Towards Efficient Auditing for Real-Time Systems

Ayoosh Bansal[1][0000−0002−4848−6850], Anant Kandikuppa[1][1111−2222−3333−4444], Chien-Ying Chen[1], Monowar Hasan[2][0000−0002−2657−0402], Adam Bates[1], and Sibin Mohan[3][0000−0002−3295−0233]

[1] University of Illinois Urbana-Champaign, Urbana-Champaign IL 61801, USA
{ayooshb2, anantk3, cchen140, batesa}@illinois.edu
[2] Wichita State University, Wichita KS 67260, USA
monowar.hasan@wichita.edu
[3] The George Washington University, Washington DC 20052, USA
sibin.mohan@gwu.edu

Abstract. System auditing is a powerful tool that provides insight into the nature of suspicious events in computing systems, allowing machine operators to detect and subsequently investigate security incidents. While auditing has proven invaluable to the security of traditional computers, existing audit frameworks are rarely designed with consideration for Real-Time Systems (RTS). The transparency provided by system auditing would be of tremendous benefit in a variety of security-critical RTS domains, (*e.g.,* autonomous vehicles); however, if audit mechanisms are not carefully integrated into RTS, auditing can be rendered ineffectual and violate the real-world temporal requirements of the RTS.
In this paper, we demonstrate how to adapt commodity audit frameworks to RTS. Using Linux Audit as a case study, we first demonstrate that the volume of audit events generated by commodity frameworks is unsustainable within the temporal and resource constraints of real-time (RT) applications. To address this, we present *Ellipsis*, a set of kernel-based reduction techniques that leverage the periodic repetitive nature of RT applications to aggressively reduce the costs of system-level auditing. *Ellipsis* generates succinct descriptions of RT applications' expected activity while retaining a detailed record of unexpected activities, enabling analysis of suspicious activity while meeting temporal constraints. Our evaluation of *Ellipsis*, using ArduPilot (an open-source autopilot application suite) demonstrates *up to 93% reduction* in audit log generation.

Keywords: Real-time systems · Auditing · Cyber-physical systems

1 Introduction

As RTS become indispensable in safety- and security-critical domains — medical devices, autonomous vehicles, manufacturing automation, smart cities, etc. [29,41,53,58] — the need for effective and precise *auditing* support is growing. Even now, event data recorders (or *black boxes*) are crucial for determining fault and liability when investigating vehicle collisions [16,17], and the need

for diagnostic event logging frameworks (*e.g.,* QNX [4], VxWorks [5] and Composite OS [62]) is well understood. However, these high-level event loggers are insufficient to detect and investigate sophisticated attacks. Concomitant with its explosive growth, today's RTS have become ripe targets for sophisticated attackers [27]. Exploits in RTS can enable vehicle hijacks [25,36], manufacturing disruptions [60], IoT botnets [31], subversion of life-saving medical devices [67] and many other devastating attacks. The COVID-19 pandemic has further shed light on the potential damage of attacks on medical infrastructure [14,61]. These threats are not theoretical, rather active and ongoing, as evidenced recently by malicious attempts to take control of nuclear power, water and electric systems throughout the United States and Europe [55].

In traditional computing systems, *system auditing* has proven crucial to detecting, investigating and responding to intrusions [20,33,34,52]. Unfortunately, comprehensive system auditing approaches are not widely used in RTS. RTS logging takes place largely at the *application layer* [16,17] or performs lightweight system layer tracing for performance profiling (*e.g.,* log syscall occurrences, but not arguments) [18]; in both cases, the information recorded is insufficient to trace attacks because the causal links between different system entities cannot be identified. The likely cause of this hesitance to embrace holistic system-layer logging is poor performance. System audit frameworks are known to impose tremendous computational and storage overheads [51] that are incompatible with the temporal requirements of many real-time applications. Thus, while we are encouraged by the growing recognition of the importance of embedded system auditing [8,26,38] and the newfound availability of Linux Audit in the Embedded Linux [3], a practical approach to RTS auditing remains an elusive goal.

Observing that performance cost of Linux Audit is ultimately dependent on the number of log events generated, and that the performance impacts of commodity auditing frameworks can be optimized without affecting the forensic validity of the audit logs, *e.g.,* through carefully reducing the number of events that need to be logged [11,13,15,32,43,47,51,65,74], we set out to tailor Linux Audit to RTS, carefully reducing event logging without impacting the forensic validity of the log. We present **Ellipsis**, a kernel-based log reduction framework that leverages the predictability of real-time tasksets' execution profiles. *Ellipsis* first profiles tasks to produce a template of their audit footprint. At runtime, behaviors consistent with this template are reduced, while any deviations from the template are audited in full, without reduction. Far from being impractical, we demonstrate a synergistic relationship between security auditing and predictable RTS workloads – *Ellipsis* is able to faithfully audit suspicious activities while incurring almost no log generation during benign typical activity.

The **contributions** of this work are:

- *Ellipsis*[1], an audit framework, uniquely-tailored to RT environments (§3).
- Detailed[2] evaluations (§4) and security analysis (§5) to demonstrate that *Ellipsis* retains relevant information while reducing event/log volume.

[1] https://bitbucket.org/sts-lab/ellipsis

[2] A technical report with supplementary material for this work is available [10].

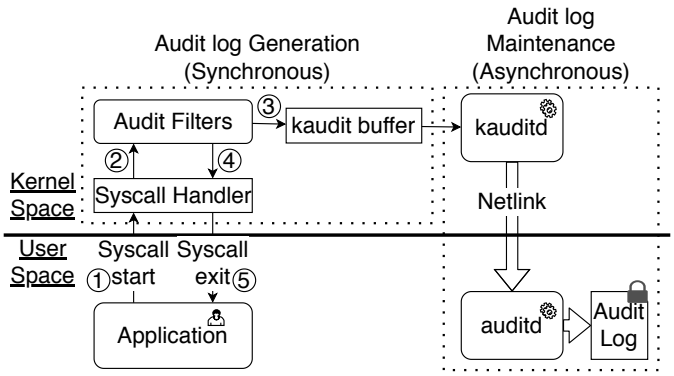

Fig. 1. Architecture of Linux Audit Framework [1].

2 Background and System Model

Linux Audit Framework. The Linux Audit system [64] provides a way to audit system activities. An overview of the Linux Audit architecture is presented in Fig. 1. When an application invokes a syscall ①, the subsequent kernel control flow eventually traverses an `audit_filter` hook ②. Linux Audit examines the context of the event, compares it to pre-configured audit rules, generating a log event if there is a match and enqueueing it in a message buffer ③ before returning control to the syscall handler ④ and then to the application ⑤. Asynchronous from this workflow, a pair of (non-RT) audit daemons (`kauditd` and `auditd`) transmit the message buffer to user space for storage and analysis. Because the daemons are asynchronous, the message buffer can overflow if syscalls occur faster than the daemon flushes to user space, resulting in event loss.

