditd* eventually gets CPU time.

As *kauditd* is a non real-time thread running with no priority, it contends with multiple background processes for CPU time resulting in high buildup of messages in the kaudit buffer. Reducing the pressure of incoming audit messages using a kernel based log reduction schemef like KCAL [15] or increasing the rate of draining the kaudit buffer could be suitable approaches to ensure lossless auditing in resource constrained systems.

4.2 Audit Latency

Experiment:

For measuring the amount of time each audit message spends in the audit subsystem, the ArduPilot application was executed over 10K iterations and the audit latency was computed from the commit time and the audit time of each audit message obtained from the log file. The non real time *kauditd* and *auditd* threads are run with a nice value of 0 and -4 respectively.

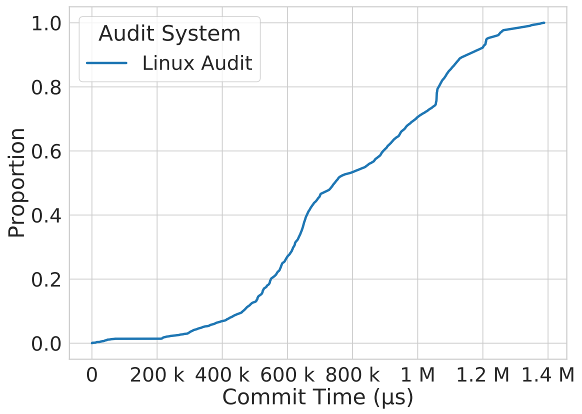

Figure 3. Audit commit latencies over time for Linux Audit plotted as a Cumulative Distribution Function (CDF). The experiment measures the time between the generation of an audit message and before the message is written to disk in userspace by the *auditd* thread. For a chosen commit time t on the x-axis, the y-axis shows the fraction of the total messages that encountered delay of less than t microseconds.

Observations: From Figure 3, we see that the audit latencies across 140k audit messages has a very wide range from 130 microseconds to 1.38 seconds with a median of $\tilde{7}40$ ms.

Discussion: The wide range of audit latencies points us to the fact that vanilla Linux Audit maybe a poor choice for real time systems, which typically demand predictable and tightly bound temporal performance guarantees. The low observed minimum audit latency suggests that the implementation of the audit subsystem itself may not entirely to blame for the high *kauditd* buffer utilization and reinforces our earlier observation that scheduling of audit threads is the primary reason for the long tail in audit latencies.

As the audit threads execute at lower non real time priorities, they rely on adequate application slack time to complete execution. With the introduction of synchronous and asynchronous overheads of auditing, the effective slack time reduces further taking away CPU time from the audit subsystem. Reliably accounting for these auditing overheads and increasing priorities of audit threads could help design a schedulable audited RT system and reign in the tail audit latencies.

4.3 Audit Throughput

Experiment:

For measuring the amount of time required to process a single audit message by the audit subsystem only, we ran a microbenchmark application that consisted of 10 *getpid* system calls and observed the intervals at which audit messages were written to disk over 10 runs. In the RT+PIN scenario, both audit threads were assigned a low real time priority and the *auditd* thread was pinned to core 2 on the machine. As

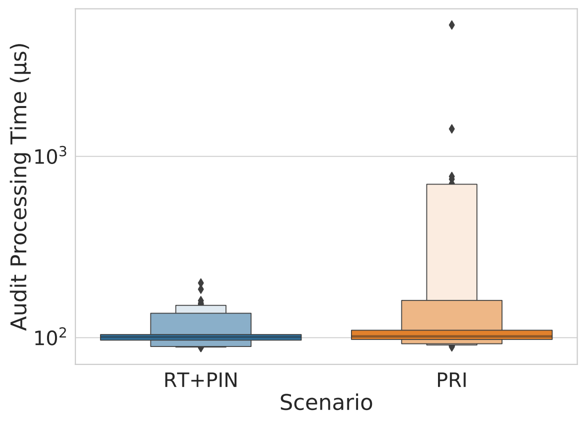

Figure 4. Audit Processing Time .

kauditd is a kernel thread, we weren't able to pin the thread to a fixed core on the system, instead relying on its real time priority to ensure that it ran ahead of any background process. Whereas in the PRI scenario, the priorities of the audit threads were increased by modifying their nice values, which resembles the default configuration of the audit subsystem.

