Measuring System Auditing Overheads in Real-Time Systems

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

Abstract

Auditing allows system operators to observe and gain insights from general purpose computing systems. The information obtained by auditing systems can be used to detect and explain unexpected behavior ranging from fault diagnosis to intrusion detection and forensics after security incidents. While such mechanisms would be beneficial for many Real-Time Systems (RTS), existing audit frameworks are rarely designed to work in this domain. If audit systems are not intergrated into real-time operating systems carefully, they can negatively impact the temporal behavior of such systems. In this paper, we attempt to understand how these integrations can be realised by measuring the performance characteristics of a commodity system audit framework (i.e. Linux Audit) using ArduPilot (an open-source autopilot application suite) as well as synthetic benchmarks.

1 Introduction

Fault detection and diagnostic tools has driven the development of a variety of event logging frameworks for many real-time operating systems, including Composite OS [55, 64], QNX [6] and VxWorks [10]. As RTS becomes increasingly important in safety and security critical domains such as medical devices, autonomous vehicles, unmanned aeronautical vehicles, critical infrastructure and smart cities [27, 38, 52, 61], the need for effective auditing support will only grow in the near future. At present, vehicle collisions and crash investigations rely on event data recorders (or black boxes) to determine fault and liability [18–20, 48].

However, with the rise in popularity, today's RTS have become prime targets for sophisticated attackers [24]. Exploits in RTS can enable vehicle hijacks [22, 32], manufacturing disruptions [62], IoT botnets [28], subversion of life-saving medical devices [16, 69] and many other devastating attacks. The COVID-19 pandemic has further shed light on the potential damage of attacks on medical infrastructure [63]. These threats are not theoretical in nature, but an active and ongoing threat, as evidence by Russian attempts to take control of nuclear power, water and electric systems throughout the United States and Europe [59] and the Stuxnet attack on Iranian nuclear centrifuges [74].

In traditional computing systems, system auditing has proven to be crucial to detecting, investigating and responding to advanced intrusions. In contrast to application layer logging that is now widely used in RTS [6, 10, 55, 64], system auditing takes away the responsibility of logging from the developer and provides a unified view of system behavior. System level logs can be parsed into a connected graph based on shared dependencies of individual events, facilitating causal analysis over the history of events within a system [15, 35, 47, 51, 58]. This capability helps security analysts trace suspicious activities to the point that the vast majority of security analysts consider audit logs to be the most crucial resource when investigating security threats [21]. Thus, we observe that auditing can help in (a) fault detection/diagnosis and (b) understanding and detecting security events.

While Linux Audit has been incorporated in the Embedded Linux distribution [5], the widespread adoption of system auditing in RTS is stymied by the poor performance of auditing frameworks, which are known to add tremendous temporal and storage overheads [45]. In a time sensitive RTS, it may be impractical to introduce such sources of temporal variance. For instance, consider an airbag deployment system in a modern automobile that has only 50 milliseconds to fully deploy the airbag after it senses a collision. As Linux Audit is known to introduce an overhead of above 40% to applications, a naive deployment of auditing can result in personal injury to passengers in the car. Further, auditing frameworks leverage asynchronous communication methods that require the use of kernel buffers to share information between various components of the audit subsystem. Given that numerous RTS might be resource constrained, race condition attacks against audit frameworks [54] are also more likely. Thus, one needs to account for managing the entire lifecycle of an audit record, from audit generation to the time the audit record gets written to disk or sent out over the network to ensure proper functioning of the system and complete audit log collection. The design and implementation of Linux Audit is described in greater detail in Section 2.

In summary, our work attempts to understand how system auditing can be incorporated into the schedule of RTS (§3) with a focus on the audit log maintenance overheads. We highlight issues with naively enabling auditing(§5) and suggest future directions to enable efficient auditing on RTS.

2 Linux Audit Framework

The Linux Audit system [66] provides a way to observe and analyse system activities. While Linux Audit can be configured to monitor high-level activities such as login attempts

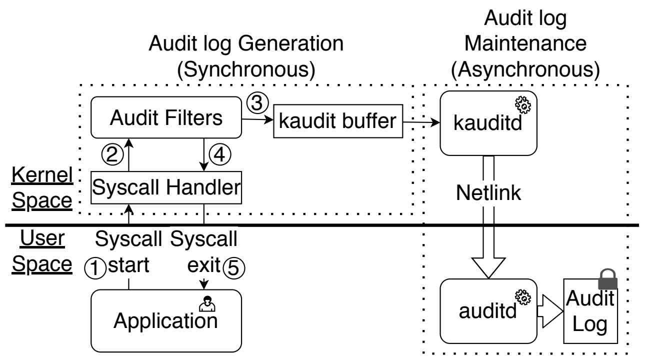

Figure 1. Architecture of Linux Audit Framework. Audit logs are generated using auditing hooks in kernel's system call (Syscall) handler and temporarily stored in the kaudit buffer. Log maintenance is handled by two background daemons, kauditd and auditd.