Although it is well-established that Linux Audit can incur large computational and storage overheads in traditional software [51], its impacts on RT applications were unclear. Encouragingly, upon conducting a detailed analysis [2] we observed that Linux Audit does not introduce significant issues of priority inversion or contention over auditing resources shared across applications (*e.g.,* `kaudit` buffer). Further, except for limited outlier cases, the latency introduced by auditing syscalls can be measured and bounded. Hence it is a good candidate for firm and soft deadline RTS as supported by RT Linux [66]. However, audit events were lost, making auditing incomplete and ineffectual while still costly for the RTS due to large storage space required to store the audit log.

RTS Properties. *Ellipsis* leverages properties unique to RT environments, as described in Table 1. In contrast to traditional applications where determining all possible execution paths is often undecidable, knowledge about execution paths is an essential component of RT application development. RTS are special purpose machines that execute well formed tasksets to fulfill predetermined tasks. Various techniques are employed to analyze the tasksets, *e.g.,* worst case execution time (WCET) analysis [19,30,35,45,57,59,76]. All expected behaviors

Table 1. RTS properties relevant to *Ellipsis*

Property	**Relevance to *Ellipsis***
Periodic tasks	Most RT tasks are periodically activated, leading to repeating behaviors. *Ellipsis* templates describe the most common repetitions.
Aperiodic tasks	Second most common form of RT tasks, Aperiodic tasks also lead to repeating behaviors, but with irregular inter-arrival times.
Code Coverage	High code coverage analyses are part of existing RTS development processes, *Ellipsis*' automated template generation adds minimal cost.
Timing Predictability	A requirement for safety and correct functioning of RTS, naïvely enabling auditing can violate this by introducing overheads and variability.
Isolation	Resources are commonly isolated in RTS to improve timing predictability. RTS auditing mechanisms should not violate resource isolation.
Special Purpose	RTS are special purpose machines, tasks are known at development *i.e.,* templates can be created before system deployment.
Longevity	Once deployed RTS can remain functional for years. *Ellipsis*' can save enormous log storage and transmission costs over the lifetime of the RTS.

of the system must be accounted for at design time in conjunction with the system designers. Any deviation is an unforeseen fault or malicious activity, which needs to be audited in full detail.

Threat Model. We consider an adversary that aims to penetrate and impact an RTS through exfiltrating data, corrupting actuation outputs, degrading performance, causing deadline violations, etc.. This attacker may install modified programs, exploit a running process or install malware on the RTS to achieve their objectives. To observe this attacker, our system adopts an aggressive audit configuration intended to capture all forensically-relevant events, as identified in prior works.[3] We assume that the underlying OS and the audit subsystem therein are trusted. This is a standard assumption in system auditing literature [12,33,46,48,56]. Far from being impractical on RTS, prior works such as Trusted Timely Computing Base provide a secure kernel that meets both the trust and temporal requirements for hosting *Ellipsis* in RT Linux [21,24,69,70]. *Ellipsis*' goal is to capture evidence of an attacker intrusion/activity without losing relevant information and hand it off to a tamper proof system. Although audit integrity is an important security goal, it is commonly explored orthogonally to other audit research due to the modularity of security solutions (*e.g.,* [12,54,75]). Therefore, we assume that once recorded to `kaudit buffer`, attackers cannot compromise the integrity of audit logs Finally, we assume that applications can be profiled in a controlled benign environment prior to being the target of attack.

[3] Specifically, our ruleset 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.`

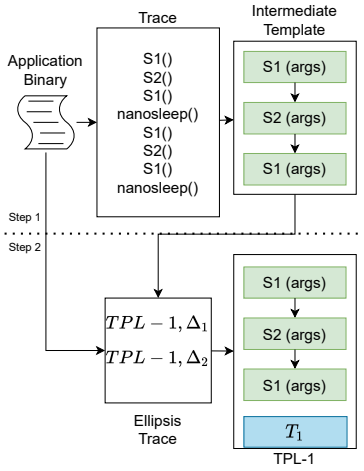

Fig. 2. *Ellipsis* template creation.

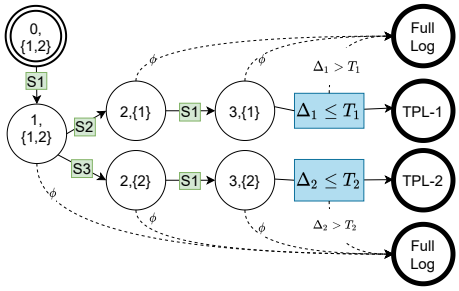

Fig. 3. Runtime template matching as an FSA with states as [syscalls matched count, {set of reachable templates}]. TPL-1 (S1, S2, S1) and TPL-2 (S1, S3, S1) are shown as example. Template matches (TPL-1, TPL-2) emit a single record, failure leads to full log store.

3 *Ellipsis*

The volume of audit events is the major limiting factor for auditing RTS. High event volume can result in event record loss, high log storage costs and large maintenance overheads [51]. We present *Ellipsis*, an audit event reduction technique designed specifically for RTS. *Ellipsis* achieves this through *templatization* of the audit event stream. Templates represent learned expected behaviors of RT tasks, described as a sequence of syscalls with arguments and temporal profile.These templates are generated in an offline profiling phase, similar to common RTS analyses like WCET [19,44]. At runtime, the application's syscall stream is compared against its templates; if a contiguous sequence of syscalls matches a template, only a single record indicating the template match is inserted into the event stream (`kaudit buffer`). Significantly, while a sequence of audited syscall events is replaced by a single record, relevant information is not lost (§5).

Model. Consider a system in which the machine operator wishes to audit a single RT task $\boldsymbol{\tau}$. An RT *task* here corresponds to a *thread* in Linux systems, identified by a combination of process and thread ids. We can limit this discussion to a single task, without losing generality, as *Ellipsis*' template creation, activation and runtime matching treat each task as independent. RT tasks are commonly structured with a one time *init* component and repeating *loops*. Let $\boldsymbol{s_i}$ denote a syscall sequence the task exhibits in a *loop* execution and $\boldsymbol{N}$ the count of different syscall execution paths τ might take (*i.e.,* $0 < i \leq N$). A *template* describes these sequences (s_i), identifying the syscalls and arguments. As noted in Section 2, RT applications are developed to have limited code paths and bounded loop iterations. Extensive analysis of execution paths is a standard part of the RTS development process. Thus, for RTS, N is finite and determinable.

Let function $\boldsymbol{len(s_i)}$ return the number of syscalls in the sequence s_i. Further, let $\boldsymbol{p_i}$ be the probability that an iteration of τ exhibits syscall sequence s_i.

