Ellipsis: Towards Efficient System Auditing for Real-Time Systems

Ayoosh Bansal*, Anant Kandikuppa*, Chien-Ying Chen*, Monowar Hasan†‡, Adam Bates*, Sibin Mohan†‡
* University of Illinois at Urbana-Champaign {ayooshb2, anantk3, cchen140, batesa}@illinois.edu
† Wichita State University monowar.hasan@wichita.edu
‡ Oregon State University sibin.mohan@oregonstate.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 events, 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. Ellipsis demonstrates a promising path forward for auditing RTS.

Index Terms—Real-time systems, Auditing, Cyber-physical systems

I. INTRODUCTION

As RTS become indispensable in safety- and security-critical domains — medical devices, autonomous vehicles, manufacturing automation, smart cities, etc. [1], [2], [3], [4] — 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 [5], [6], [7], [8] and the need for diagnostic event logging frameworks (e.g., QNX [9], VxWorks [10] and Composite OS [11], [12]) 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 [13]. Exploits in RTS can enable vehicle hijacks [14], [15], manufacturing disruptions [16], IoT botnets [17], subversion of life-saving medical devices [18] and many other devastating attacks. The COVID-19 pandemic has further shed light on the potential damage of attacks on medical infrastructure [19], [20]. 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 [21].

In traditional computing systems, system auditing has proven crucial to detecting, investigating and responding to intrusions [22], [23], [24], [25]. System auditing takes place at the kernel layer and creates a new event for every syscall that is issued. Not only does this approach take the responsibility of event logging out of the hands of the application developer, it also provides a unified view of system activity in a way that application-specific logging simply cannot. In particular, systems logs can be iteratively parsed into a connected graph based on the shared dependencies of individual events, facilitating causal analysis over the history of events within a system [26], [27], [28], [29], [30], [31], [32], [33], [34], [35]. This capability is invaluable to defenders when tracing suspicious activities [23], [25], [24], to the point that the vast majority of cyber analysts consider audit logs to be the most important resource when investigating threats [22]. Hence, the deployment of system-level audit capabilities can help on multiple fronts: (a) fault detection/diagnosis and (b) understanding and detecting security events.

Unfortunately, comprehensive system auditing approaches are not widely used in RTS. RTS logging takes place largely at the application layer [5], [6] or performs lightweight system layer tracing for performance profiling (e.g., log syscall occurrences, but not arguments) [36]; 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 [37] that are incompatible with the temporal requirements of many real-time applications. Furthermore, system auditing can introduce unpredictable behaviors (say, due to the need to flush out a full audit buffer) and timing perturbations, not to mention priority inversions and inter-application contentions — all of which can have significant negative
A. Linux Audit Framework

The Linux Audit system [51] provides a way to audit system activities. An overview of the Linux Audit architecture is presented in Figure 1. When an application invokes a syscall, the subsequent kernel control flow eventually traverses an audit_filter hook. 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 enqueuing it in a message buffer before returning control to the syscall handler and then to the application. Asynchronous from this workflow, a pair of (non-RT) audit daemons (kauditd and auditd) work in tandem to 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 audit records 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 [37], its impacts on RT applications were unclear. Linux Audit not only adds additional latency to each syscall as it generates log messages but also introduces the shared kauditd buffer whose access is coordinated using a spinlock. These changes could potentially wreak havoc on RT task sets as a result of changing execution profiles, resource contention or priority inversion [52]. Encouragingly, upon conducting a detailed analysis (Appendix A) we observed that Linux Audit does not introduce significant issues of priority inversion or contention over auditing resources shared across applications (e.g., kauditd buffer). Further, except for limited outlier cases, the latency introduced by auditing syslogs can be measured and bounded. Hence it is a good candidate for firm and soft deadline RTS as supported by RT Linux [53]. 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. While Linux Audit can be configured to monitor high-level activities such as login attempts [42], its primary utility (and overhead) comes from tracking low-level syscalls, which is the focus of this paper.

B. RTS Properties

Ellipsis leverages properties unique to RT environments, that we describe here. 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. RT Applications structure commonly involve repeating loops that are excellent targets for conversion to templates. Various techniques are employed to analyze the tasksets with high code coverage e.g., worst case execution time (WCET) analysis for real time tasks [54], [55], [56], [57], [58], [59], [60]. All expected behaviors of the system must be accounted for at design time in conjunction with the system

https://bitbucket.org/sts-lab/ellipsis
TABLE I: RTS properties relevant to Ellipsis

| Property          | Relevance to Ellipsis                                                                                                           | Sections |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------|----------|
| Periodic tasks    | Most RT tasks are periodically activated, leading to repeating behaviors. Ellipsis templates describe the most common repetitions. | II-B1, III |
| Aperiodic tasks   | Second most common form of RT tasks. Aperiodic tasks also lead to repeating behaviors, but with irregular inter-arrival times. | II-B1, III |
| Code Coverage     | High code coverage analyses are part of existing RTS development processes, Ellipsis' automated template generation adds minimal cost. | II-B2, VI-D |
| Timing Predictability | A requirement for safety and correct functioning of RTS, naively enabling auditing can violate this by introducing overheads and variability. | IV-I, VI, A-B |
| Isolation         | Resources are commonly isolated in RTS to improve timing predictability. RTS auditing mechanisms should not violate resource isolation. | A-C, A-D |
| Special Purpose   | RTS are special purpose machines, tasks are known at development i.e., templates can be created before system deployment.     | III, VI  |
| Longevity         | Once deployed RTS can remain functional for years. Ellipsis' can save enormous log storage and transmission costs over the lifetime of the RTS. | IV-E  |

designers. Any deviation is an unforeseen fault or malicious activity, which needs to be audited in full detail. Table I contains a summary of RTS features and constraints, with references to sections in this work that discuss, evaluate or leverage these features. In this work we show that applications from two different classes of RTS follow this model; a control application (Ardupilot [61]) and a video analysis application (Motion [62]).

1) Repetition of Sequences: In their seminal work on Intrusion Detection, Hofmeyr et al. [63] established that normal behavior of an application can be profiled as sequences of syscall. This works exceptionally well for RTS as they feature limited tasks with limited execution paths on a system. Unlike general purpose systems, RTS run limited predefined tasks. The requirements of reliability, safety and timing predictability imply that RTS have limited execution paths which can be tested and analyzed to ensure the previously mentioned requirements. The RTS survey [64] further found that 82% of the RTS contained tasks with periodic activation. Periodically activated tasks with limited execution paths will invariably lead to high repetitions of certain sequences. Yoon et al. [60] demonstrated the existence of repeating syscall sequences in a RTS. The reliable repetition of behaviors has also led to profile driven techniques being successfully employed towards achieving predictable temporal behaviors in RTS [65].

2) Code Coverage: A recently published survey of industry practitioners in RTS [64] noted that the five most important system aspects for industrial RTS were Functional correctness, Reliability and availability, System safety, Timing predictability, System security. The role of code coverage in software testing is well established [66], [67]. Prior works have established the correlation between code coverage and reliability of software [68], [69], [70]. Software safety standards include structural code coverage as a requirement [71], [72]. Timing predictability in RTS is ensured by coding standards/guidelines [73], [56], [58] and worst case execution time (WCET) analysis [59], [55]. Therefore, high code coverage is an integral component of RTS development process. Template generation for Ellipsis therefore does not introduce a significant additional burden. Template sequences are determinable in the course of existing development processes for RTS.