[9], its primary utility (and overhead) comes from tracking low-level system calls, which is the focus of this paper. An overview of the Linux Audit architecture is presented in Figure 1. When an application invokes a system call (①), the subsequent kernel control flow eventually traverses an `audit_filter` hook (②). Linux Audit examines the context of the event and compares it to pre-configured audit rules, generates a new log event and enqueues it in a message buffer if there is a match (③) before returning control to the system call handler (④) and then to the application (⑤). Asynchronous from this workflow, a pair of (non-real-time) audit daemons, `kauditd` and `auditd`, that run in kernel and user spaces respectively, empty the message buffer to user space for storage, distribution and analysis. Because the transport of logs is asynchronous, it is possible for the kaudit buffer to overflow if system calls occur faster than the daemon flushes to user space, creating the potential for event loss.

3 Real-Time Scheduling

We now provide a brief overview of real-time scheduling under auditing using the two representative timelines given in Figure 2. The baseline scenario shows execution timeline for two periodic tasks that are not under audit. Audit overheads can be divided into two parts: (i) part $A(\cdot)$ represents the task execution with additional synchronous overhead of log generation and (ii) part $B(\cdot)$ represents the processing time required to maintain the audit logs, transporting them from kaudit buffer to userspace and eventually to persistent storage (as shown in Figure 1). Audit log maintenance (Task $B(\cdot)$) is composed of `kauditd` and `auditd` daemons that run with background priority.

Task $B(\cdot)$ varies with the number of log events that need to be maintained. Any additional overheads of log maintenance,

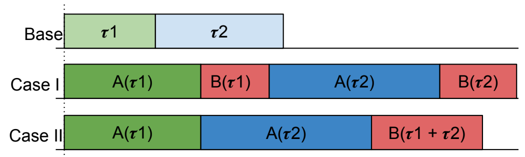

Figure 2. Sample timelines for two periodic tasks. *Base* is without auditing. *Case I and II* show potential schedules with auditing related overheads. $A(\cdot)$ is the increased computation time of the task including audit log generation overhead. $B(\cdot)$ represents the runtime of *kauditd* and *auditd* daemons as they handle the logs generated by the real-time tasks.

like transporting them to a remote server can also be trivially included in this component.

Depending on the scheduling algorithm, two forms of interleaving of the tasks with the audit daemons (i.e., Task $B(\cdot)$) are possible, as shown in Figure 2: execute audit maintenance tasks individually with each task (Case I), or execute audit maintenance task once for all tasks (Case II).

Case I is especially suitable for memory constrained embedded systems. To provide lossless auditing the kaudit buffer only as large as the maximum number of logs generated by any one instance of any task in the task set is required. However, Case I imposes far stringent temporal requirements for Task $B(\cdot)$, inheriting the priority of the real time task being audited. Case II relaxes the temporal requirement, allowing Task $B(\cdot)$ to run with the lowest real-time priority. However, to provide lossless auditing, it requires a much larger kaudit buffer, large enough to store all audit logs for one hyper period of the task set.

For either scenario, we require estimates of the following metrics for computing the schedule with auditing enabled: (i) synchronous overhead of log generation and (ii) asynchronous overheads for log maintenance per system call event. Our work focuses on estimating costs associated with (ii) while prior work [45] already provides a rough estimate of synchronous overheads due to Linux Audit.

4 Measurement Setup

All measurements are conducted on the experimental setup described in Table 1. Audit rules[1] were obtained from prior work [26, 30, 43, 46, 53, 67, 73] and were configured to capture system call events from our benchmarking applications only (*i.e.* background processes were not audited). The kaudit buffer size was configured to the maximum value that allowed stable application execution. To reduce the risk of external perturbations, we disable power management and force the processor to run at the highest frequency.

[1]Specifically, our rule set audits `execve, read, readv, write, writev, sendto, recvfrom, sendmsg, recvmsg, mmap, mprotect, link, symlink, clone, fork, vfork, open, close, creat, openat, mknodat, mknod, dup, dup2, dup3, bind, accept, accept4, connect, rename, setuid, setreuid, setresuid, chmod, fchmod, pipe, pipe2, truncate, ftruncate, sendfile, unlink, unlinkat, socketpair,splice, init_module`, and `finit_module`.