Sequence Identification. The first step towards template creation is identification of sequences and their probability of occurrences. Identification of cyclic syscall behaviors has been addressed in the auditing literature [42,50], with past solutions require binary analysis, code annotations, stack analysis or a combination. While any technique that yields s_i and p_i can be employed here, including the prior mentioned ones, we developed a highly automated process, leveraging RT task structure and Linux Audit itself. The application is run for long periods of time and audit trace collected. We observe that RT tasks typically end with calls to `sleep` or `yield` that translate to `nanosleep` and `sched_yield` syscalls in Linux. Periodic behaviors can also be triggered by polling timerfds to read events from multiple timers by using `select` and `epoll_wait` syscalls. We leverage these syscalls to identify boundaries of task executions within the audit log and then extract sequences of syscall invocations. Figure 2 provides an overview of this process. We also modified Linux Audit to include the Thread ID in log messages helping disambiguate threads belonging to a process. This first step yields the per task syscall sequences exhibited by the application and their properties: length, probability of occurrence, and the arguments. These syscall sequences are then converted into intermediate templates, each entry of which includes the syscall name along with the arguments. This first step can also be iterated with intermediate templates loaded to reduce previously extracted sequences, though in practice such iterations were not required.

Sequence Selection. A subset of intermediate templates are chosen to be converted to final templates. This choice is based on the tradeoff between the benefit of audit event volume reduction and the memory cost as defined later in eq. (3) and (5), respectively. As we discuss in detail in Section 5 the security tradeoff is minimal. Let's assume $\boldsymbol{n}$ sequences are chosen to be reduced, where $0 \leq n \leq N$. As noted earlier, *Ellipsis* treats each task independently, the value of n is also independent for each task.

Template Creation. For the next step, Fig. 2 Step 2, these n templates are loaded and application profiled again to collect temporal profile for each template *i.e.,* the expected duration and inter-arrival intervals for each template. The intermediate templates are enriched with this temporal information, to yield the final templates. Templates are stored in the form of text files and occupy negligible disk space, *e.g.,* ArduPilot templates used for evaluation (§4) occupied 494 bytes of space on disk total. This whole process is highly automated, given an application binary with necessary inputs, using the template creation toolset.[1]

Ellipsis Activation. We extend the Linux Audit command-line `auditctl` utility to transmit templates to kernel space. Once templates are loaded, *Ellipsis* can be activated using `auditctl` to start reducing any matching behaviors. This extended `auditctl` can also be used to activate/deactivate *Ellipsis* and load/unload templates, however, these operations are privileged, identical to deactivating Linux Audit itself. System administrators can use this utility to easily update templates as required, *e.g.,* in response to application updates.

Table 2. Parameters from Case Study

Task Name	N	I	$len(s_i)$	p_i	f
arducopter	5	100	$[14, 15, 17, 17, 18]$	$[0.95, 0.02, 0.01, 0.01, 0.01]$	679
ap-rcin	1	182	$[16]$	$[1]$	2
ap-spi-0	5	1599	$[1, 1, 1, 2, 2]$	$[0.645, 0.182, 0.170, 0.001, 0.001]$	0

Runtime Matching. Given the template(s) of syscall sequences, an *Ellipsis* kernel module, extending from Linux Audit syscall hooks, filters syscalls that match a template. The templates are modeled as a finite state automaton (FSA), (Figure 3), implemented as a collection of linked lists in kernel memory. While the RT task is executing, all syscall sequences allowed by the automaton are stored in a temporary task-specific buffer. If the set of events fully describes an automaton template, *Ellipsis* discards the contents of the task-specific buffer and enqueues a single record onto the `kaudit` buffer to denote the execution of a templatized activity. Alternatively, *Ellipsis* enqueues the entire task-specific buffer to the main kaudit buffer if *(a)* a syscall occurs that is not allowed by the automaton, *(b)* the template is not fully described at the end of the task instance or *(c)* the task instance does not adhere to the expected temporal behavior of the fully described template. Thus, the behavior of each task instance is reduced to a single record when the task behaves as expected. For any abnormal behavior, the complete audit log is retained.

Audit Event Reduction. Let the task τ be executed for $\boldsymbol{I}$ iterations and $\boldsymbol{f}$ denote the number of audit events in *init* phase. The number of audit events generated by $\boldsymbol{\tau}$ when audited by Linux Audit $(\boldsymbol{E_A})$, when *Ellipsis* reduces n out of total N sequences $(\boldsymbol{E_E})$, and the reduction $(E_A - E_E)$ are given by

$$E_A = I * (\textstyle\sum_{i=1}^{N}(p_i * len(s_i))) + f \tag{1}$$

$$E_E = I * (\textstyle\sum_{i=1}^{n} p_i + \sum_{i=n+1}^{N}(p_i * len(s_i))) + f \tag{2}$$

Ellipsis' Event reduction — Audit events for n sequences

$$E_A - E_E = I * (\textstyle\sum_{i=1}^{n}(p_i * len(s_i)) - \sum_{i=1}^{n} p_i) \tag{3}$$

Iterations — *Ellipsis* events for n sequences

As evident from eq. (3), to maximize reduction, long sequences with large p_i values must be chosen as the n sequences for reduction. RT applications, like control systems, autonomous systems and even video streaming, feature limited execution paths for majority of their runtimes [39]. This property has been utilized by Yoon *et al.* in a prior work [76]. Therefore, for RT applications the distribution of p_i is highly biased *i.e.,* certain sequences s_i have high probability of occurrence. Table 2 provides example values for the parameters used, determined during the *Sequence Identification* step in template creation for the evaluation application ArduPilot(§4).

Storage Size Reduction. Let $\boldsymbol{B_A}$ denote the average cost of representing a syscall event in audit log and $\boldsymbol{B_E}$ denote the average cost of representing *Ellipsis*' template match record. By design, $B_E <= B_A$; B_E is a constant 343 bytes, while

B_A averaged 527 bytes (1220 max) in our evaluation. Noting that the init events (f) are not reduced by *Ellipsis*, the disk size reduction *i.e.,* difference in sizes of $\boldsymbol{\tau}$'s audit log for Linux Audit $(\boldsymbol{L_A})$ and *Ellipsis* $(\boldsymbol{L_E})$ is:

$$L_A - L_E = I * (B_A * \sum_{i=1}^{n}(p_i * len(s_i)) - B_E * \sum_{i=1}^{n} p_i) \tag{4}$$

From Equation (3) and (4), *Ellipsis*' benefits come from the audit events count and log size becoming independent of sequence size ($len(s_i)$) for the chosen n sequences, multiplied further by repetitions of these sequences ($I * p_i$). *Ellipsis* behaves identical to Linux Audit for any sequence that is not included as a template, *i.e.,* $i \geq n + 1$ in Equation (2).