C. Threat Model

We consider an adversary that aims to penetrate and impact an RTS through exfiltration of data, corruption of actuation outputs, degradation of 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. 

We assume that the underlying OS and the audit subsystem therein are trusted. This is a standard assumption in system auditing literature [27], [23], [74], [75], [76]. 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 [77], [78], [79], [80].

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., [27], [81], [82]). 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, such as pre-deployment testing and verification.

1 Specifically, our ruleset audits execve, read, ready, 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, setresuid, setresgid, chmod, fchmod, pipe, pipe2, truncate, ftruncate, sendfile, unlink, unlinkat, socketpair, splice, init_module, and finit_module.
TABLE II: Symbols Summary

| Symbol | Description |
|--------|-------------|
| S1, S2, S3 | System Calls |
| TPL - X | Template-X where X ∈ N |
| Δi | Observed runtime for ith instance of templatized task |
| TX | Temporal constraint of TPL - X |
| x,y} | Unique state ID for Ellipsis FSA |
| x | number of system calls matched |
| y | set of potential template matches |
| τ | A RT task |
| si | ith syscall sequence exhibited by τ |
| N | Count of possible si for τ, 0 < i ≤ N |
| len(si) | number of syscalls in si |
| pi | probability of occurrence of corresponding si |

III. 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 [37]. 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 profile2. These templates are generated in an offline profiling phase, similar to common RTS analyses like WCET [54], [83].

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). Matched syscalls are never inserted into the event stream, reducing the number of events generated by the auditing system. Significantly, while a sequence of audited syscall events is replaced by a single record, relevant information is not lost (§V).

A. Model

Consider a system in which the machine operator wishes to audit a single RT task τ. 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. Furthermore, we modified Linux Audit to include thread ids in audit event records.

RT tasks are commonly structured with a one time init component and repeating loops. Let si denote a syscall sequence the task exhibits in a loop execution and N the count of different syscall execution paths τ might take (i.e., 0 < i ≤ N). A template describes these sequences (si), identifying the syscalls and arguments. As noted in Section II-B, 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 len(si) return the number of syscalls in the sequence si. Further, let pi be the probability that an iteration of τ exhibits syscall sequence si.

B. Sequence Identification

Fig. 2: Ellipsis template creation. Syscalls are denoted by S1 and S2. Application is traced to identify repeating syscall sequences, then audited with Ellipsis, using the intermediate template, to get temporal constraint (T1). This creates a profile for task’s execution time (Δ1), yielding temporal constraints (T1). Intermediate templates enriched with temporal constraints are the final templates.

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 [84], [85], with past solutions require binary analysis, code annotations, stack analysis or a combination. While any technique that yields si and pi 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 disambiguating threads belonging to a process, yielding thread level sequences. This first step yields

2 Template examples are available as Appendix B
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 thread-level 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.

C. 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 (7), respectively. As we discuss in detail in Section V the security tradeoff is minimal. Let’s assume $n$ 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.

D. Template Creation

For the next step, Figure 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 (§IV) 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.

E. 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.

F. 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.

G. Audit Event Reduction

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

$$E_A = I \sum_{i=1}^{N} (p_i \times len(s_i)) + f$$  \hspace{1cm} (1)

$$E_E = I \sum_{i=1}^{n} p_i + \sum_{i=n+1}^{N} (p_i \times len(s_i)) + f$$  \hspace{1cm} (2)

$$E_A - E_E = I \sum_{i=1}^{n} (p_i \times len(s_i)) - \sum_{i=1}^{n} p_i$$

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 [86]. This property has been utilized by Yoon et al. in a prior work [60]. Therefore, for RT applications the distribution of $p_i$ is highly biased i.e., certain sequences $s_i$ have high
probability of occurrence. Table III provides example values for the parameters used, determined during the Sequence Identification step in template creation for the evaluation application ArduPilot [IV].

### H. Storage Size Reduction

Let \( B_A \) denote the average cost of representing a syscall event in audit log and \( B_E \) denote the average cost of representing Ellipsis’ template match record. Thus \( B_A \) represents the average size over all events in the Linux Audit log, whereas in Ellipsis syscall sequences that match a template will be removed and replaced with a template match event of an average size \( B_E \). By design, \( B_E < B_A \); \( B_E \) is a constant 343 bytes, while \( B_A \) averaged 527 bytes (1220 bytes 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 \( \tau \)’s audit log for Linux Audit \((L_A)\) and Ellipsis \((L_E)\) is:

\[
L_A = I \times (B_A \times \sum_{i=1}^{N} (p_i \times \text{len}(s_i))) + f \times B_A \tag{4}
\]

\[
L_E = I \times (B_E \times \sum_{i=1}^{n} p_i) + B_A \times \sum_{i=n+1}^{N} (p_i \times \text{len}(s_i)) + f \times B_A \tag{5}
\]

The reduction in log size is given by:

\[
L_A - L_E = I \times (B_A \times \sum_{i=1}^{n} (p_i \times \text{len}(s_i)) - B_E \times \sum_{i=1}^{n} p_i) \tag{6}
\]

From (3) and (6), Ellipsis’ benefits come from the audit events count and log size becoming independent of sequence size \((\text{len}(s_i))\) for the chosen \( n \) sequences, multiplied further by repetitions of these sequences \((I \times p_i)\). Ellipsis behaves identical to Linux Audit for any sequence that is not included as a template, i.e., \( i \geq n + 1 \) in (2) and (5). The impact of any inaccuracies in determining \( p_i \) can be minimized by increasing \( n \), the number of sequences converted to templates.

### I. Memory Tradeoff

The tradeoff for Ellipsis’ benefits are computational overheads (evaluated in §IV-G and §IV-H) and the memory cost of storing templates \((M_T)\). Let \( M_{\text{fixed}} \) be memory required per template, excluding syscall, while \( M_{\text{syscall}} \) be the memory required for each syscall in the template. On 32 bit kernel \( M_{\text{fixed}} = 116 \) and \( M_{\text{syscall}} = 56 \) bytes, determined by sizeof data structures. As an example, 3 templates from evaluation occupied 2 KB in memory (Appendix B)

\[
M_T = M_{\text{fixed}} \times n + M_{\text{syscall}} \times \sum_{i=1}^{N} \text{len}(s_i) \tag{7}
\]

For reference, the parameters for the application detailed in Section IV-B are provided in Table III. Complete templates for the same can be found in Appendix B. The 3 templates used for the case study took 2 KB of memory space.

While templates occupy memory space, as per (7), Ellipsis minimizes \( k\text{audit} \) buffer usage, reducing the memory space that must be dedicated to it. These savings are not considered in (7). Section IV-D evaluates the reduction in \( k\text{audit} \) buffer occupancy when using Ellipsis.