Table 1. Evaluation Platform

Platform	Raspberry Pi 4 [7]
Processor	ARM Cortex A-72 4 core [1]
Memory	4 GB
OS	Linux 4.19 [41]
Patches	Preempt RT [68]
Kernel Source	raspberrypi/linux [8]
Application	Ardupilot [12]

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 [42]

4.1 Real-Time Task Model

In this paper, we consider a multi-core system with M identical cores, running real-time applications in a preemptive operating system (*e.g.*, Linux). The system consists of N real-time tasks and scheduled using a fixed-priority scheduling policy. We also assume that the system is schedulable, *i.e.*, the worst-case response time (WCRT) for each task is less than or equal to its deadline.

There can be other non-real-time tasks scheduled by other non-real-time schedulers in the system. In most modern operating systems, non-real-time tasks only get to run during the slack time (*i.e.*, when no real-time tasks are in the run queue) and hence have little impact on the real-time task schedule. It's worth noting that a "task" (the term that is typically used in the Real-Time community) corresponds to a "thread" in Linux systems and these two terms are used interchangeably in this paper.

4.2 Tooling

In order to measure the asynchronous overheads of auditing as mentioned in Section 3, we first attempted to trace the behavior of audit threads âĂŤ kauditd and auditd, using kernel tracing tools âĂŤ ftrace [65] and eBPF [4].

Ftrace is an internal tracer that helps developers get visibility into the Linux kernel. It is typically used for debugging or analyzing latencies and performance issues that take outside the user space. But on evaluating ftrace, we found the latencies introduced by the tool to be significant as compared to the actual overheads. This was counter-productive to our goals of measuring asynchronous overheads in high fidelity.

BCC [3] is a toolkit for creating efficient kernel tracing and manipulation programs built upon eBPF. It includes tools that help measure function latency and counts with low overhead among other functionality. But on further analysis, we found that eBPF and the Linux real-time patch were incompatible [25], thus making BCC unsuitable for our study.

In light of these limitations, we chose to instrument the audit subsystem to capture high resolution event times. We first increased the resolution of the log generation time to nanoseconds by modifying the kernel component and introduced a new timestamp in the audit log, referred to as a *commit timestamp*, by modifying the audit userspace utility before the audit log files were written to the disk.

The difference between the two timestamps for a given event gives us visibility into the audit latency per event which includes a portion of the synchronous audit overhead along with the entire asynchronous overhead. Further, considering Case II from Section 3, the log maintenance overheads for system call events can be measured using the difference between commit timestamps of consecutive events.

4.3 Benchmarks

For obtaining measurements from a realistic RTS, we chose ArduPilot as our real world benchmark. ArduPilot is an open source autopilot application that can fully control various classes of autonomous vehicles such as quadcopter, rovers, submarines and fixed wing planes [12]. It has been installed in over a million vehicles and has been the basis for many industrial and academic projects. We chose the quadcopter variant of ArduPilot as it has the most stringent temporal requirements within the application suite.

But we found a real-world application to be a poor choice for a benchmark for scenarios which required fine-grained control over the frequency of log generation and overall system load. This required the development of a synthetic benchmark consisting of a multi-threaded application generating a user configured sequence of *getpid* system calls. The *getpid* system call was a natural choice as it has a low runtime overhead and doesn't cause significant side effects in the system.

5 Measurements

In the following section we discuss the measurements that we performed as part of our study. We start out by measuring the kaudit buffer utilization and audit latency to highlight the pitfalls of naively enabling auditing on a RTS. We then adapt the audit system to real-time scheduling as described in Section 3 and measure the log maintenance overhead.

5.1 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 3, we see that for Linux Audit, the utilization of the kaudit buffer rises quickly in the 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

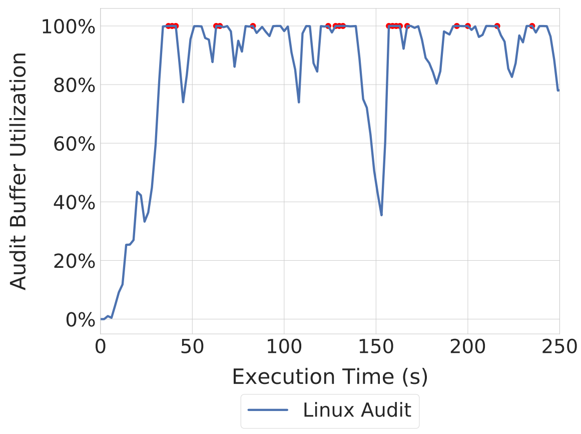

Figure 3. Kaudit buffer utilization over time for Linux Audit. Linux Audit's `backlog` value was measured every 2 seconds during the execution of the application and plotted as a percentage of the max backlog limit. Additional red annotations signify all times when buffer utilization was 100% and hence audit events might be lost. At these points Linux Audit performance is actually worse than depicted, but was constrained by the allotted buffer space.