Memory Tradeoff. The tradeoff for *Ellipsis*' benefits are computational overheads (evaluated in §4.5 and §4.6) and the memory cost of storing templates ($\boldsymbol{M_\tau}$). Let $\boldsymbol{M_{fixed}}$ be memory required per template, excluding syscalls, while $\boldsymbol{M_{syscall}}$ be the memory required for each syscall in the template. On 32 bit kernel $M_{fixed} = 116$ and $M_{syscall} = 56$ bytes, determined by *sizeof* data structures. As an example, 3 templates from evaluation occupied 2 KB in memory.

$$M_\tau = M_{fixed} * n + M_{syscall} * \sum_{i=1}^{n} len(s_i) \tag{5}$$

Extended Reduction Horizon. Until now we have limited the horizon of reduction to individual task *loop* instances. We can further optimize by creating a single record that describes multiple consecutive matches of a template. This higher performance system is henceforth referred to as **Ellipsis-HP**. When a *Ellipsis-HP* match fails, a separate record is logged for each of the base template matches along with complete log sequence for the current instance (*i.e.,* the base behavior of *Ellipsis*). *Ellipsis-HP* performs best when identical sequences occur continuously, capturing all sequence repetitions in one entry.

$$E^{Best}_{Ellipsis\text{-}HP} = n + I * \sum_{i=n+1}^{N}(p_i * len(s_i)) + f \tag{6}$$

4 Evaluation

We evaluate *Ellipsis* and *Ellipsis-HP* using ArduPilot [9], a safety-critical firm-deadline autopilot application. We show that our auditing systems *(a)* perform *lossless auditing within the application's temporal requirements*, where Linux Audit would lose audit events or violate application's safety constraints (§4.3), *(b)* achieve high audit log volume reduction during benign activity, *(c)* enjoy minimal computational overhead even in an artificially created worst case scenarios (§4.5). Using a set of synthetic tasks we also show that the *Ellipsis*' overhead per syscall scales independent of the size of template (§4.6).

4.1 Setup

All measurements were conducted on 4GB Raspberry Pi 4 running Linux 4.19. The RT kernel from raspberrypi/linux [2] was used with AUDIT and AUDITSYSCALL kconfigs enabled. To reduce computational variability due to external

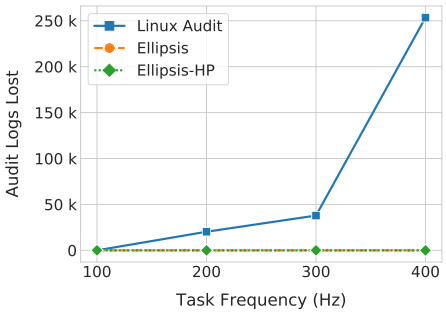

Fig. 4. (§4.3) Number of audit events lost vs. frequencies of the primary loop in ArduPilot, for 100K iterations.

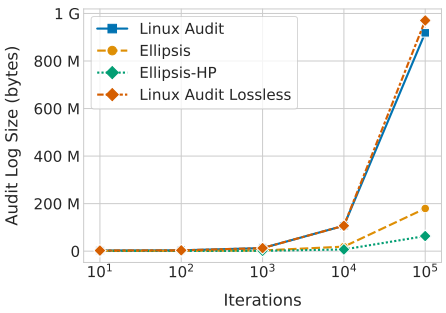

Fig. 5. (§4.4) Total size on disk of the audit log (Y-axis), captured for different number of iterations (X-axis).

perturbations we disabled power management, directed all kernel background tasks/interrupts to core 0 using the *isolcpu* kernel argument, and set CPU frequency Governor to Performance. Audit rules for capturing syscall events were configured to match against our benchmark application (*i.e.,* background process activity was not audited). We set the `kaudit` buffer size to 50K as any larger values led to system panic and hangs.

4.2 ArduPilot

ArduPilot is an open source autopilot application that can fully control various classes of autonomous vehicles such as quadcopters, rovers, submarines and fixed wing planes [9]. 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, called ArduCopter, as it has the most stringent temporal requirements within the application suite. For this application the RPi4 board was equipped with a Navio2 Autopilot hat [6] to provide real sensors and actuator interfaces for the application. We instrumented the application for measuring the runtime overheads introduced by auditing. Among the seven tasks spawned by ArduPilot, we focus primarily on a task named FastLoop for evaluating temporal overheads as it includes the stability and control tasks that need to run at a high frequency to keep the QuadCopter stable and safe.

Among the syscalls observed in the trace of ArduPilot, we found that only a small subset of syscalls were relevant to forensic analysis [28]: `execve, openat, read, write, close` and `pread64`. Upon running the template generation script on the application binary, we obtained the most frequently occuring templates for three tasks ($n = 1$, for each task), consisting of 14 `write`, 16 `pread64` calls and 1 `read` call, respectively. These templates include expected values corresponding to the file descriptor and count arguments of the syscalls. Templates were loaded into the kernel when evaluating *Ellipsis* or *Ellipsis-HP*.

4.3 Audit Completeness

Experiment. We ran the application for 100K iterations at task frequencies of 100 Hz, 200 Hz, 300 Hz and 400 Hz[4] , measuring audit events lost. The fast dynamics of a quadcopter benefit from the lower discretization error in the ArduPilot's PID controllers at higher frequencies [71] leading to more stable vehicle control.

Observations. Figure 4 compares the log event loss for Linux Audit, *Ellipsis* and *Ellipsis-HP* across multiple task frequencies. We observe that Linux Audit lost log events at all task frequencies above 100 Hz. In contrast, *Ellipsis* and *Ellipsis-HP* did not lose audit event log at any point in the experiment.

Discussion. Because this ArduCopter task performs critical stability and control function, reducing task frequency to accomodate Linux Audit may hay considerable detrimental effects. Further investigation revealed that Linux Audit dropped log events due to `kaudit` buffer overflow, despite the buffer size being 50K. In contrast, *Ellipsis* is able provide auditing for the entire frequency range without suffering log event loss. Better yet, throughout the experiment the max buffer occupancy was just 2.5K for *Ellipsis* and 1.5K for *Ellipsis-HP*.[2]

4.4 Audit Log Size Reduction

Experiment. We ran the ArduCopter application over multiple iterations in 10 to 100K range to simulate application behavior over varying runtimes. For each iteration count, we measure the size on disk of the recorded log.

Observations. Figure 5 compares the storage costs in terms of file size on disk in bytes. The storage costs for all systems over shorter runs was found to be comparable, as the cost of auditing the initialization phase of the application ($B_A * f$) tends to dominate over the periodic loops. Over a 250 second runtime (10^5 iterations) the growth of log size in *Ellipsis* was drastically lower compared to vanilla Linux Audit, with storage costs reducing by 740 MB, or **80%**, when using *Ellipsis*. *Ellipsis-HP* provides a more aggressive log size reduction option by lowering storage costs by 860MB, or **93%**, compared to Linux Audit. *Linux Audit Lossless* estimates the log size had Linux Audit not lost any log events.

