EFFICIENT SYSTEM AUDITING FOR REAL-TIME SYSTEMS

BY

ANANT KANDIKUPPA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Adviser:

Assistant Professor Adam Bates

ABSTRACT

Auditing is a powerful tool that provides machine operators with the mechanisms to observe, and glean insights from, generic computing systems. The information obtained by auditing systems can be used to detect and explain suspicious activity, from fault/error diagnosis to intrusion detection and forensics after security incidents. While such mechanisms would be beneficial for Real-Time Systems (RTS), existing audit frameworks are rarely designed for this domain. If audit mechanisms are not carefully integrated into real-time operating systems, they can negatively impact the temporal constraints of RTS. In this paper, we demonstrate how to *apply commodity audit frameworks to real-time systems*. We design novel kernel-based reduction techniques that leverage the periodic, repetitive, nature of real-time (RT) applications to aggressively reduce the costs/overheads of a system-level auditing, *viz.,* Linux Audit (a popular open source audit framework). This is coupled with a rigorous analysis to understand the conflicts between the temporal requirements of RT applications and the audit subsystem. Our approach, *Ellipsis*, generates succinct behaviors of RT application and retains a *lossless record* of process activity, enabling analysis/detection of unexpected activity while *meeting temporal constraints*. Our evaluation of *Ellipsis*, using ArduPilot (an open-source autopilot application suite) and synthetically generated tasksets, demonstrates *up to 93% reduction* in audit event generation.

ACKNOWLEDGMENTS

I would first like to thank my adviser, Professor Adam Bates for his guidance and encouragement over the past two years. I really appreciate his willingness to provide me this research opportunity which has helped me grow as a researcher and software developer.

Next, I would like to thank my collaborators. The work presented in this thesis is shared with an amazing group of student and faculty members, namely Ayoosh Bansal, CY Chen, Monowar Hasan and Professor Sibin Mohan. I really appreciate their valuable contributions and support that helped bring this project to fruition.

Finally, I would like to thank my parents for their constant love and support that helped me navigate these challenging and uncertain times during the pandemic.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

The need for effective fault detection and diagnostic tools has prompted the development of a variety of event logging frameworks for real-time operating systems, including Composite OS [1, 2], QNX [3] and VxWorks [4]. As RTS become increasingly indispensable in safety- and security-critical domains such as medical devices, autonomous vehicles, unmanned aeronautical vehicles, manufacturing automation, electricity grids and smart cities [5, 6, 7, 8], the need for effective and precise *auditing* support is only growing. Even now, when investigating vehicle collisions and crashes, event data recorders (or *black boxes*) are crucial for determining fault and liability [9, 10, 11, 12]. However, 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 disruption [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 in nature, but active and ongoing, as evidenced most 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. System auditing takes place at the *kernel layer* and creates a new event for every system call 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 [22, 23, 24, 25, 26, 27, 28, 28, 29, 30, 31]. This capability is invaluable to defenders when tracing suspicious activities [32, 33, 34], to the point that the vast majority of cyber analysts consider audit logs to be the most important resource when investigating threats [35]. 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* [9, 10] or performs lightweight system layer tracing for performance profiling [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], which is incompatible with the strict temporal requirements of many real-time applications. For instance, consider an airbag deployment system in a modern automobile that has only 50 milliseconds to fully deploy the airbag after it senses a collision [38]. A naïve deployment of auditing in such a system could directly result in personal injury to passengers in the car. Furthermore, system auditing will introduce unpredictable behaviors (say, due to the need to flush out the audit buffer when full) and timing perturbations, not to mention priority inversions and inter-application contentions — all of which can have significant negative impacts on the safety of the RTS. Thus, while we are encouraged by the growing recognition of the importance of embedded auditing [39, 40, 41] and the newfound availability of Linux Audit in the Embedded Linux distribution [42], a practical approach to RTS auditing remains an elusive goal.

This work presents a thorough exploration of aggressive system auditing within real-time environments. We first conduct an analysis of Linux Audit's adherence to real-time scheduling principles, discovering that while Audit introduces large overheads and increased variance to each system call, it does not introduce inter-application resource contention or priority inversion (§2). Observing that the stability and performance cost of Linux Audit is ultimately dependent on the number of log events generated, we then set out to tailor Linux Audit to real-time applications, carefully reducing event logging without impacting the forensic validity of the logs. We present **Ellipsis**, a kernel-based log reduction framework that leverages the predictability of real-time taskset's execution profiles (§3). *Ellipsis* first profiles tasks to produce a template of their audit footprint. At runtime, behaviors that are consistent with this template are reduced. A task that deviates from the template is audited with lossless fidelity. Far from being impractical, we demonstrate a synergistic relationship between security auditing and predictable RTS workloads, faithfully auditing suspicious activities while incurring almost no log generation during benign typical activity. The contributions of this work can be summarized as:

- We conduct a thorough performance analysis of an exemplar commodity auditing framework, Linux Audit, to determine its impact on real-time applications (§2). We find that Linux Audit adds large overheads to each system call, which, if not considered in system design, would cause deadline violations in real-time applications under audit.