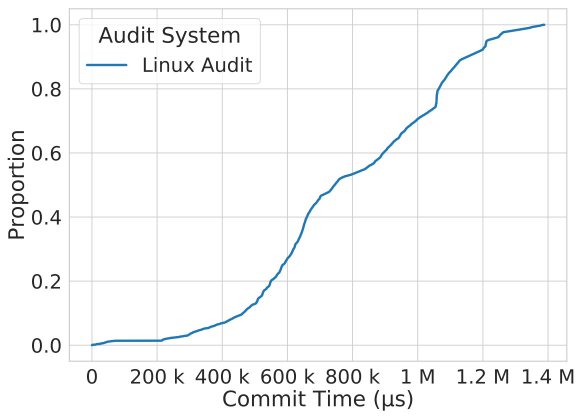

Figure 4. Audit commit latencies over time for Linux Audit plotted as a Cumulative Distribution Function (CDF). The experiment measures the time between the generation of an audit message and before the message is written to disk in user space by the *auditd* thread. For a chosen commit time *t* on the x-axis, the y-axis shows the fraction of the total messages that encountered delay of less than *t* microseconds.

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 scheme like KCAL [45] or increasing the rate of draining the kaudit buffer could be suitable approaches to ensure lossless auditing in resource constrained systems.

5.2 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 using the commit time and the creation time of each audit message obtained from the audit log file. The non-real-time *kauditd* and *auditd* threads are assigned defaults nice value of 0 and -4 respectively.

Observations: From Figure 4, 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.

5.3 Log Maintenance Overhead

Experiment:

For measuring the amount of time required to process a single audit message by the asynchronous audit threads, we ran a micro benchmark 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, we replicate the scenario described in Case II of Section 3 by assigning both kernel threads a low real time priority. We further pinned the *auditd* thread to core 2 on the machine. As *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

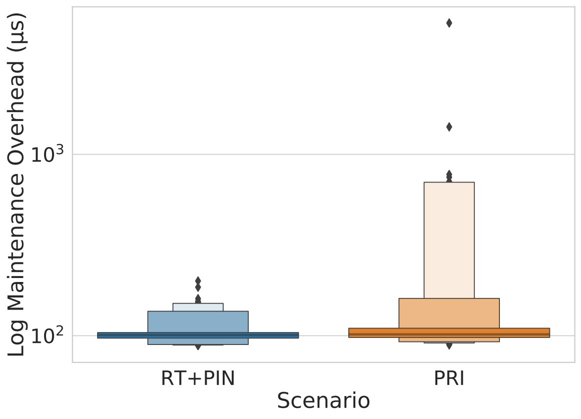

Figure 5. Log maintenance overhead in microseconds over different system configurations. For *RT+PIN*, we assign low real time priorities to the audit threads and pin *auditd* to core 2. For *PRI* we assign low non-real time priorities to both audit threads, similar to the default configuration for the audit system.

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 âĂŤ where only the user space daemon gets a priority boost.

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

Discussion: The order of magnitude reduction in maintenance overhead corresponding to a system call event highlights the competitions for processor time in the system. Assigning a low real time priority to both threads and pinning the user space *auditd* thread to a core, reduces this competition and allows the audit subsystem to occupy all the slack time in the system, thereby eliminating delays due to preemption and context switching.
A naive approach to ensure lossless auditing could be to introduce an additional core in the system dedicated to auditing tasks. If the real time application has enough slack to process audit logs generated in a hyper period, we can assign a low real-time priority to both audit threads to avoid additional hardware resources. The estimates for audit processing times obtained from the experiment can help arrive at a lower bound on the required slack in the system. For real time systems which undergo rigorous schedulability analysis before deployment, choosing either approach can be a conscious decision based on the timing profile of the system.

5.4 Summary of Results

Based on the measurements, we find that naively enabling auditing on a real-time system can have serious implications on the completeness of audit logs. Log reduction techniques can help us ensure completeness of audit logs and accounting for synchronous and asynchronous overheads introduced by auditing frameworks can help reliably schedule real-time tasks.