Observations: From Figure 4, we observe that for the PRI case, the interval for processing audit messages varies from 88 μs to 5ms. Introducing real time priorities and core pinning reduces the variance in measurements and reduces the maximum interval to 200 μs.

Discussion:

Wrap up discussion - fairly obvious that low priority + core pinning would have worked - reduces time wasted in moving task around and moves it above background task, ensures CPU time - need to add an additional core to ensure enough CPU time for the buffer - that could be reasonable as it can be factored in during the design phase itself - having an upper bound on the audit throughput helps plan for schedulability

4.4 Summary of Results

Summarize takeaways - timing comes in handy to understand resource allocation - we can understand if log reduction is required once we are maxed out on our resource allocation.

5 Discussion

6 Related Work

6.1 Auditing Real Time Systems

The current state of CPS security has been studied in [5, 11, 20]. Sadehgi et al. [20] argue that the security playbook used for typical IT systems cannot be replicated for industrial CPS, because availability is a fundamental requirement for CPS. Their work also reinforces our assumption that real-time systems can be resource-constrained in terms of computation, memory, and energy.

With the use of CPS increasing in medical devices, automobiles, smart grids, and industrial equipment, attackers are able to exploit this ever growing attack surface and cause serious damage [11]

In order to detect ongoing attacks, intrusion detection systems have been designed specifically with the challenges of cyber-physical systems in mind [17, 23, 25]. These solutions can often require dedicated hardware [23] or depend on static analysis of the application code [25] to perform intrusion detection, which constrains the adoption of these solutions.

On the other hand, specific attacks such as ScheduLeak [8] have also been designed to target and exploit real-time systems as well. These attacks cannot be detected by the traditional intrusion detection systems as they utilize scheduling side channels, which are not monitored in a typical server setting.

Show that the methodology here should be suitable across domains as all auditing systems follow similar architectural patterns

6.2 System Auditing

Several audit logging systems have been proposed or are in active use today [2, 10, 16, 18]. [18] leverages Linux Security Module to monitor operations on kernel data structures. [6] provides a general and secure framework for writing such provenance systems at the operating system level. Linux Audit framework [2] is the most practical and widely used. The framework provides a general logging infrastructure that allows the integration of plugins to extend the system.

These provenance systems have associated storage and runtime costs [7, 15]. They also suffer from a dependency explosion which slows down forensic analysis. Execution partioning techniques such as BEEP, ProTracer, Windows Reduction decompose long running applications into autonomous units of work and remove false dependencies to address the problem of dependency explosion. We use similar ideas in Ellipsis, but without needing to add any additional instrumentation. The well-formed nature of the real-time tasks further ensures correctness of these execution units (i.e templates).

In the case of resource-constrained cyber-physical systems, these costs are of greater significance as compared to typical server-based workloads. CPS systems tend to have lower available storage as well as stricter schedules, which make such costly solutions unsuitable.

Through our work, we want to adapt these general provenance systems for real-time systems, by reducing the associated storage and runtime costs as a start.

6.3 Real Time Scheduling

Talk about real time scheduling, how we take cues from real time garbage collectors to setup and understand the behavior of the audit system.

7 Conclusion

This project is awesome.

References

[1] Raspberry pi 4 model b specifications âĂŞ raspberry pi.

[2] System auditing red hat enterprise linux 6. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security_guide/chap-system_auditing.

[3] Upstream security releases 2020 automotive cybersecurity report and announces first automotive threat intelligence service.

[4] Iot proliferation and widespread 5g: A perfect botnet storm, Mar 2020.

[5] Altawy, R., and Youssef, A. M. Security tradeoffs in cyber physical systems: A case study survey on implantable medical devices. *IEEE Access 4* (2016), 959–979.