Discussion. The observations line up with our initial hypothesis that the bulk of the audit logs generated during a loop iteration would exactly match the templates. Thus, in *Ellipsis* by reducing all the log messages that correspond to a template down to a single message, we see a vast reduction in storage costs while ensuring the retention of all the audit data. *Ellipsis-HP* takes this idea further by eliminating audit log generation over extended periods of time if the application exhibits expected behaviors only. For RTS that are expected to run for months or even years without failing, these savings are crucial for continuous and complete security audit of the system. Motion, a soft deadline application with numerous execution paths (N = 26) achieved similarly high reduction rations ($81\% - 98\%$) under varying configurations options and inputs.[2]

[4] Frequency values are chosen based on application support: https://ardupilot.org/copter/docs/parameters-Copter-stable-V4.1.0.html#sched-loop-rate-scheduling-main-loop-rate

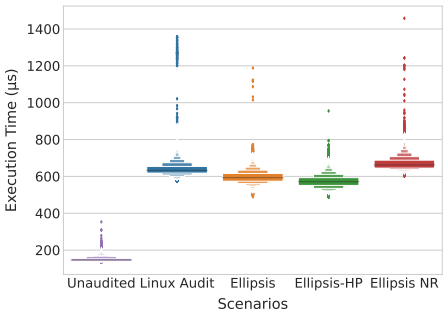

Fig. 6. (§4.5) Comparison of runtime overheads of ArduPilot main loop. Task period and deadline is 2500 μs.

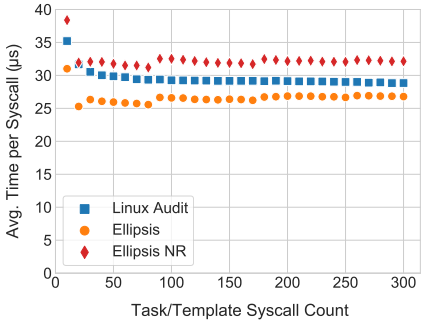

Fig. 7. (§4.6) Avg. execution latency of `getpid` syscall (Y-axis) with varying task/template lengths (X-axis)

4.5 ArduPilot: Runtime Overheads

Experiment. This evaluation measures the execution time in microseconds (μs), for the Fast Loop task of ArduPilot, for 1000 iterations, under various auditing setups. The small number of iterations kept the generated log volume within `kaudit buffer` capacity, avoiding overflows and audit events loss in any scenario. This avoids polluting the overhead data with instances of event loss. The time measurement is based on the monotonic timer counter. This process was repeated 100 times. To evaluate the absolute worst case for *Ellipsis*, the *Ellipsis* NR (No Reduction) scenario modifies the ArduCopter template so that it always fails at the last syscall. *Ellipsis* NR is also the worst case for *Ellipsis-HP*.

Observations. Figure 6 shows the distribution of 100 execution time samples for each scenario. *Ellipsis*, *Ellipsis-HP* and *Ellipsis* NR have nearly the same overhead as Linux Audit. On average, *Ellipsis*'s overhead is *0.93*x and *Ellipsis-HP*'s overhead is *0.90*x of Linux Audit. The observed maximum overheads show a greater improvement. *Ellipsis*'s observed maximum overhead is *0.87*x and *Ellipsis-HP*'s *0.70*x of Linux Audit. *Ellipsis* NR shows a *1.05*x increase in average overhead and *1.07*x increase in maximum observed overhead.

Discussion. Ellipsis adds additional code to syscall auditing hooks, which incurs small computational overheads. When template matches fail (*Ellipsis* NR), this additional overhead is visible, although the overhead is not significantly worse the baseline Linux Audit. However, in the common case where audit events are reduced by *Ellipsis*, this cost is masked by reducing the total amount of log collection and transmission work performed by Linux Audit. This effect is further amplified in *Ellipsis-HP* owing to its greater reduction potential (§4.4). Thus, *Ellipsis*'s runtime overhead depends on the proportion of audit information reduced in the target application. Thus, while reducing the runtime overhead of auditing is not *Ellipsis*' primary goal, it nonetheless enjoys a modest performance improvement by reducing the total work performed by the underlying audit framework.

4.6 Synthetic Tasks: Overhead Scaling

Experiment. Because *Ellipsis* adds template matching logic in the critical execution path of syscalls, a potential concern is the overhead growth for tasks with long syscall sequences. In this experiment we measure execution time for tasks that execute varying counts of `getpid` syscalls (10, 20, 30 ... 300). `getpid` is a low latency non-blocking syscall, which allows us to stress-test the auditing framework. As the max template length (*i.e.,* syscall count) observed in real application loops was 29, we analyze workloads of roughly 10 times that amount, *i.e.,* 300. The execution time for each task is measured 100 times. Since the tasks have a single execution path *i.e.,* a fixed count of `getpid` syscalls, *Ellipsis*' audit events reduction always succeeds. For *Ellipsis* NR (No Reduction) we force template matches to fail at the last entry (same as §4.5).

Observations. Figure 7 shows the average syscall response time as the number of syscalls in the task loop increases. The primary observation of interest is that the *time to execute a syscall is roughly constant*, independent of the number of syscalls in the task and template. The higher value at the start is due to the non syscall part of the task that quickly becomes insignificant for tasks with higher number of syscalls. We only show average latency as the variance is negligible ($< 1.3\mu s$).

Discussion. Ellipsis scales well as the overhead per syscall remains independent of template size, even in the worst case scenario of *Ellipsis* NR. When log reduction succeeds the overhead is reduced. When the log reduction fails the overhead is not significantly worse than Linux Audit.

4.7 Summary of Results

Ellipsis provides complete audit events retention while meeting temporal requirements of the ArduPilot application, with significantly reduced storage costs. *Ellipsis-HP* further improves the reduction ratios. The temporal constraint allows additional temporal checks, detecting anomalous latency spikes with effectively no additional log size overhead during normal operation.[2]

5 Security Analysis

The security goal of *Ellipsis*, indeed auditing in general, is to record all forensically-relevant information, thereby aiding in the investigation of suspicious activities. The previous section established *Ellipsis*' ability to dramatically reduce audit event generation for benign activities, freeing up auditing capacity. We now discuss the security implications of *Ellipsis*.

Stealthy Evasion. If a malicious process adheres to the expected behavior of benign tasks, the associated logs will be reduced. The question, then, is whether a malicious process can perform meaningful actions while adhering to the benign templates. If *Ellipsis* exclusively matched against syscall IDs only, such a feat may be possible; however, *Ellipsis* also validates syscalls' arguments

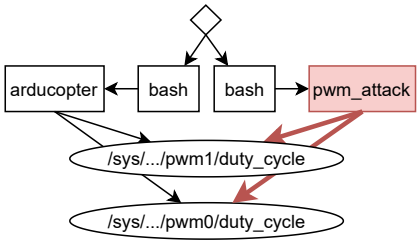

Fig. 8. (§5) Attack graph created using *Ellipsis* audit logs.

and temporal constraints, effectively validating both the *control flow* and *data flow* before templatization. Thus making it exceedingly difficult for a process to match a template while affecting the RTS in any meaningful way. For example, an attacker might try to substitute a read from a regular file with a read from a sensitive file; however, doing so would require changing the file handle argument, failing the template match. Thus, at a minimum *Ellipsis* provides comparable security to commodity audit frameworks, and may actually provide improved security by avoiding the common problem of log event loss. A positive side effect of *Ellipsis* is built in partitioning of execution flows, benefiting provenance techniques that utilize such partitions [42,49,50].