6 Discussion

6.1 Audit Log Reduction

Our findings show that reducing the rate of incoming audit messages is an important factor in achieving lossless audit logging. Future work can focus on evaluating the applicability of existing kernel based log reduction methods [45, 47] and propose novel solutions that account for the periodic and repetitive nature of real-time tasks to reduce rate of audit log generation, improving kernel buffer utilization and reducing storage costs.

6.2 Schedulability Analysis

In order to demonstrate that our estimates for auditing overheads are reasonably accurate, future work can subject our task model to a rigorous schedulability analysis using randomized workloads. Such an analysis would also enable us to compare the relative impact of enhancements to the audit system and understand the trade off between temporal integrity and completeness of audit logs.

6.3 Log Storage

Our study assumes that audit logs are written to disk, and thus measures the commit time of the audit logs accordingly. Linux Audit supports usage of customizable plugins that enable pushing the logs over the network to a centralized repository for archival and analysis [2]. The network writes may introduce variability in the asynchronous overheads, which would need to be accounted for when deploying a real application.

6.4 Wider Applicability

Our study measures the asynchronous performance of the Linux Audit framework. Nevertheless, we find that asynchronous logging mechanisms are common and also apply to other auditing mechanisms we are aware [15, 44, 47, 49, 50]. In fact, block-based I/O operations are always asynchronous within kernels unless explicitly configured for synchronous operation. Still, future work is need to explore the applicability of our observations on other platforms.

7 Related Work

7.1 System Auditing

Due to its value in threat detection and investigation, system auditing is the subject of renewed interest in traditional systems. While a number of experimental audit frameworks have incorporated notions of data provenance [15],[56],[60],[57], the bulk of this work is also based on commodity audit frameworks such as Linux Audit. Techniques have also been proposed to extract threat intelligence from large volumes of log data [30, 33, 37, 58]. In this work, we seek to understand if such techniques are directly applicable to RTS while adhering to the unique temporal and resource constraints of the domain. Our study finds that existing auditing solutions may not be directly suited for RTS and there is a need for solutions that reduce the amount of audit logs generated by building on the notion of execution partitioning of log activity [30, 31, 37, 39, 46] to remove redundant data.

7.2 Forensic Reduction

A lot of work has gone into improving the cost-utility ratio of system auditing by pruning, summarizing, or compressing audit data that is unlikely to be used during investigations [13, 14, 17, 29, 34, 40, 44, 67, 73]. Of these, Ma et al.'s KCAL [45] and ProTracer [47] systems are a few that inline their reduction methods into the kernel. These approaches only consider the impact of synchronous auditing overheads on system performance, but in order to design a long-running RTS, we need to understand system behavior over long runs which includes the log maintenance overhead. Our findings suggest that userspace reduction solutions might not be effective in RTS as resource constraints might cause loss of audit information in the kernel. Although it is unclear if the existing kernel based approaches are suitable for real-time applications, inline kernel log reduction seems to provide a promising path forward to enable efficient system auditing in RTS.

7.3 Auditing RTS

Although auditing has been widely acknowledged as an important aspect of securing embedded devices [11, 23, 36], the unique challenges that arise when auditing RTS have received only limited attention. Wang et al. present ProvThings, an auditing framework for monitoring IoT smart home deployments [71], but rather than audit low-level embedded device activity their system monitors API-layer flows on the IoT platform's cloud backend. Tian et al. present a block-layer auditing framework for portable USB storage that can be used to diagnose integrity violations [70]. Their embedded device emulates a USB flash drive, but does not consider system call auditing of real-time applications. Wu et al. present a network-layer auditing platform that captures the temporal properties of network flows and can thus detect temporal interference [72]. Whereas their system uses auditing to diagnose performance problems in networks, the present study considers the performance problems created by auditing within real-time applications. In contrast to these systems, this study highlights the challenges of RTS auditing by attempting to naively incorporate the vanilla Linux Audit system into the real-time task schedule.

8 Conclusion

In this work we discuss challenges of incorporating Linux Audit in a real-time schedule of an RTS. We show that a naively enabling auditing is not sufficient and a synergistic co-design is required in order to enable system auditing on RTS in the future.

9 Metadata

The presentation of the project can be found at:

https://zoom/cloud/link/

The code/data of the project can be found at:

https://github.com/anantk17/rt-audit-measurement

References

[1] ARM Cortex A-72. https://developer.arm.com/ip-products/processors/cortex-a/cortex-a72.