- We present *Ellipsis*, an audit framework and reduction system that is uniquely-tailored to real-time environments (§3). To the best of our knowledge, *Ellipsis* is the first RTS-aware auditing framework and the first to exploit the well-formed nature of real-time

tasks.

- *Schedulability Analysis:* to demonstrate the newfound practicality of system auditing in RTS, we conduct a thorough schedulability analysis of Linux Audit and *Ellipsis* (§4) along with an evaluation using the ArduPilot application (§5). We discover that *Ellipsis* is able to collect all audit information while meeting the application's temporal requirements.

- *Security Analysis:* we conduct a detailed security analysis to demonstrate that *Ellipsis* provides comparable security to Linux Audit at greatly reduced costs (§6).

To facilitate further exploration into the challenges and opportunities of RTS auditing, we will open source *Ellipsis* post publication.

CHAPTER 2: AUDIT FRAMEWORK ANALYSIS

Although it is well-established that Linux Audit can incur large computational and storage overheads in traditional software [37], its specific impacts on RT applications are unclear. Linux Audit not only adds additional latency to each system call as it generates log messages but also introduces the shared global kaudit buffer whose access is coordinated using a spinlock. These changes could potentially cause havoc on RT task sets as a result of changing execution profiles, resource contention or priority inversion [43]. In this section, we analyze the auditing framework's impact on RT task sets. Log maintenance tasks are not included in this analysis as the daemons, *kauditd* and *auditd*, that handle this run with a background priority and cannot directly impact the higher priority RT tasks.

2.1 LINUX AUDIT FRAMEWORK

The Linux Audit system [45] provides a way to observe and analyse system activities. While Linux Audit can be configured to monitor high-level activities such as login attempts [44], its primary utility (and overhead) comes from tracking low-level system calls, which is the focus of this paper. An overview of the Linux Audit architecture is presented in Figure 2.1. When an application invokes a system call ①, the subsequent kernel control flow eventually traverses an `audit_filter` hook ②. Linux Audit examines the context of the event and compares it to pre-configured audit rules, generates a new log event and enqueues it in a message buffer if there is a match ③ before returning control to the system call handler ④ and then to the application ⑤. Asynchronous from this workflow, a pair of (non-real-time) audit daemons, `kauditd` and `auditd`, that run in kernel and user spaces respectively, empty the message buffer to user space for storage, distribution and analysis. Because the transport of logs is asynchronous, it is possible for the kaudit buffer to overflow if system calls occur faster than the daemon flushes to user space, creating the potential for event loss.

2.2 EVALUATION

Setup. All measurements were conducted on 4GB Raspberry Pi 4 [46] running Linux 4.19. The kernel was sourced from raspberrypi/linux [47] with additional kconfig (CONFIG_PREEMPT_RT_FULL, CONFIG_AUDIT, CONFIG_AUDITSYSCALL) enabled. Audit rules were configured to only match against our benchmark application *i.e.*, background

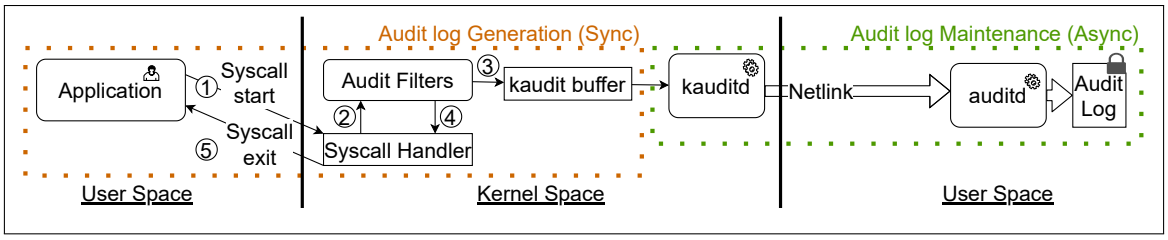

Figure 2.1: Architecture of Linux Audit Framework [44].

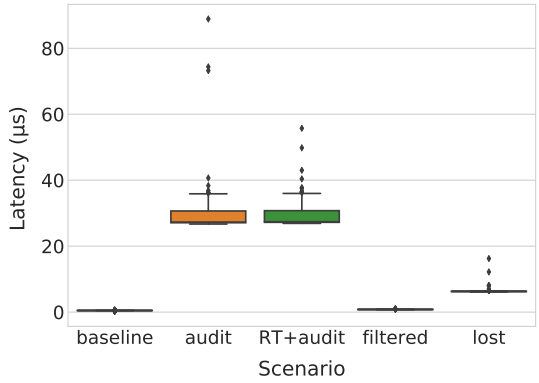

Figure 2.2: Latency distribution of executing `getpid` for various auditing scenarios.