Information Loss. Another concern is whether *Ellipsis* templates remove forensically-relevant information. The following is an example `write` as would be recorded by Linux Audit.

```
type=SYSCALL msg=audit(1601405431.612391366:5893333): arch=40000028 syscall=4
 per=800000 success=yes exit=7 a0=4 a1=126ab0 a2=1 a3=3 items=0 ppid=1513 pid
=1526 tid=1526 auid=1000 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0
fsgid=0 tty=pts0 ses=1 comm="arducopter" exe="/home/pi/ardupilot/build/navio2
/bin/arducopter" key=(null)
```

The record above, if reduced with *Ellipsis* and reconstructed using the *Ellipsis* log and templates, yields:

```
type=SYSCALL msg=audit([1601405431.612391356, 1601405431.612391367]:∅): arch
=40000028 syscall=4 per=800000 success=yes exit=7 a0=4 a1=∅ a2=1 a3=∅ items=0
 ppid=1513 pid=1526 tid=1526 auid=1000 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid
=0 sgid=0 fsgid=0 tty=pts0 ses=1 comm="arducopter" exe="/home/pi/ardupilot/
build/navio2/bin/arducopter" key=(null)
```

∅ denotes values that could not be reconstructed and [min, max] denote where a range is known but not the exact value. Nearly all of the information in an audit record can be completely reconstructed, including *(a)* all audit events executed by a task, in order of execution, *(b)* forensically relevant arguments. On the other hand, information not reconstructed is *(a)* accurate timestamps, *(b)* a monotonically increasing audit ID, *(c)* forensically irrelevant syscall arguments. The effect of this lost information is that fine grained inter-task event ordering and interleaving cannot be reconstructed. This loss of information is minimal and at worst increases the size of attack graph of a malicious event. We now demonstrate *Ellipsis*'s ability to retain forensically relevant information.

Demonstration: Throttle Override Attack. Autopilot applications are responsible for the safe operation of autonomous vehicles. ArduPilot periodically updates actuation signals that control rotary speed of motors that power rotors. The periodic updates are responsible for maintaining vehicle stability and safety.

Attack Scenario. Let's consider a stealthy attacker who wants to destabilize or take control over the unmanned drones. To achieve this, the attacker first gains control of a task on the system and attempts to override the control signals. An actuation signal's effect depends on the duration for which it controls the vehicle, therefore, naïvely overriding an actuation signal is not a very effective attack as the control task may soon update it to the correct value, reducing the attack's effect. The attacker instead leverages side channel attacks such as Scheduleak [23] during the reconnaissance phase of the attack to learn when the control signals are updated. Armed with this knowledge, the attacker overrides the actuation signals immediately after the original updates, effectively taking complete control, with little computational overhead. We use the ArduPilot setup as in described earlier (§4.2). Using tools provided with Scheduleak [23], a malicious task is able to override actuation signals generated by ArduPilot. This setup is run for 250 seconds and audit logs collected with *Ellipsis*.

Results. Overriding throttle control signals involves writing to files in `sysfs`. This attack behavior can be observed in audit logs as sequences of `openat`, `write` and `close` syscalls. Combining templates with the obtained audit log yields the attack graph in Figure 8. *Ellipsis* correctly identifies that ArduPilot is only exhibiting benign behaviors, reducing its audit logs. *Ellipsis* preserves detailed attack behaviors for the malicious syscall sequences. *Ellipsis* did not lose audit events throughout the application runtime. In contrast, Linux Audit loses audit events (§4.3), potentially losing critical forensic evidence.

Discussion. Scheduleak [23] invokes `clock_gettime` syscall frequently to infer task activation times. Such syscalls are irrelevant for commonly used forensic analysis as they don't capture critical information flows. Despite the lack of visibility in the reconnaissance phase of the attack, auditing can capture evidence of attacker interference that creates new information flows, as shown in Figure 8. We have demonstrated that when a process deviates from the expected behaviors, *e.g.,* due to an attack, *Ellipsis* provides the same security as Linux Audit. Additionally, *Ellipsis* all but eliminates the possibility of losing portions of the malicious activity due to `kaudit` buffer overflow. However, it is impossible to guarantee that no events will ever be lost with malicious activities creating unbounded new events. *Ellipsis* improves upon Linux Audit by *(a)* freeing up auditing resources which can then audit malicious behaviors, and *(b)* reducing the audit records from benign activities that must be analyzed as part of forensic provenance analysis. Stealthy attacks like this also show the role of auditing in improving vulnerability detection and forensic analysis on RTS.

6 Discussion

System Scope & Limitations. *Ellipsis* is useful for any application that has predictable repeating patterns. When sequence sets are too large with no high probability sequences, it may be possible that too much of system memory would be required to achieve significant log reduction. That said, a large number of possible sequences is not detrimental to *Ellipsis* as long as there exist some high probability sequences. *Ellipsis*'s efficacy is also not dependent on specific scheduling policies unless tasks share process and thread ids; if task share process/thread ids and the scheduler can reorder them, *Ellipsis* cannot distinguish between event chains, leading to unnecessary template match failures.

Auditing Hard RTS. *Ellipsis*, like Linux Audit and Linux itself, is unsuitable for hard-deadline RTS. All synchronous audit components must meet the temporal requirements for Hard RTS with bounded WCET, including syscall hooks and *Ellipsis* template matching. Additionally the `kaudit buffer` occupancy must have a strict upper bound. In this paper *Ellipsis* takes a long step forward, deriving high confidence empirical bounds (§4.5) to enable *Ellipsis*' use in firm- or soft-deadline RTS, which are prolific [7]. However, the strict bounds required for Hard RTS are a work in progress.

Unfavorable Conditions. We consider here the impact of using *Ellipsis* to audit hypothetical RTS where our assumptions about RTS properties do not hold. If the RTS may execute previously unknown syscall sequences, extra events would exist in the audit log. The audit log recorded by *Ellipsis* would thus be larger. Since safety, reliability and timing predictability are important requirements for RTS [7] the gaps in code coverage can only be small. Hence the unknown syscall sequences will not have a major impact on audit events and log size. If known syscall sequences have near uniform probability of occurrence, simply using templates for them all achieves high reduction ($n = N$). The tradeoff is additional memory required to store templates which is a small cost (Eq. (5)). Finally, if the above are combined, sequences with substantial probability of occurrence would remain untested during the RTS development. For such a system, functional correctness, reliability, safety or timing predictability cannot be established, making this RTS unusable.

7 Related Work

Auditing RTS. Although auditing has been widely acknowledged as an important aspect of securing embedded devices [8,26,38], challenges unique to auditing RTS have received limited attention. Wang et al. present ProvThings, an auditing framework for monitoring IoT smart home deployments [72], 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 [68]. Their embedded device emulates a USB flash drive, but does not consider syscall auditing of RT applications. Wu et al. present a network-layer

auditing platform that captures the temporal properties of network flows and can thus detect temporal interference [73]. Whereas their system uses auditing to diagnose performance problems in networks, the presented study considers the performance problems created by auditing within real-time applications.