[2] audisp-remote(8) - linux man page. https://linux.die.net/man/8/audisp-remote. Last accessed 12-04-2020.

[3] BCC. https://github.com/iovisor/bcc. Last accessed 12-04-2020.

[4] eBPF. https://ebpf.io/. Last accessed 12-04-2020.

[5] Embedded linux. https://elinux.org/Main_Page.

[6] The instrumented microkernel. https://www.qnx.com/developers/docs/6.4.1/neutrino/sys_arch/trace.html.

[7] Raspberry Pi 4 Model B specifications - Raspberry Pi.

[8] Raspberry Pi Linux 4.19 Preempt RT. https://github.com/raspberrypi/linux/tree/rpi-4.19.y-rt.

[9] System auditing. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security_guide/chap-system_auditing#sec-audit_system_architecture.

[10] Tracealyzer for vxworks. https://percepio.com/docs/VxWorks/manual/.

[11] Anderson, M. Securing embedded linux. https://elinux.org/images/5/54/Manderson4.pdf.

[12] ArduPilot Development Team and Community. ArduPilot. https://ardupilot.org/, 2020.

[13] Bates, A., Butler, K. R., and Moyer, T. Take only what you need: Leveraging mandatory access control policy to reduce provenance storage costs. In *7th USENIX Workshop on the Theory and Practice of Provenance (TaPP 15)* (Edinburgh, Scotland, July 2015), USENIX Association.

[14] Bates, A., Tian, D., Hernandez, G., Moyer, T., Butler, K. R., and Jaeger, T. Taming the Costs of Trustworthy Provenance through Policy Reduction. *ACM Trans. on Internet Technology 17*, 4 (sep 2017), 34:1–34:21.

[15] 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)* (Washington, D.C., Aug. 2015), USENIX Association, pp. 319–334.

[16] Begg, R. Step up cyber hygiene: Secure access to medical devices. https://www.machinedesign.com/medical-design/article/21128232/step-up-cyber-hygiene-secure-access-to-medical-devices.

[17] Ben, Y., Han, Y., Cai, N., An, W., and Xu, Z. T-tracker: Compressing system audit log by taint tracking. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)* (Dec 2018), pp. 1–9.

[18] BIGELOW, P. Tesla/ntsb feud shows complications of crash investigations involving autonomous systems. https://www.caranddriver.com/news/a19785733/teslantsb-feud-shows-complications-of-crash-investigations-involving-autonomous-systems/.

[19] Böhm, K., Kubjatko, T., Paula, D., and Schweiger, H.-G. New developments on edr (event data recorder) for automated vehicles. *Open Engineering 10*, 1 (2020), 140–146.

[20] Bose, U. The black box solution to autonomous liability. *Wash. UL Rev. 92* (2014), 1325.

[21] Cimpanu, C. Hackers are increasingly destroying logs to hide attacks. https://www.zdnet.com/article/hackers-are-increasingly-destroying-logs-to-hide-attacks/. Last accessed 04-20-2019.

[22] Crane, C. Automotive cyber security: A crash course on protecting cars against hackers. https://www.thesslstore.com/blog/automotive-cyber-security-a-crash-course-on-protecting-cars-against-hackers/.

[23] Day, R., and Slonosky, M. Securing connected embedded devices using built-in rtos security.

[24] Department of Homeland Security. Cyber physical systems security. https://www.dhs.gov/science-and-technology/cpssec.

[25] Edge, J. Bpf and the realtime patch set. https://lwn.net/Articles/802884/. Last accessed 12-04-2020.

[26] Gehani, A., and Tariq, D. SPADE: Support for Provenance Auditing in Distributed Environments. In *Proceedings of the 13th International Middleware Conference* (Dec 2012), Middleware '12.

[27] Gurgen, L., Gunalp, O., Benazzouz, Y., and Gallissot, M. Self-aware cyber-physical systems and applications in smart buildings and cities. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2013), IEEE, pp. 1149–1154.

[28] Hahad, M. Iot proliferation and widespread 5g: A perfect botnet storm. https://www.scmagazine.com/home/opinion/executive-insight/iot-proliferation-and-widespread-5g-a-perfect-botnet-storm/.

[29] Hassan, W. U., Aguse, N., Lemay, M., Moyer, T., and Bates, A. Towards Scalable Cluster Auditing through Grammatical Inference over Provenance Graphs. In *Proceedings of the 25th ISOC Network and Distributed System Security Symposium* (San Diego, CA, USA, February 2018), NDSS'18.