[6] Bates, A., Tian, D. J., Butler, K. R., and Moyer, T. Trustworthy whole-system provenance for the linux kernel. In *24th {USENIX} Security Symposium ({USENIX} Security 15)* (2015), pp. 319–334.

[7] Bates, A., Tian, D. J., Hernandez, G., Moyer, T., Butler, K. R. B., and Jaeger, T. Taming the costs of trustworthy provenance through policy reduction. *ACM Trans. Internet Technol. 17*, 4 (Sept. 2017).

[8] Chen, C., Mohan, S., Pellizzoni, R., Bobba, R. B., and Kiyavash, N. A novel side-channel in real-time schedulers. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (April 2019), pp. 90–102.

[9] for Devices, C., and Health, R. Urgent/11.

[10] Gehani, A., and Tariq, D. Spade: support for provenance auditing in distributed environments. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing* (2012), Springer, pp. 101–120.

[11] Humayed, A., Lin, J., Li, F., and Luo, B. Cyber-physical systems securityâĂŤa survey. *IEEE Internet of Things Journal 4*, 6 (Dec 2017), 1802–1831.

[12] Karim, M. E., and Phoha, V. V. Cyber-physical systems security. In *Applied Cyber-Physical Systems* (New York, NY, 2014), S. C. Suh, U. J. Tanik, J. N. Carbone, and A. Eroglu, Eds., Springer New York, pp. 75–83.

[13] Lee, K. H., Zhang, X., and Xu, D. High accuracy attack provenance via binary-based execution partition. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013* (2013), The Internet Society.

[14] Liu, C. L., and Layland, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM 20*, 1 (Jan. 1973), 46âĂŞ61.

[15] Ma, S., Zhai, J., Kwon, Y., Lee, K. H., Zhang, X., Ciocarlie, G., Gehani, A., Yegneswaran, V., Xu, D., and Jha, S. Kernel-supported cost-effective audit logging for causality tracking. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 241–254.

[16] Ma, S., Zhang, X., and Xu, D. Protracer: Towards practical provenance tracing by alternating between logging and tainting.

[17] Pellizzoni, R., Paryab, N., Yoon, M., Bak, S., Mohan, S., and Bobba, R. B. A generalized model for preventing information leakage in hard real-time systems. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium* (April 2015), pp. 271–282.

[18] Pohly, D. J., McLaughlin, S., McDaniel, P., and Butler, K. Hi-fi: Collecting high-fidelity whole-system provenance. In *Proceedings of the 28th Annual Computer Security Applications Conference* (New York, NY, USA, 2012), ACSAC âĂŹ12, Association for Computing Machinery, p. 259âĂŞ268.

[19] Rajkumar, R., Lee, I., Sha, L., and Stankovic, J. Cyber-physical systems: The next computing revolution. In *Design Automation Conference* (June 2010), pp. 731–736.

[20] Sadeghi, A., Wachsmann, C., and Waidner, M. Security and privacy challenges in industrial internet of things. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)* (June 2015), pp. 1–6.

[21] Team, A. D. Ardupilot Home.

[22] Xu, Z., Wu, Z., Li, Z., Jee, K., Rhee, J., Xiao, X., Xu, F., Wang, H., and Jiang, G. High fidelity data reduction for big data security dependency analyses. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS âĂŹ16, Association for Computing Machinery, p. 504âĂŞ516.

[23] Yoon, M., Mohan, S., Choi, J., Kim, J., and Sha, L. Securecore: A multicore-based intrusion detection architecture for real-time embedded systems. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)* (April 2013), pp. 21–32.

[24] Zetter, K. An unprecedented look at stuxnet, the world's first digital weapon, Jun 2017.

[25] Zimmer, C., Bhat, B., Mueller, F., and Mohan, S. Time-based intrusion detection in cyber-physical systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems* (New York, NY, USA, 2010), ICCPS âĂŹ10, Association for Computing Machinery, p. 109âĂŞ118.