### J. 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.

\[
E_A - E_{\text{Best, Ellipsis-HP}} = I \times \sum_{i=n+1}^{N} (p_i \times \text{len}(s_i)) - n \tag{8}
\]

\[
E_A - E_{\text{Best, Ellipsis-HP}} = I \times \sum_{i=1}^{n} (p_i \times \text{len}(s_i)) - n \tag{9}
\]

### K. Temporal Constraints

Ellipsis and Ellipsis-HP, when used to reduce the long and most frequently occurring sequences, decrease both the volume of audit events generated and the size of log that must be stored. RTS are sensitive to time intervals between events, thus, Ellipsis also considers temporal checks in the template matching process (§III-D and §III-F). Ellipsis-HP adds additional checks for inter-arrival times of different task instances. Note that the earlier discussion on log sizes assumes that temporal constraints are always met. An evaluation of the impact of temporal constraints on log size is provided in Section IV-I.

---

**TABLE III: Parameters from Case Study**

| Task Name | \( N \) | \( I \) | \( \text{len}(s_i) \) | \( p_i \) | \( f \) |
|-----------|--------|--------|----------------|--------|--------|
| arducopter | 5      | 100    | [14, 15, 17, 18] | [0.95, 0.02, 0.01, 0.01] | 679    |
| ap-rin    | 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      |
IV. Evaluation

We evaluate Ellipsis and Ellipsis-HP using two real time applications, ArduPilot [61], a safety-critical firm-deadline autopilot application. Motion [62], conversely, is a soft real-time video analysis application where the application execution paths and behavior are variable depending on configuration and inputs. With these applications 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 (§IV-C), (b) achieve high audit log volume reduction during benign activity, in most cases (§IV-E and §IV-F) (upto 97.55%), even when application can exhibit various behaviors (§IV-F), (c) enjoy minimal computational overhead even in an artificially created worst case scenarios (§IV-G). Using a set of synthetic tasks we also show that the Ellipsis’ overhead per syscall scales independent of the size of template (§IV-H). Unless specified otherwise (§IV-F and §IV-H), all evaluations were conducted using the ArduPilot application.

A. Setup

All measurements were conducted on 4GB Raspberry Pi 4 [87] running Linux 4.19. The RT kernel from raspberrypi/linux [88] was used with additional kconf (CONFIG_PREEMPT_RT_FULL, CONFIG_AUDIT, CONFIG_AUDITSYSCALL) enabled. To reduce computational variability due to external perturbations we disabled power management, directed all kernel background tasks/interrupts to core 0 using the isolepu kernel argument, and set CPU frequency Governor to Performance [89]. 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.

B. 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 [61]. It has been installed in over a million vehicles and has been the basis for many industrial and academic projects [90], [91]. 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 [92] 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 [93]: execve, openat, read, write, close and pread64. Upon running the template generation script on the application binary, we obtained the most frequently occurring 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 as well as temporal constraints. Templates were loaded into the kernel when evaluating Ellipsis or Ellipsis-HP. Auditing was set up to audit invocations of the syscalls made by the ArduPilot application as mentioned above. Complete templates are provided in Appendix B.

C. Audit Completeness

Experiment. We ran the application for 100K iterations at task frequencies of 100 Hz, 200 Hz, 300 Hz and 400 Hz, 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 [94] 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 accommodate Linux Audit may have considerable detrimental effects. Further investigation (§IV-D) revealed that Linux Audit dropped log events due to kaudit buffer overflow, despite the buffer size being 50K. In contrast, Ellipsis is able to provide

3 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
auditing for the entire frequency range without suffering log event loss.

D. Audit Buffer Utilization

Experiment. The size of the kaudit buffer is determined by a “backlog limit” configuration, that controls the number of outstanding audit messages allowed in the kernel [95]. The default configuration is 8192 but as noted before (§IV-A) we set it to 50K. The kaudit buffer state was sampled periodically, once every 2 seconds, by querying the audit command-line utility auditctl during the execution of the application for 100K iterations. Figure 5 shows the comparison of the percentage utilization of the audit buffer by Linux Audit, Ellipsis and Ellipsis-HP over time.

Observations. From Figure 5, we see that for Linux Audit, the utilization of the kaudit buffer rises quickly and remains close to 100% for the majority of the runtime, resulting in loss of audit messages, as measured earlier (§IV-C). In contrast, Ellipsis and Ellipsis-HP ensure that the buffer utilization remains negligible throughout the execution. As noted before (§IV-A), buffer size is already set to the largest value the platform can support without panics or hangs.

Discussion. When the kaudit buffer is full, new audit messages are lost; hence, to ensure that suspicious events are recorded, it is essential that the buffer is never full. Ellipsis is able to keep the buffer from overflowing by reducing the number of outstanding audit logs generated in the system. The variations that we see in the plots can be attributed to the scheduling of the non real-time kaudit thread that is responsible for sending the outstanding audit messages to userspace for retention on disk. We observe that the backlog builds with time when kaudit isn’t scheduled and drops sharply when kaudit eventually gets CPU time.

However there are two limitations to using auditctl to estimate memory usage. First, kaudit buffer size does not consider the additional memory used by Ellipsis and Ellipsis-HP to maintain templates in memory and perform runtime matching. Manual calculations yielded a memory overhead of less than 100 KB, or 1% of the buffer size. Second, the relatively slow sampling rate of 0.5 Hz can miss transient changes in buffer utilization. auditctl reports buffer occupancy at the moment it is invoked. However, running auditctl at a higher frequency leads to changes in application profile. So we ran further experiments to determine the minimum kaudit buffer size with which Ellipsis and Ellipsis-HP can still achieve complete auditing. These further experiments are free from any sampling limitation. We find that a buffer of 2.5K for Ellipsis and 1.5K for Ellipsis-HP was enough to support lossless auditing under normal operation. This reduced memory requirement is valuable for RTS that run on resource constrained platforms. The reduced time that the buffer holds audit logs, reduces the attack window for recently identified race condition attacks on the audit buffer [96]. While the buffer utilization under normal operation is vastly reduced, the buffer limit should still be kept larger than the observed minimum utilization to capture anomalous behavior without loss.

E. 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. 10^5 iterations take around 250 seconds to complete. For Linux Audit we measure the actual size of log on disk albeit with potential log loss. Linux Audit Lossless provides an estimate of the actual size of the log on disk if the auditing was lossless.

Discussion. Figure 6 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 (BA * 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 using the number of logs lost and the average size of each log entry.

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.

F. Motion: Audit Log Size Reduction

Motion [62] is a soft real time video analysis application where the application execution paths and behavior vary depending on configuration and inputs.

Experiment. In this experiment, using the same setup as prior experiments, we run Motion with varying configurations and report the log reduction percentage. Motion application detects movement in camera inputs and saves images when movement is detected within frames. We point a camera to a screen showing a still image (Input = Still) or a video with random motion (Input = Motion). Motion’s behavior is determined via its configuration file. emulate motion when set to on, it causes an image to be saved at a fixed rate of 2 Hz. When emulate motion is set to off, an image is stored only when motion is detected in the input stream. Thus Motion Detection is Enabled when emulate motion=off and Disabled when emulate motion=on. Application logging verbosity is set to minimum level for Limited and maximum level for Verbose. For each configuration, green colored option increases execution variability compared to the other. For each combination templates are learned over a 120 second execution. The application is then audited with Linux Audit and Ellipsis for 300 seconds each.