[30] Hassan, W. U., Guo, S., Li, D., Chen, Z., Jee, K., Li, Z., and Bates, A. NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage. In *26th ISOC Network and Distributed System Security Symposium* (February 2019), NDSS'19.

[31] Hassan, W. U., Noureddine, M., Datta, P., and Bates, A. OmegaLog: High-Fidelity Attack Investigation via Transparent Multi-layer Log Analysis. In *27th ISOC Network and Distributed System Security Symposium* (February 2020), NDSS'20.

[32] Hayes, J. Hackers under the hood. https://eandt.theiet.org/content/articles/2020/03/hackers-under-the-hood/.

[33] Hossain, M. N., Milajerdi, S. M., Wang, J., Eshete, B., Gjomemo, R., Sekar, R., Stoller, S., and Venkatakrishnan, V. SLEUTH: Real-time attack scenario reconstruction from COTS audit data. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC, Aug. 2017), USENIX Association, pp. 487–504.

[34] Hossain, M. N., Wang, J., Sekar, R., and Stoller, S. D. Dependence-preserving data compaction for scalable forensic analysis. In *Proceedings of the 27th USENIX Conference on Security Symposium* (USA, 2018), SEC'18, USENIX Association, pp. 1723–1740.

[35] King, S. T., and Chen, P. M. Backtracking intrusions. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 223–236.

[36] Kohei, K. Recent security features and issues in embedded systems.

[37] Kwon, Y., Wang, F., Wang, W., Lee, K. H., Lee, W.-C., Ma, S., Zhang, X., Xu, D., Jha, S., Ciocarlie, G., Gehani, A., and Yegneswaran, V. Mci: Modeling-based causality inference in audit logging for attack investigation. In *Proc. of the 25th Network and Distributed System Security Symposium (NDSS'18)* (2018).

[38] Lee, I., Sokolsky, O., Chen, S., Hatcliff, J., Jee, E., Kim, B., King, A., Mullen-Fortino, M., Park, S., Roederer, A., et al. Challenges and research directions in medical cyber–physical systems. *Proceedings of the IEEE 100*, 1 (2011), 75–90.

[39] Lee, K. H., Zhang, X., and Xu, D. High Accuracy Attack Provenance via Binary-based Execution Partition. In *Proceedings of NDSS '13* (Feb. 2013).

[40] Lee, K. H., Zhang, X., and Xu, D. Loggc: Garbage collecting audit log. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (New York, NY, USA, 2013), CCS '13, Association for Computing Machinery, pp. 1005–1016.

[41] Linux Kernel. The Linux Kernel Archives.

[42] Linux Kernel Organization, Inc. Cpu frequency and voltage scaling code in the linux kernel. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

[43] Liu, Y., Zhang, M., Li, D., Jee, K., Li, Z., Wu, Z., Rhee, J., and Mittal, P. Towards a Timely Causality Analysis for Enterprise Security. In *Proceedings of the 25th ISOC Network and Distributed System Security Symposium* (San Diego, CA, USA, February 2018), NDSS'18.

[44] Ma, S., Lee, K. H., Kim, C. H., Rhee, J., Zhang, X., and Xu, D. Accurate, low cost and instrumentation-free security audit logging for windows. In *Proceedings of the 31st Annual Computer Security Applications Conference* (New York, NY, USA, 2015), ACSAC 2015, Association for Computing Machinery, pp. 401–410.

[45] 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.

[46] Ma, S., Zhai, J., Wang, F., Lee, K. H., Zhang, X., and Xu, D. MPI: Multiple Perspective Attack Investigation with Semantic Aware Execution Partitioning. In *26th USENIX Security Symposium* (August 2017).

[47] Ma, S., Zhang, X., and Xu, D. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *Proceedings of NDSS '16* (2016).

[48] Martyn, A. Tesla blames drivers who wreck its cars but wonâĂŹt hand over crash data without a court order. https://www.consumeraffairs.com/news/tesla-blames-drivers-who-wreck-its-cars-but-wont-hand-over-crash-data-without-a-court-order-053018.html.

[49] Microsoft. Process monitor v3.60. https://docs.microsoft.com/en-us/sysinternals/downloads/procmon. Last accessed 12-04-2020.

[50] Microsoft. Windows etw framework conceptual tutorial. https://docs.microsoft.com/en-us/windows/win32/wes/windows-event-log. Last accessed 12-04-2020.

[51] Milajerdi, S. M., Gjomemo, R., Eshete, B., Sekar, R., and Venkatakrishnan, V. Holmes: Real-time apt detection through correlation of suspicious information flows. In *2019 2019 IEEE Symposium on Security and Privacy (SP)* (Los Alamitos, CA, USA, may 2019), IEEE Computer Society.