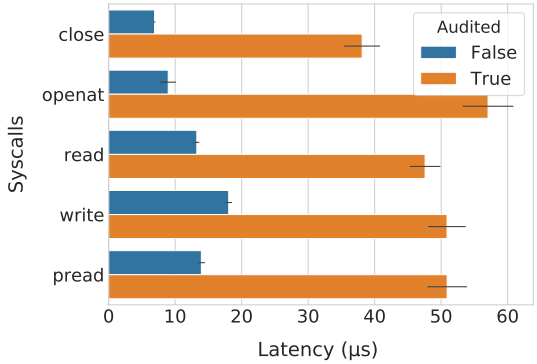

Figure 2.3: Overheads of auditing different system calls using the RT+audit scenario.

process activity was not audited. We borrow an aggressive audit configuration that captures all forensically-relevant events on the system [29, 32, 48, 49, 50, 51, 52].[1] Kaudit buffer size of 50,000 was used. To isolate the baseline latency of the Linux Audit framework, we use the `getpid` system call because it is non-blocking and does not require any arguments. To further reduce external perturbations, we disable power management, set CPU governor to performance [53] and direct all kernel background tasks/interrupts to core 0. In each case the latency is measured by taking the difference of monotonic timer counter (clock_gettime(CLOCK_MONOTONIC, t)) before and after the syscall. To reduce error, the latency to read the timer values is subtracted from the syscall latency.

Temporal Analysis. We first measure the overhead added by Linux Audit when processing an individual system call. Figure 2.2 shows the latency to execute the `getpid` system call in varying conditions. Each column shows a box plot of system call execution latency over 1000 iterations. The *baseline* case is the latency to execute the system call with auditing

[1] 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`.

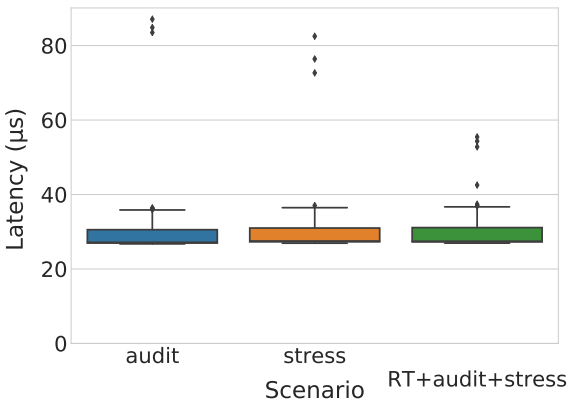

Figure 2.4: Overhead of auditing the `getpid` benchmark application with background stress.

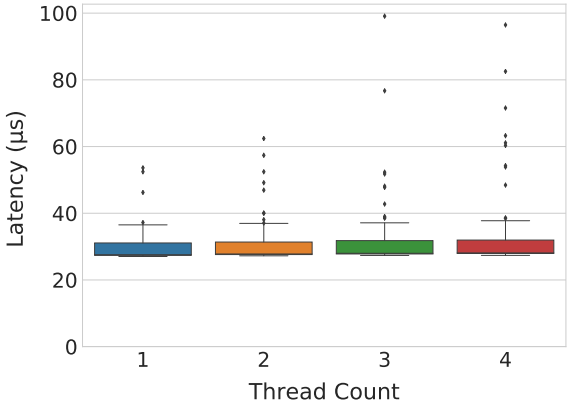

Figure 2.5: Latency of a system call with parallel executing threads.

disabled and no other application running. For the second *audit* scenario, auditing is enabled and every invocation of the system call is recorded by the audit system. For the *RT+audit* scenario, we execute the previous scenario with the application executing system calls under audit but running with a RT priority. The impact of auditing on system calls that are not included in audit rules and thus not traced eventually is shown in the *filtered* scenario. In all earlier scenarios, the kaudit buffer never overflows and complete logs are captured. *lost* shows the latency when kaudit buffer is full and all logs are lost. Figure 2.2 shows auditing overhead added to every system call execution. The observed maximum overhead was just under 100 μs. This observed max reduces to 60 μs when the application under audit is assigned a RT priority. If a system call is excluded from audit rules or the event log is lost due to the buffer being full, the overhead is much smaller at $\leq 1\mu s$ and $\leq 20\mu s$, respectively. Figure 2.3 shows the latencies (x-axis in μs) for *baseline* (Audited = False) and *audited* (Audited = True) scenarios for different system calls. Earlier observations hold true for different system calls.