Observations. We provide the rate of audited syscall events, size on disk for audit logs of both Linux Audit and Ellipsis with the log size reduction percentage in Table IV. In most cases a log reduction of > 90% is achieved. The lowest reduction occurs when the application only processes the camera feed, never saving any images. The resultant log, with low number of syscalls and lowest size, contains disproportionately high events from the setup phase of the application, leading to a lower reduction by Ellipsis, which is still quite high at 81.44% . We also experimented with doubled rate of storing images (4 Hz) but no differences were observed, as is expected. Log loss was not observed in any scenario.

Discussion. Ellipsis achieves high audit event and log reduction even when the application can have variable execution paths, N = 26 for this evaluation. The only requirement is that the execution paths also be encountered in the template generation step. The properties of repeating execution paths, shared by ArduPilot and Motion are present across a vast majority of RT applications.

G. 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 capture the distribution of these measurements over longer application runs. Ellipsis and Ellipsis-HP refers to the normal execution of the application with their respective reduction techniques. 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 forced to fail after the longest possible partial match before eventually failing to reduce the events. Ellipsis NR is also the worst case for Ellipsis-HP.

Observations. Figure 7 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 0.93x and Ellipsis-HP’s overhead is 0.90x of Linux Audit. The observed maximum overheads show a greater improvement. Ellipsis’s observed maximum overhead is 0.87x and Ellipsis-HP’s 0.70x of Linux

### TABLE IV: (§IV-F) Motion Application Log Reduction for different configurations

| Index | Input  | Detection | Application Logging | Syscall Rate | Linux Audit Log Size | Ellipsis Log Size | Reduction |
|-------|--------|-----------|---------------------|--------------|----------------------|------------------|-----------|
| 1     | Motion | Disabled  | Verbose             | 48.2 / s     | 7.6 MB               | 0.19 MB          | 97.55 %   |
| 2     | Still  | Disabled  | Verbose             | 48.0 / s     | 7.6 MB               | 0.24 MB          | 96.80 %   |
| 3     | Motion | Enabled   | Verbose             | 44.2 / s     | 6.9 MB               | 0.29 MB          | 95.83 %   |
| 4     | Motion | Disabled  | Limited             | 33.2 / s     | 5.4 MB               | 0.26 MB          | 95.21 %   |
| 5     | Still  | Disabled  | Limited             | 27.6 / s     | 4.5 MB               | 0.25 MB          | 94.38 %   |
| 6     | Motion | Enabled   | Limited             | 29.7 / s     | 4.8 MB               | 0.41 MB          | 91.55 %   |
| 7     | Still  | Enabled   | Verbose             | 20.9 / s     | 3.2 MB               | 0.30 MB          | 89.83 %   |
| 8     | Still  | Enabled   | Limited             | 8.4 / s      | 1.3 MB               | 0.24 MB          | 81.44 %   |

4https://www.youtube.com/watch?v=ELhIDdGz7M
Audit. Ellipsis NR shows a 1.05x increase in average overhead and 1.07x increase in maximum observed overhead compared to Linux Audit.

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 than 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 (§IV-E). 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’s primary goal, it nonetheless enjoys a modest performance improvement by reducing the total work performed by the underlying audit framework.

H. 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. Temporal constraints are not used. 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 §IV-G). This represents the worst case scenario.

Observations. Figure 8 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 (< 1.3µs)\(^5\).

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.

I. Temporal Constraint Policy

Experiment. We explore here the impact of different policies for temporal constraints. Temporal constraints are applied, intra-task, for Ellipsis and additionally inter-task for Ellipsis-HP. While the constraint values on expected runtimes and expected inter-arrival times of task instances are learned and applied separately for each task, a common policy can be enforced. For example the policy max implies that all timing constraints are set to the maximum value that was observed for them during the learning phase. Other policies explored in this experiment are based on the average (µ) and standard deviation (σ) of the time intervals observed during the learning phase. The none policy disables all temporal constraints and represents the best case in this experiment.

Observations. Figure 9 shows the impact of different temporal constraint policies on log size. With more stringent timing constraints, fewer task instances are observed to adhere to constraints leading to an increase in log size. max and none

\(^5\) Where isolcpu is not used, Ellipsis has especially low execution time variability, detailed evaluation available as Appendix C
A. 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 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 [84], [29], [85].

B. 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.

```plaintext
type=SYSCALL msg=audit(1601405431.612391366:5893333): arch =40000028 syscall=4 per=800000 success=1 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 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:

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

J. Summary of Results

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

V. 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.

A. 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
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, naively 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 [97] 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 (§IV-B). Using tools provided with Scheduleak [97], 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 10a. 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 (§IV-C), potentially losing critical forensic evidence.

Discussion. Scheduleak [97] 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 10a. 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.

D. Demonstration: Data Exfiltration Attack

Motion [62] monitors camera images and detects motion by tracking pixel changes between consecutive image frames. It is primarily used for surveillance and stores images when movement is detected. Images are stored at a location specified by the system administrator.

Attack Scenario. The attacker inserts malicious code into the victim application to save images to an attacker controlled location when motion is detected, as shown in Figure 10b. The attacker can exploit another process running on the system in order to exfiltrate these images out of the system at a later point in time, successfully leaking sensitive information. The attack can be realized using the following code snippet, developed by Yoon et al. [60]:

```c
const char* orig_target_dir = cnt->conf.target_dir;
cnt->conf.target_dir =="/tmp";
event(cnt, EVENT_IMAGE_DETECTED, &cnt->imgs.img_ring[ 
cnt->imgs.img_ring_out], NULL, NULL, &cnt->imgs.
img_ring[cnt->imgs.img_ring_out].timestamp_tv);
cnt->conf.target_dir = orig_target_dir;
```

Experimental Setup. Using the same platform setup as in rest of this paper, we ran Motion v4.3.2 using a webcam as a video source. While not commonly considered forensically relevant, we include ioctl, rt_sigprocmask and gettimeofday in our audit ruleset as these syscalls are used to capture frames from video devices and maintain video frame rates. By running the template generation tools we obtained two templates that describe how Motion (i) captures an image frame and (ii) captures an image frame with movement and saves it to the file system. Both Motion and the malicious application are audited by Ellipsis for 5 minutes. We introduce movement in the camera’s field of view to trigger image stores. Images get stored to both benign and malicious locations in the system.

Results. Ellipsis correctly reduces audit logs that correspond to the capture of image frames where motion is not detected because that behavior matches the templates. As the attacker inserts code to copy image frames describing movement, Ellipsis observes additional occurrences of openat, write and close syscalls that differ from behavior described by the
templates, therefore retaining complete audit logs generated in response to observed movement.

VI. DISCUSSION

A. System Scope & Limitations

Ellipsis is useful for any application that has predictable repeating patterns. When sequence counts are too numerous 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. Lastly, we note that while we have motivated our design by discussing periodic tasks, Ellipsis is able to work effectively on any predictable execution profiles; for example, Ellipsis would also be effective for aperiodic or time table triggered tasks, which are significantly prevalent in industrial RTS [64].

B. Auditing Hard Deadline 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 kaudt buffer occupancy must have a strict upper bound. In this paper Ellipsis takes a long step forward, deriving high confidence empirical bounds (§IV-G) to enable Ellipsis’ use in firm- or soft-deadline RTS, which are prolific [64]. However, the strict bounds required for Hard RTS are a work in progress.

C. Unfavorable Conditions

We consider here the impact of using Ellipsis to audit hypothetical RTS where our assumptions about RTS properties do not hold (§II-B). 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 [64] 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. (7)). 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.

D. Code Coverage

While high code coverage is important in our motivating use case of RT applications where reliability is a concern [70], it should be noted that perfect code coverage is not a requirement for the use of Ellipsis. Ellipsis can be deployed on any system for any application under audit, where the log reduction benefit is proportional to the ratio of runtime spent in previously analysed execution paths that are included as templates. Perfect code coverage does allow for accurate objective analysis of Ellipsis’ audit event and log reduction potential, using (3) and (6), as perfect code coverage analysis would yield complete sequences \(s_i\).

E. Deployment Considerations

The mechanisms for template use are fully flexible. Any sequence for any task can be independently reduced with Ellipsis. However, to use Ellipsis beneficially, sequences with high probability of occurrence \((p_i)\) should be chosen i.e., top \(n\) sequences by high \(p_i\) out of total \(N\). The primary tradeoff is the memory cost of storing templates, as in (7). For an RTS with limited memory, using (2) and (7), \(n\) value can be chosen for each task independently to minimize the Ellipsis events generated. The parameter \(n\) is chosen independently for each task, allowing highly optimized use of main memory available for storing templates. A second tradeoff is security. As the information lost by Ellipsis is minimal (§V), the tradeoff on security is also minimal.

VII. RELATED WORK

System Auditing. Due to its value in threat detection and investigation, system auditing is a subject of interest in traditional systems. While a number of experimental audit frameworks have incorporated notions of data provenance [27], [31], [76], [98] and taint tracking [45], [29], the bulk of this work is also based on commodity audit frameworks such as Linux Audit. Techniques have also been proposed to efficiently extract threat intelligence from voluminous log data [99], [100], [46], [23], [30], [24], [101], [102], [32], [33], [34], [25], [85], [28], [103], [35], [104], [105]; in this work, we make the use of such techniques applicable to RTS through the design of a system audit framework that is compatible with temporally constrained applications. Our approach to template generation in Ellipsis shares similarities with the notion of execution partitioning of log activity [84], [32], [85], [23], [24], 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 application instrumentation to facilitate. Further, the well-formed nature of real-time tasks ensures the correctness of our execution units i.e., templates.

Auditing RTS. Although auditing has been widely acknowledged as an important aspect of securing embedded devices [38], [39], [40], challenges unique to auditing RTS have received limited attention. Wang et al. present ProvThings, an auditing framework for monitoring IoT smart home deployments [106], 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 [107]. 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 [108]. Whereas their system uses auditing to diagnose performance problems in networks, the presented study considers the performance problems created by auditing within RT applications.

Forensic Reduction. Significant effort has been dedicated to improving the cost-utility ratio for system auditing by pruning, summarizing, or otherwise compressing audit data that is unlikely to be of use during investigations [43], [44], [45], [46], [47], [48], [49], [50], [105], [109], [110]. However these approaches address the log storage overheads and not the voluminous event generation that is prohibitive to RTS auditing (§IV-C). KCAL [37] and ProTracer [29] 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 [47]), 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 [23], [24], [32], [84], [85], 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. To our knowledge, this work is the first to address the need for forensic reduction of system logs in RTS.

VIII. 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.

REFERENCES

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

[2] I. Lee, O. Sokolsky, S. Chen, J. Hatchiff, E. Jee, B. Kim, A. King, M. Mullen-Fortino, S. Park, A. Roederer et al., “Challenges and research directions in medical cyber-physical systems,” Proceedings of the IEEE, vol. 100, no. 1, pp. 75–90, 2011.

[3] L. Monostori, B. Kaidar, T. Bauemhansl, S. Kondoh, S. Kumara, G. Reinhart, O. Sauer, G. Schuh, W. Sihn, and K. Ueda, “Cyber-physical systems in manufacturing,” Cirp Annals, vol. 65, no. 2, pp. 621–641, 2016.

[4] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, “Cyber-physical systems: the next computing revolution;” in Design Automation Conference. IEEE, 2010, pp. 731–736.

[5] K. Böhm, T. Kubiatko, D. Paula, and H.-G. Schweiger, “New developments on edr (event data recorder) for automated vehicles,” Open Engineering, vol. 10, no. 1, pp. 140–146, 2020.

[6] U. Bose, “The black box solution to autonomous liability,” Wash. UL Rev., vol. 92, p. 1325, 2014.

[7] P. BIGELOW, “Tesla/ntsb feud shows complications of crash investigations involving autonomous systems,” 2018. [Online]. Available: https://www.caranddriver.com/news/19785733/teslantsb-feud-shows-complications-of-crash-investigations-involving-autonomous-systems/

[8] A. Martyn, “Tesla blames drivers who wreck its cars but won’t hand over crash data without a court order,” 2018. [Online]. Available: https://www.consumeraffairs.com/news/tesla-blames-drivers-who-wreck-its-cars-but-wont-hand-over-crash-data-without-a-court-order-053018.html

[9] “The instrumented microkernel,” 2020. [Online]. Available: https://www.qnx.com/developers/docs/c6.4/nctrimno/sys_arch_trace.html

[10] “Tracealyzer for vxworks,” 2020. [Online]. Available: https://perceptis.com/docs/VxWorks/manual/

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

[12] J. Song and G. Parmer, “C’mon: a predictable and dependable system for predictable and dependable computing,” in 23rd IEEE Real-Time and Embedded Technology and Applications Symposium. IEEE, 2015, pp. 247–258.

[13] Department of Homeland Security, “Cyber physical systems security,” 2020. [Online]. Available: https://www.dhs.gov/science-and-technology/cpssec

[14] C. Crane, “Automotive cyber security: A crash course on protecting cars against hackers,” 2020. [Online]. Available: https://www.thessistore.com/blog/automotive-cyber-security-a-crash-course-on-protecting-cars-against-hackers/

[15] J. Hayes, “Hackers under the hood,” 2020. [Online]. Available: https://eandt.theiet.org/content/articles/2020/03/hackers-under-the-hood/

[16] D. Shepherd, “Industry 4.0: the development of unique cybersecurity,” 2020. [Online]. Available: https://www.manufacturingglobal.com/technology/industry-4-0-development-unique-cybersecurity

[17] M. Halahd, “Iot proliferation and widespread 5g: A perfect botnet storm,” 2020. [Online]. Available: https://www.scmagazine.com/home/opinion/Executive-insight/iot-proliferation-and-widespread-5g-a-perfect-botnet-storm/

[18] The MITRE Corporation, “Medical device cybersecurity,” 2018. [Online]. Available: https://www.mitre.org/sites/default/files/publications/pr-18-1550-Medical-Device-Cybersecurity-Playbook.pdf

[19] R. Begg, “Step up cyber hygiene: Secure access to medical devices,” 2020. [Online]. Available: https://www.machinedesign.com/medical-design/article/2112832/step-up-cyber-hygiene-secure-access-to-medical-devices

[20] G. Slabodkin, “Coronavirus chaos ripe for hackers to exploit medical device vulnerabilities,” 2020. [Online]. Available: https://www.mediachive.com/news/coronavirus-chaos-ripe-for-hackers-to-exploit-medical-device-vulnerabilities/575717/

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

[22] 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.

[23] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, “NoDoze: Combating Threat Alert Fatigue with Automated Provenance Triage,” in 26th ISO/IEC Network and Distributed System Security Symposium, ser. NDSS’19. February 2019.

[24] W. U. Hassan, M. Noudrefine, P. Datta, and A. Bates, “OmegaLog: High-Fidelity Attack Investigation via Transparent Multi-layer Log
Analysis,” in 27th ISOC Network and Distributed System Security Symposium, ser. NDSS’20, February 2020.

[25] R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatashrishan, “Holmes: Real-time apt detection through correlation of suspicious information flows,” in 2019 IEEE International Symposium on Security and Privacy (SP). Los Alamitos, CA, USA: IEEE Computer Society, may 2019. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP.2019.900026

[26] S. T. King and P. M. Chen, “Backtracking intrusions,” in Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, ser. SOSP ’03. New York, NY, USA: ACM, 2003, pp. 223–236. [Online]. Available: http://doi.acm.org/10.1145/45445.945467

[27] A. Bates, D. Tian, K. R. Butler, and T. Moyer, “Trustworthy Whole-System Provenance for the Linux Kernel,” in Proceedings of 24th USENIX Security Symposium, Aug, 2015.

[28] K. Pei, Z. Gu, B. Saltariformaggio, S. Ma, F. Wang, Z. Zhang, L. Si, X. Zhang, and D. Xu, “Hercule: Attack story reconstruction via community discovery on correlated log graph,” in Proceedings of the 32Nd Annual Conference on Computer Security Applications, ser. ACSAC ’16. New York, NY, USA: ACM, 2016, pp. 583–595. [Online]. Available: http://doi.acm.org/10.1145/2991079.2991122

[29] S. Ma, X. Zhang, and D. Xu, “ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting,” in Proceedings of NDSS ’16, Feb. 2016.

[30] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. Stoller, and V. Venkatashrishan, “SLEUTH: Real-time attack scenario reconstruction from COTS audit data,” in 26th USENIX Security Symposium (USENIX Security ’17), Vancouver, BC, USENIX Association, Aug. 2017, pp. 487–504. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/hossain

[31] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Selitzer, and J. Bacon, “Practical whole-system provenance capture,” in Proceedings of the 2017 Symposium on Cloud Computing, ser. SoCC ’17. New York, NY, USA: ACM, 2017, pp. 405–418. [Online]. Available: http://doi.acm.org/10.1145/3127479.3129249

[32] Y. Kwon, F. Wang, W. Wang, K. H. Lee, W.-C. Lee, S. Ma, X. Zhang, D. Xu, S. Jha, G. Ciocarlie, A. Gehani, and V. Vynecwarsan, “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.

[33] Y. Liu, M. Zhang, D. Li, K. Jee, Z. Li, Z. Wu, J. Rhee, and P. Mittal, “Towards a Timely Causality Analysis for Enterprise Security,” in Proceedings of the 25th ISOC Network and Distributed System Security Symposium, ser. NDSS’18, San Diego, CA, February 2018.

[34] S. M. Milajerdi, R. Gjomemo, B. Eshete, and V. Venkatashrishan, “Experiences from applying wcet for counter autonomous misuse,” in Proceedings of the Seventh IEEE International Workshop on Object and Component-Oriented Real-Time Distributed Computing, 2007.

[35] S. M. Milajerdi, B. Eshete, R. Gjomemo, and V. N. Venkatashrishan, “Kernel-supported cooperative auditing for low-cost and efficient intrusion detection,” in Proceedings of 24th IEEE Symposium on Security and Privacy, ser. SP ’03. New York, NY, USA: ACM, 2003, pp. 1324–1337. [Online]. Available: http://doi.acm.org/10.1145/1145280.851267

[36] Y. Tang, D. Li, Z. Li, M. Zhang, K. Jee, X. Xiao, Z. Wu, J. Rhee, F. Xu, and Q. Li, “Nodemerge: Template based efficient data reduction for big-data causality analysis,” in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, ser. CCS ’18. New York, NY, USA: ACM, 2018, pp. 401–410. [Online]. Available: http://doi.acm.org/10.1145/3243763.3243784

[37] C. Burguierie and C. Rochange, “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.

[38] J. Gustafsson and A. Ermedahl, “Experiences from applying wcet analysis in industrial settings,” in 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC’07). IEEE, 2007, pp. 382–392.

[39] L. Hatton, “Safer language subsets: an overview and a case history, misra c,” Information and Software Technology, vol. 46, no. 7, pp. 465–472, 2004.

[40] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” J. ACM, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[41] P. Puschner and A. Burns, “Writing temporally predictable code,” in Proceedings of the Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems.(WORDS 2002). IEEE, 2002, pp. 85–91.

[42] K. Kohei, “Recent security features and issues in embedded systems,” in Proceedings of the 2007 IEEE International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Springer, 2004, pp. 146–160.
[104] Q. Wang, W. U. Hassan, D. Li, K. Jee, X. Yu, K. Zou, J. Rhee, Z. Zhen, W. Cheng, C. A. Gunter, and H. Chen, “You Are What You Do: Hunting Stealthy Malware via Data Provenance Analysis,” in 27th ISOC Network and Distributed System Security Symposium, ser. NDSS’20. February 2020.

[105] V. Sundaram, P. Eugster, and X. Zhang, “Prius: Generic hybrid trace compression for wireless sensor networks,” in Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems, 2012, pp. 183–196.

[106] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, “Fear and Logging in the Internet of Things,” in Proceedings of the 25th ISOC Network and Distributed System Security Symposium, ser. NDSS’18. February 2017.

[107] D. J. Tian, A. Bates, K. R. B. Butler, and R. Rangaswami, “Provush: Block-level provenance-based data protection for usb storage devices,” in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, ser. CCS ’16. New York, NY, USA: ACM, Oct 2016.

[108] Y. Wu, A. Chen, and L. T. X. Phan, “Zeno: Diagnosing performance problems with temporal provenance,” in 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). Boston, MA: USENIX Association, 2019, pp. 395–420. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/wu

[109] C. Chen, H. T. Lehri, L. Kuan Loh, A. Altur, L. Jia, B. T. Loo, and W. Zhou, “Distributed provenance compression,” in Proceedings of the 2017 ACM International Conference on Management of Data, 2017, pp. 203–218.

[110] M. N. Hossain, J. Wang, R. Sekar, and S. D. Stoller, “Dependence-preserving data compaction for scalable forensic analysis,” in Proceedings of the 27th USENIX Conference on Security Symposium, ser. SEC’18. Berkeley, CA, USA: USENIX Association, 2018, pp. 1723–1740. [Online]. Available: http://dl.acm.org/citation.cfm?id=3277203.3277331

[111] Y. Wu, A. Chen, and L. T. X. Phan, “Zeno: Diagnosing performance problems with temporal provenance.” in NSDI, 2019, pp. 395–420.

[112] “Zipkin,” 2021. [Online]. Available: https://zipkin.io/

[113] N. S. Agency, “Information systems security organization,” 2020. [Online]. Available: https://www.commoncriteriaportal.org/files/ppfiles/capp.pdf

[114] ——, “Labeled security protection profile,” 2020. [Online]. Available: https://www.commoncriteriaportal.org/files/ppfiles/lspp.pdf

[115] F. o. A. S. Steven Aftergood, “National industrial security program operating manual,” 2020. [Online]. Available: https://fas.org/sgp/library/nispom.htm

[116] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun, “Wcet(m) estimation in multi-core systems using single core equivalence,” in 2015 27th Euromicro Conference on Real-Time Systems, 2015, pp. 174–183.
APPENDIX A

AUDIT FRAMEWORK ANALYSIS

The Linux Audit system provides a way to observe and analyze system activities. While Linux Audit can be configured to monitor high-level activities such as login attempts, its primary utility (and overhead) comes from tracking low-level syscalls, which is the focus of this work.

A. Setup

We use a 4GB Raspberry Pi 4 [87] running Linux 4.19. The kernel from raspberrypi/linux [88] was used with kconfig (CONFIG_PREEMPT_RT_FULL, CONFIG_AUDIT, CONFIG_AUDIT_ON_SYSCALL) enabled. To reduce computational variability due to external perturbations we disable power management, direct all kernel background tasks/inter-upts to core 0 using the isolcpu kernel argument and set CPU frequency Governor to Performance [89]. Audit rules for capturing syscall events were configured to only match against our benchmark application (i.e., background process activity was not audited). We set the kauditd buffer size to 50K as it was found to be the highest stable configuration possible for the evaluation platform. Larger values led to system panic and hangs.

A microbenchmark, code below, is executed to observe variations in syscall execution time in the presence of external factors such as auditing, real time scheduling priorities, background stress and parallel execution. Since the goal of this analysis is to measure the overheads imposed on syscalls by the audit framework, it is necessary to minimize the latency of the syscall itself. For this reason, our analysis primarily uses getpid, a low latency non-blocking syscall that does not require any arguments. Other syscalls were also evaluated to ensure the generality of observations (§A-B, Fig. 12).