Forensic Reduction. Significant effort has been dedicated to improving the cost-utility ratio for system auditing by pruning or compressing audit data that is unlikely to be of use during investigations [11,13,15,22,32,37,43,47,63,65,74]. However these approached address the log storage overheads and not the voluminous event generation that is prohibitive to RTS auditing (§4.3). KCAL [51] and ProTracer [49] systems are among the few that, like *Ellipsis*, inline their reduction methods into the kernel. Regardless of their layer of operation, these approaches are often based on an observation that certain log semantics are not forensically relevant (*e.g.,* temporary file I/O [43]), but it is unclear whether these assumptions hold for real-time cyber-physical environments, *e.g.,* KCAL or ProTracer would reduce multiple identical reads syscalls to a single entry. However, a large number of extra reads can cause catastrophic deadline misses. Forensic reduction in RTS, therefore, needs to be cognizant of the characteristics of RTS or valuable information can be lost. Our approach to template generation in *Ellipsis* shares similarities with the notion of *execution partitioning* of log activity [33,34,40,42,50], which decomposes long-lived applications into autonomous units of work to reduce false dependencies in forensic investigations. Unlike past systems, however, our approach requires no instrumentation to facilitate. Further, leveraging the well-formed nature of real-time tasks ensures the correctness of our execution units *i.e.,* templates.

8 Conclusion

Ellipsis is a novel audit event reduction system that exemplifies synergistic application-aware co-design of security mechanisms for RTS. *Ellipsis* allows RT applications to be audited while meeting the temporal requirements of the application. The role of auditing in securing real-time applications can now be explored and enhanced further. As showcased with Auditing in this work, other security mechanisms from general purpose systems warrant a deeper analysis for their use in RTS.[5]

Acknowledgements The material presented in this paper is based upon work supported by the Office of Naval Research (ONR) under grant number N00014-17-1-2889 and the National Science Foundation (NSF) under grant numbers CNS 1750024, CNS 1932529, CNS 1955228, CNS 2055127, CNS 2145787 and CNS 2152768. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

[5] A technical report with further evaluations, template examples, security demonstrations and expanded RTS properties survey is available [10].

References

1. System auditing (2018), https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security_guide/chap-system_auditing
2. Raspberry Pi Linux 4.19 Preempt RT (2019), https://github.com/raspberrypi/linux/tree/rpi-4.19.y-rt
3. Embedded linux (2020), https://elinux.org/Main_Page
4. The instrumented microkernel (2020), https://www.qnx.com/developers/docs/6.4.1/neutrino/sys_arch/trace.html
5. Tracealyzer for vxworks (2020), https://percepio.com/docs/VxWorks/manual/
6. Navio2 board (2021), https://navio2.emlid.com/
7. Akesson, B., et al.: An empirical survey-based study into industry practice in real-time systems. In: IEEE Real-Time Systems Symposium. IEEE (2020)
8. Anderson, M.: Securing embedded linux (2020), https://elinux.org/images/5/54/Manderson4.pdf
9. ArduPilot Development Team and Community: Ardupilot (2020), https://ardupilot.org/
10. Bansal, A., et al.: Ellipsis: Towards efficient system auditing for real-time systems (2022). https://doi.org/10.48550/ARXIV.2208.02699
11. Bates, A., et al.: Take Only What You Need: Leveraging Mandatory Access Control Policy to Reduce Provenance Storage Costs. In: 7th Workshop on the Theory and Practice of Provenance. TaPP'15 (Jul 2015)
12. Bates, A., et al.: Trustworthy Whole-System Provenance for the Linux Kernel. In: Proceedings of 24th USENIX Security Symposium (Aug 2015)
13. Bates, A., et al.: Taming the Costs of Trustworthy Provenance through Policy Reduction. ACM Trans. on Internet Technology **17**(4), 34:1–34:21 (sep 2017)
14. Begg, R.: Step up cyber hygiene: Secure access to medical devices (2020), https://www.machinedesign.com/medical-design/article/21128232/step-up-cyber-hygiene-secure-access-to-medical-devices
15. Ben, Y., et al.: T-tracker: Compressing system audit log by taint tracking. In: 2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS). pp. 1–9 (Dec 2018)
16. Böhm, K., et al.: New developments on edr (event data recorder) for automated vehicles. Open Engineering **10**(1), 140–146 (2020)
17. Bose, U.: The black box solution to autonomous liability. Wash. UL Rev. (2014)
18. Brandenburg, B., Anderson, J.: Feather-trace: A lightweight event tracing toolkit. In: Proceedings of the third international workshop on operating systems platforms for embedded real-time applications. pp. 19–28 (2007)
19. Burguiere, C., Rochange, C.: History-based schemes and implicit path enumeration. In: 6th International Workshop on Worst-Case Execution Time Analysis (WCET'06). Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2006)
20. Carbon Black: Global incident response threat report. https://www.carbonblack.com/global-incident-response-threat-report/november-2018/ (November 2018), last accessed 04-20-2019
21. Casimiro, A., et al.: How to build a timely computing base using real-time linux. In: 2000 IEEE International Workshop on Factory Communication Systems. Proceedings (Cat. No. 00TH8531). pp. 127–134. IEEE (2000)
22. Chen, C., et al.: Distributed provenance compression. In: Proceedings of the 2017 ACM International Conference on Management of Data. pp. 203–218 (2017)

23. Chen, C.Y., et al.: Schedule-based side-channel attack in fixed-priority real-time systems. Tech. rep. (2015)
24. Correia, M., et al.: The design of a cots real-time distributed security kernel. In: European Dependable Computing Conference. pp. 234–252. Springer (2002)
25. Crane, C.: Automotive cyber security: A crash course on protecting cars against hackers (2020), https://www.thesslstore.com/blog/automotive-cyber-security-a-crash-course-on-protecting-cars-against-hackers/
26. Day, R., Slonosky, M.: Securing connected embedded devices using built-in rtos security (2020), http://mil-embedded.com/articles/securing-connected-embedded-devices-using-built-in-rtos-security/
27. Department of Homeland Security: Cyber physical systems security (2020), https://www.dhs.gov/science-and-technology/cpssec
28. Gehani, A., Tariq, D.: SPADE: Support for Provenance Auditing in Distributed Environments. In: Proceedings of the 13th International Middleware Conference. Middleware '12 (Dec 2012)
29. Gurgen, L., et al.: Self-aware cyber-physical systems and applications in smart buildings and cities. In: 2013 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1149–1154. IEEE (2013)
30. Gustafsson, J., Ermedahl, A.: Experiences from applying wcet analysis in industrial settings. In: 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing. pp. 382–392. IEEE (2007)
31. Hahad, M.: Iot proliferation and widespread 5g: A perfect botnet storm (2020), https://www.scmagazine.com/home/opinion/executive-insight/iot-proliferation-and-widespread-5g-a-perfect-botnet-storm/
32. Hassan, W.U., et al.: Towards Scalable Cluster Auditing through Grammatical Inference over Provenance Graphs. In: Proceedings of the 25th ISOC Network and Distributed System Security Symposium. NDSS'18, San Diego, CA, USA (February 2018)
33. Hassan, W.U., et al.: NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage. In: 26th ISOC Network and Distributed System Security Symposium. NDSS'19 (February 2019)
34. Hassan, W.U., et al.: OmegaLog: High-Fidelity Attack Investigation via Transparent Multi-layer Log Analysis. In: 27th ISOC Network and Distributed System Security Symposium. NDSS'20 (February 2020)
35. Hatton, L.: Safer language subsets: an overview and a case history, misra c. Information and Software Technology **46**(7), 465–472 (2004)
36. Hayes, J.: Hackers under the hood (2020), https://eandt.theiet.org/content/articles/2020/03/hackers-under-the-hood/
37. Hossain, M.N., et al.: Dependence-preserving data compaction for scalable forensic analysis. In: Proceedings of the 27th USENIX Conference on Security Symposium. pp. 1723–1740. SEC'18, USENIX Association, Berkeley, CA, USA (2018)
38. Kohei, K.: Recent security features and issues in embedded systems (2020), https://elinux.org/images/e/e2/ELC2008_KaiGai.pdf
39. Konrad, S., Cheng, B.H.: Real-time specification patterns. In: Proceedings of the 27th international conference on Software engineering. pp. 372–381 (2005)
40. Kwon, Y., et al.: 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)
41. Lee, I., et al.: Challenges and research directions in medical cyber–physical systems. Proceedings of the IEEE **100**(1), 75–90 (2011)