[52] Monostori, L., Kádár, B., Bauernhansl, T., Kondoh, S., Kumara, S., Reinhart, G., Sauer, O., Schuh, G., Sihn, W., and Ueda, K. Cyber-physical systems in manufacturing. *Cirp Annals 65*, 2 (2016), 621–641.

[53] Paccagnella, R., Datta, P., Hassan, W. U., Bates, A., Fletcher, C. W., Miller, A., and Tian, D. Custos: Practical Tamper-Evident

Auditing of Operating Systems Using Trusted Execution. In *27th ISOC Network and Distributed System Security Symposium* (February 2020), NDSS'20.

[54] Paccagnella, R., Liao, K., Tian, D. J., and Bates, A. Logging to the Danger Zone: Race Condition Attacks and Defenses on System Audit Frameworks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (2020), CCS'20.

[55] Parmer, G. A. *Composite: A component-based operating system for predictable and dependable computing.* Boston University, 2010.

[56] Pasquier, T., Han, X., Goldstein, M., Moyer, T., Eyers, D., Seltzer, M., and Bacon, J. Practical whole-system provenance capture. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), pp. 405–418.

[57] Pasquier, T. F. J. ., Singh, J., Eyers, D., and Bacon, J. Camflow: Managed data-sharing for cloud services. *IEEE Transactions on Cloud Computing 5*, 3 (2017), 472–484.

[58] Pei, K., Gu, Z., Saltaformaggio, B., Ma, S., Wang, F., Zhang, Z., Si, L., Zhang, X., and Xu, D. Hercule: Attack story reconstruction via community discovery on correlated log graph. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (New York, NY, USA, 2016), ACSAC '16, Association for Computing Machinery, pp. 583–595.

[59] Perlroth, N., and Sanger, D. E. Cyberattacks Put Russian Fingers on the Switch at Power Plants, U.S. Says. https://www.nytimes.com/2018/03/15/us/politics/russia-cyberattacks.html, 2018.

[60] 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 '12, Association for Computing Machinery, pp. 259–268.

[61] 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.

[62] Shepherd, D. Industry 4.0: the development of unique cybersecurity. https://www.manufacturingglobal.com/technology/industry-40-development-unique-cybersecurity.

[63] Slabodkin, G. Coronavirus chaos ripe for hackers to exploit medical device vulnerabilities. https://www.medtechdive.com/news/coronavirus-chaos-ripe-for-hackers-to-exploit-medical-device-vulnerabilitie/575717/.

[64] Song, J., and Parmer, G. C'MON: a predictable monitoring infrastructure for system-level latent fault detection and recovery. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium* (2015), IEEE, pp. 247–258.

[65] Steven Rostedt. ftrace - function tracer. https://www.kernel.org/doc/html/latest/trace/ftrace.html. Last accessed 12-04-2020.

[66] SUSE LINUXAG. Linux Audit-Subsystem De-sign Documentation for Linux Kernel 2.6, v0.1. Available at http://uniforum.chi.il.us/slides/HardeningLinux/LAuS-Design.pdf, 2004.

[67] Tang, Y., Li, D., Li, Z., Zhang, M., Jee, K., Xiao, X., Wu, Z., Rhee, J., Xu, F., and Li, Q. Nodemerge: Template based efficient data reduction for big-data causality analysis. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2018), CCS '18, Association for Computing Machinery, pp. 1324–1337.

[68] The Linux Foundation. Real-time linux. https://wiki.linuxfoundation.org/realtime/start, 2020.

[69] The MITRE Corporation. Medical device cybersecurity. https://www.mitre.org/sites/default/files/publications/pr-18-1550-Medical-Device-Cybersecurity-Playbook.pdf.

[70] Tian, D. J., Bates, A., Butler, K. R. B., and Rangaswami, R. Provusb: Block-level provenance-based data protection for usb storage devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, Oct 2016), CCS '16, ACM.

[71] Wang, Q., Hassan, W. U., Bates, A., and Gunter, C. Fear and Logging in the Internet of Things. In *Proceedings of the 25th ISOC Network and Distributed System Security Symposium* (February 2017), NDSS'18.

[72] Wu, Y., Chen, A., and Phan, L. T. X. Zeno: Diagnosing performance problems with temporal provenance. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, 2019), USENIX Association, pp. 395–420.

[73] 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 '16, Association for Computing Machinery, pp. 504–516.

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