```c
for (i = 0; i < 1000; i++) {
    clock_gettime(CLOCK_MONOTONIC, start_data);
    syscall(1); // Replaced with specific syscalls
    clock_gettime(CLOCK_MONOTONIC, stop_data);
    clock_gettime(CLOCK_MONOTONIC, empty_start);
    clock_gettime(CLOCK_MONOTONIC, empty_stop);
    latency = timespec_subtract(start_data, stop_data)
           timespec_subtract(empty_start, empty_stop);
}
```

B. Temporal Analysis

**Experiment:** We first measure the overhead added by Linux Audit when processing an individual syscall. Figure 11 shows the latency to execute the benchmark application issuing a getpid syscall. Each column shows a box plot of getpid execution latency over 1000 iterations. The baseline scenario has auditing disabled and no other application running. For the audit scenario, the baseline is repeated but the benchmark application is under audit for the getpid syscall. For the RT+audit scenario, we execute the previous scenario with the benchmark application running at an RT priority. In the filtered scenario, the benchmark application is still under audit but getpid is no longer in the ruleset. In all earlier scenarios, the kauditd buffer never overflows and complete logs are captured; the lost scenario shows the latency imposed on getpid when the audit framework attempts to create a log event but is unable to due to buffer overflow, causing the event to be lost.

**Observations:** As can be seen, the observed maximum overhead was just under 100 µs (audit). This max reduces to 60 µs when the application under audit is assigned an RT priority. If a syscall is excluded from audit rules or the event log is lost due to the buffer being full, the overhead is much smaller at ≤ 1µs and ≤ 20µs, respectively. Figure 12 shows the latencies for baseline (Audited = False) and RT + audit (Audited = True) scenarios for different syscalls. These latencies appear to hold roughly consistent for different syscalls.

C. Priority Inversion

**Experiment:** Priority inversion [52], an instance of resource contention in which a higher priority task is blocked by a lower priority task, is a serious threat to the stability of RTS. In the case of system auditing, a potential concern is that a low priority task’s usage of auditing could block a higher priority task by the virtue of shared usage of auditing resources and services e.g., backlog buffer and audit daemons. To investigate this, we repeat the RT+audit scenario from the past experiment, introducing a stress application running parallel in the background. The stress application is audited and executes the same getpid workload over 4 threads, but is run at a lower non-RT priority. Figure 13 reports the latency of syscall execution for the main benchmark application only, i.e., not including the stress application’s latencies.

**Observations:** The RT+audit+stress scenario in Figure 13 is equivalent to the RT+audit scenario in Figure 11, except with the addition of the stress workload. We find that when an audited application runs with high priority it does not suffer any additional contention from any non-real-time audited workloads being run in parallel, as we observe negligible latencies compared to the baseline. Hence, the audit framework does not induce priority inversion when auditing applications with different priority. This is an expected (albeit reassuring) result because the insertion of new logs into the kaudit buffer does not block and audit daemons are not run at RT priority.

D. Resource Contention

**Experiment:** Linux Audit introduces a resource shared among RT tasks being audited, the kauditd buffer protected by a spinlock. Parallel accesses can lead to contention and blocking, even between tasks with same RT priority. To test the presence of contention, we repeat the earlier benchmark in which a single-threaded application issues getpid calls. While measuring the latency of one thread, we introduce an increasing number of additional threads running the same benchmark at same RT priority. The threads are synchronized via a barrier to start executing syscalls at the same time. We provision the kauditd buffer such that it is large enough to prevent overflow. The benchmark has a tight loop that
runs only the `getpid` syscall in each thread, hence ruling out the processor cache or memory bandwidth as sources of contention.

**Observations:** The execution times for the syscalls from the thread under observation is shown in Figure 14. In the average case, we observe only a small difference in the latency of `getpid` regardless of the parallel workloads. While the observed worst case overhead is greater with 3 or 4 threads, it is still under 100 $\mu$s even when the tasks on all 4 cores are being audited. Delays due to contention would occur if multiple threads try to access the spinlock at the same time; but even with the fast `getpid` call the threads minimally contend on the spinlock. This result intuitively follows as the shared spinlock covers a small critical section containing fast pointer manipulations only, making contention uncommon even in this unfavorable scenario with repeated calls to a fast syscall.

### E. Remarks

Encouragingly, we observe that Linux Audit does not introduce significant issues of priority inversion or contention. Hence it is a good candidate for RTS. While contention is possible due to the spinlock on the `kauditd` buffer, this cost does not impact the average latency of auditing as the number of parallel threads increases. Further, except for limited outlier cases, the latency introduced by syscalls can be measured and bounded. This works well for the latency-sensitive RT systems that RT Linux is intended for. However, the storage and management of these audit logs remains a significant challenge for resource-bound devices e.g., our experiments in §A-D generated over 2 MB of audit logs in 50 ms.
## Appendix B
### Templates for ArduPilot

Table V.

| Thread/Task   | ardulcopter | ap-rcin | ap-spi-0 |
|---------------|-------------|---------|----------|
| Syscall Count | 14          | 16      | 1        |
| Expected runtime (ns) | 1303419 | 671567 | 0        |
| Expected inter-arrival time (ns) | 5012313 | 20029121 | 2010477 |

ArduPilot yielded 3 templates. System call numbers and their corresponding arguments, a0 - a4, were extracted from the audit logs. read, write, pread64 have syscall numbers 3, 4 and 180 respectively. Argument values of -1 and temporal constraint values of 0 denote that these arguments are ignored. 4:3:-1:1:-1 then indicates a write syscall with a0 as 3, a2 as 1. a1 and a3 are not forensically relevant. Table V describes the complete templates. An execution sequence matching a template is reduced to a single line in the audit logs at runtime as shown in the following example:

```
<template>
tYPE=SYSCALL MSG=audit (1601405431612391356:5893330)  
  : arch=40000028  : per=800000  : template=arducopter  
  : rep=10  : stime=160140531589320747  : etime=16014054312287042  
  : pid=1208  : uid=1261  : tid=1261  : euid=0  : suid=0  : fsuid=0  
  : egid=0  : sgid=0  : fsgid=0  : tty=pts0  : ses=3  
  : comm="arducopter"  
  : exec="/home/pi/ardupilot/build/navio2/bin/arducopter"  
  : key=(null)
</template>
```

Some fields in Ellipsis log are distinct from standard Linux Audit logs:

- **template**: The name of the template. This is the first line of a template file e.g., [arducopter, ap-rcin, ap-spi-0] in Table V.
- **rep**: The number of consecutive repetitions of the template this entry represents. For Ellipsis the rep value is always 1.
- **stime**: Timestamp of the first syscall in this reduced sequence, unit is nano seconds.
- **etime**: Timestamp of the last syscall in this reduced sequence, unit is nano seconds.

## Appendix C
### Overhead Variability

Fig. 15: Syscall execution time variability.

In all other experiments in this work, we direct kernel background tasks and interrupts away from the cores running real time applications using the isolcpu kernel argument, as described earlier in system setup. The same setup was used to determine the Ellipsis overhead scaling.

In Figure 15 the Isolcpu case is based on the same result, arranged to show the variability introduced in the latency to execute a getpid syscall, in the combined data of all task lengths. The Y-axis in Figure 15 thus shows the latency to execute tasks of length \( t \in [10, 20, 30... 300] \) divided by the length of the task. Therefore Figure 15, Isolcpu case shows that execution latency of a syscall, average and variance, is independent of task length or template length or use of Ellipsis.

While isolation techniques are typically employed in multicore RTS [116], there can exist RTS where due to resource constraints, kernel background processes and interrupts cannot be diverted away from the cores running real time applications. The No Isolcpu case in Figure 15 shows the latency to execute a getpid syscall for the same setup as before but without the use isolcpu kernel command line argument. Ellipsis significantly reduces the variability when log reduction succeeds.

Latency variability is directly related to the number of audit logs generated as only when a log is generated do the auditing hooks in syscall code paths interact with log handling daemons of Linux Audit via the kaudit buffer. Ellipsis’s reduction of log generation therefore also reduces variability in a syscall execution time by reducing the number of logs generated. This benefit of Ellipsis can be valuable to RTS that cannot dedicate a cpu core to interrupts and background processes.
APPENDIX D

AUDIT LOG RECONSTRUCTION

This section shows how information can be constructed back from Ellipsis output. It also notes what specific information is lost in the compression decompression process. For sake of brevity this example shows a simplified stencil of length 3 and only considers Ellipsis.

Let’s assume that three events are recorded at runtime that would have generated the following log without Ellipsis:

```
type=SYSCALL msg=audit((1601405431.6123931367, 1601405431.6123931367, 1601405431.6123931367):2): arch=40000028 syscall=4 per=800000 success=yes exit=7 a0=5 a1=126 a2=1 a3=0 items=0 ppid=1513 pid=1526 tid=1526 auid=1000 uid=0 euid=0 suid=0 fsuid=0 gid=0 egid=0 sgid=0 fsgid=0 tty=pts0 ses=1 comm="arducopter" exe="/home/pi/ardupilot/build/navio2/bin/arducopter" key=(null)
```

As can be inferred from above, except an audit ID, all information can be reconstructed. Arguments that were not reconstructed here, were explicitly ignored in the template as they did not exist or were deemed irrelevant for forensic analysis. Event timings are inexact, but bounded. The range of uncertainty depends on whether Ellipsis or Ellipsis-HP is being used and the temporal policy. Each process always has some unmodified audit entries, like exe from process spawn. Process wide constant entries like PROCTITLE can be reconstructed based on the audit information from the setup phase of the application. The loss of exact event timings also loses the exact interleaving of events across different tasks. But real-time tasks are designed to not have inter-task interference. Further, a successful iteration of the periodic task which meets its timing constraints has no further negative implications for future iterations.

Let’s further assume that following template was loaded for the process.

```
arducopter
3
1303419
5012313
4:3:-1:1:-1
4:4:-1:1:-1
4:5:-1:1:-1
```

Ellipsis compresses the three events into a single line as below:

```
type=SYSCALL msg=audit((1601405431.6123931367, 1601405431.6123931367, 1601405431.6123931367):2): arch=40000028 syscall=4 per=800000 template=arducopter rep=1 stime=1601405431612391356 etime=1601405431612391367 ppid=1513 pid=1526 tid=1526 auid=1000 uid=0 euid=0 suid=0 fsuid=0 gid=0 egid=0 sgid=0 fsgid=0 tty=pts0 ses=1 comm="arducopter" exe="/home/pi/ardupilot/build/navio2/bin/arducopter" key=(null)
```

Using the template and the compressed line of log, following three lines can be reconstructed. 0 denotes values that could not be reconstructed and [min, max] enclose values for which range is known but not the exact value.

```
type=SYSCALL msg=audit((1601405431.612391356):2): arch=40000028 syscall=4 per=800000 success=yes exit=8 a0=3 a1=126 a2=1 a3=0 items=0 ppid=1513 pid=1526 tid=1526 auid=1000 uid=0 euid=0 suid=0 fsuid=0 gid=0 egid=0 sgid=0 fsgid=0 tty=pts0 ses=1 comm="arducopter" exe="/home/pi/ardupilot/build/navio2/bin/arducopter" key=(null)
```

```
type=SYSCALL msg=audit((1601405431.612391333):4 per=800000 success=yes exit=7 a0=5 a1=126 a2=1 a3=0 items=0 ppid=1513 pid=1526 tid=1526 auid=1000 uid=0 euid=0 suid=0 fsuid=0 gid=0 egid=0 sgid=0 fsgid=0 tty=pts0 ses=1 comm="arducopter" exe="/home/pi/ardupilot/build/navio2/bin/arducopter" key=(null)
```

```
type=SYSCALL msg=audit((1601405431.612391334):4 per=800000 success=yes exit=7 a0=5 a1=126 a2=1 a3=0 items=0 ppid=1513 pid=1526 tid=1526 auid=1000 uid=0 euid=0 suid=0 fsuid=0 gid=0 egid=0 sgid=0 fsgid=0 tty=pts0 ses=1 comm="arducopter" exe="/home/pi/ardupilot/build/navio2/bin/arducopter" key=(null)
```