42. Lee, K.H., et al.: High Accuracy Attack Provenance via Binary-based Execution Partition. In: Proceedings of NDSS '13 (Feb 2013)
43. Lee, K.H., et al.: LogGC: Garbage Collecting Audit Log. In: Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security. pp. 1005–1016. CCS '13, ACM, New York, NY, USA (2013)
44. Li, Y.T.S., Malik, S.: Performance analysis of embedded software using implicit path enumeration. In: Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems. pp. 88–98 (1995)
45. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM **20**(1), 46–61 (Jan 1973)
46. Liu, Y., et al.: Towards a timely causality analysis for enterprise security. In: NDSS (2018)
47. Ma, S., et al.: Accurate, low cost and instrumentation-free security audit logging for windows. In: Proceedings of the 31st Annual Computer Security Applications Conference. pp. 401–410. ACSAC 2015, ACM, New York, NY, USA (2015)
48. Ma, S., et al.: Protracer: Towards practical provenance tracing by alternating between logging and tainting. In: NDSS (2016)
49. Ma, S., et al.: ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In: Proceedings of NDSS '16 (Feb 2016)
50. Ma, S., et al.: MPI: Multiple Perspective Attack Investigation with Semantic Aware Execution Partitioning. In: 26th USENIX Security Symposium (August 2017)
51. Ma, S., et al.: Kernel-supported cost-effective audit logging for causality tracking. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18). pp. 241–254. USENIX Association, Boston, MA (2018)
52. Milajerdi, S.M., et al.: Holmes: Real-time apt detection through correlation of suspicious information flows. In: 2019 2019 IEEE Symposium on Security and Privacy (SP). IEEE Computer Society, Los Alamitos, CA, USA (may 2019)
53. Monostori, L., et al.: Cyber-physical systems in manufacturing. Cirp Annals **65**(2), 621–641 (2016)
54. Paccagnella, R., et al.: Custos: Practical Tamper-Evident Auditing of Operating Systems Using Trusted Execution. In: 27th ISOC Network and Distributed System Security Symposium. NDSS'20 (February 2020)
55. Perlroth, N., 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)
56. Pohly, D., et al.: Hi-Fi: Collecting High-Fidelity Whole-System Provenance. In: Proceedings of the 2012 Annual Computer Security Applications Conference. ACSAC '12, Orlando, FL, USA (2012)
57. Puschner, P., Burns, A.: Writing temporally predictable code. In: Proceedings of the Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems.(WORDS 2002). pp. 85–91. IEEE (2002)
58. Rajkumar, R., et al.: Cyber-physical systems: the next computing revolution. In: Design Automation Conference. pp. 731–736. IEEE (2010)
59. Sandell, D., et al.: Static timing analysis of real-time operating system code. In: International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. pp. 146–160. Springer (2004)
60. Shepherd, D.: Industry 4.0: the development of unique cybersecurity (2020), https://www.manufacturingglobal.com/technology/industry-40-development-unique-cybersecurity

61. Slabodkin, G.: Coronavirus chaos ripe for hackers to exploit medical device vulnerabilities (2020), https://www.medtechdive.com/news/coronavirus-chaos-ripe-for-hackers-to-exploit-medical-device-vulnerabilitie/575717/
62. Song, J., 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. pp. 247–258. IEEE (2015)
63. Sundaram, V., et al.: Prius: Generic hybrid trace compression for wireless sensor networks. In: Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems. pp. 183–196 (2012)
64. SUSE LINUXAG: Linux Audit-Subsystem Design Documentation for Linux Kernel 2.6, v0.1. Available at http://uniforum.chi.il.us/slides/HardeningLinux/LAuS-Design.pdf (2004)
65. Tang, Y., et al.: Nodemerge: Template based efficient data reduction for big-data causality analysis. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 1324–1337. CCS '18, ACM, New York, NY, USA (2018)
66. The Linux Foundation: Real-Time Linux (2018), https://wiki.linuxfoundation.org/realtime/start
67. The MITRE Corporation: Medical device cybersecurity (2018), https://www.mitre.org/sites/default/files/publications/pr-18-1550-Medical-Device-Cybersecurity-Playbook.pdf
68. Tian, D.J., et al.: Provusb: Block-level provenance-based data protection for usb storage devices. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, New York, NY, USA (Oct 2016)
69. Veríssimo, P., Casimiro, A.: The timely computing base model and architecture. IEEE Transactions on Computers **51**(8), 916–930 (2002)
70. Veríssimo, P., et al.: The timely computing base: Timely actions in the presence of uncertain timeliness. In: Proceeding International Conference on Dependable Systems and Networks. DSN 2000. pp. 533–542. IEEE (2000)
71. Wang, L.: PID control system design and automatic tuning using MATLAB/Simulink. John Wiley & Sons (2020)
72. Wang, Q., et al.: Fear and Logging in the Internet of Things. In: Proceedings of the 25th ISOC Network and Distributed System Security Symposium. NDSS'18 (February 2017)
73. Wu, Y., et al.: Zeno: Diagnosing performance problems with temporal provenance. In: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). pp. 395–420. USENIX Association, Boston, MA (2019)
74. Xu, Z., et al.: High fidelity data reduction for big data security dependency analyses. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 504–516. CCS '16, ACM, New York, NY, USA (2016)
75. Yagemann, C., et al.: Validating the integrity of audit logs against execution repartitioning attacks. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. CCS '21 (2021)
76. Yoon, M.K., et al.: Learning execution contexts from system call distribution for anomaly detection in smart embedded system. In: Proceedings of the Second International Conference on Internet-of-Things Design and Implementation (2017)