Priority Inversion. We repeat the previous test with a non-real-time priority stress application running in parallel. The stress application executes the same `getpid` workload over 4 threads, and is also audited. Figure 2.4 reports the latency of system call execution *for the main benchmark application only.* Audited application running with high priority did not suffer any additional contention from any non-real-time audited workloads being run in parallel *i.e.,* no priority inversions were observed in this experiment.

Resource Contention. Linux Audit introduces a resource shared among RT tasks being audited, the kaudit buffer, which could lead to contention and blocking. 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. The threads are synchronized via a barrier to start executing at the same time. The benchmark has a tight loop that runs only the `getpid` system call in each thread, hence ruling out the processor cache or memory bandwidth as sources of contention. The timing variations for executing the system calls on the thread under observation is shown in Figure 2.5. In the average case, we observe only a small difference in the latency of `getpid` regardless of the parallel workloads. The observed worst case overhead is still under 100 μs, even when all 4 cores are using the buffer concurrently. This result intuitively follows as the shared spinlock covers a small critical section containing pointer manipulations.

2.3 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 kaudit buffer, this cost does not impact the average latency of auditing as the number of parallel threads increases. Further, except for a limited number of outlier cases, *the latency introduced by system calls can be measured and bounded.* This works well for latency sensitive RTS that RT Linux is intended for. However, the storage and management of these audit logs remains a significant challenge for resource-bound devices – for example, our brief experiments in §2.2 generated over 2 MB of audit logs in 50 ms. In the next section, we introduce an optimization for Linux Audit that mitigates this burden.

CHAPTER 3: ELLIPSIS

The volume of audit logs is a major limiting factor for their continued use on RTS. Reducing the volume is necessary for lossless auduting on resource constrained devices and also vastly reduces the average overhead of the Audit system [37]. In this section, we present *Ellipsis*, an audit reduction technique designed specifically for RTS. *Ellipsis* takes advantage of the predictive nature of RT tasks to learn templates of application behavior. Log messages are filtered at runtime if the task's behavior adheres to the template. We now describe *templates* (§3.1), our approach to *template creation* (§3.2) and *runtime matching* (§3.3).

3.1 TEMPLATES

Templates are representations of learned expected behaviors of RT tasks. They contain a sequence of system calls and their expected arguments. Let's say we have a single periodic task τ which we want to audit with *Ellipsis* and thus first need to create templates for. RT tasks are commonly structured with a one time *init* component and repeating *loops*. Let s_i denote a system call sequence the task exhibits in a *loop* iteration, where $0 < i \leq N$ and N is dependent on how many different execution paths τ might take. Let function $len(s_i)$ return the number of system calls in a sequence s_i. Let S denote the set of all such system call sequences that an instance of τ might exhibit, $S = [s_1, s_2, s_3 ... s_N]$. Let's further assume that each sequence s_i has a probability of occurrence, denoted by p_i.

Conjecture 3.1. N is finite for real time tasks and S can be fully determined.

Real Time tasks are developed to have limited code paths and bounded loop iterations. Complete static analysis is an important part of worst case analysis of real time tasks.

Conjecture 3.2. The distribution of p_i is highly biased *i.e.,* certain sequences s_i will have much higher probability of occurrence as compared to others.

Such repetition of certain sequences has been shown in prior works in literature [54] and case study shown later in this work (§5). Let's now assume that the sequences are arranged in decreasing order of probability of occurrence i.e. $p_1 \geq p_2 \geq p_3 ... \geq p_N$. First n sequences are chosen to be reduced by *Ellipsis* where $1 \leq n \leq N$. Let's say that the task τ executed for a large number of iterations I. Let's denote the number of events in *init* phase by f. Let B denote the length in bytes of each event when stored on disk. While $B_{Ellipsis}$ and B_{Audit} vary based on event types and arguments, the design of *Ellipsis* ensures that $B_{Ellipsis} < B_{Audit}$.

Size on disk for the log (L) for Linux Audit (L_{Audit}), *Ellipsis* ($L_{Ellipsis}$) and log size reduction can thus be calculated as:

$$L_{Audit} = I * B_{Audit} * \sum_{i=1}^{N}(p_i * len(s_i)) + f * B_{Audit} \tag{3.1}$$

$$L_{Ellipsis} = I * (B_{Ellipsis} * \sum_{i=1}^{n} p_i + B_{Audit} * \sum_{i=n+1}^{N}(p_i * len(s_i))) + f * B_{Audit} \tag{3.2}$$

Size reduction — Audit log size for n sequences

$$L_{Audit} - L_{Ellipsis} = I * (B_{Audit} * \sum_{i=1}^{n}(p_i * len(s_i)) - B_{Ellipsis} * \sum_{i=1}^{n} p_i) \tag{3.3}$$

Iterations — *Ellipsis* log size for n sequences

vs These log size savings come at the cost of storing the template in memory at runtime:

$$Runtime\_Memory\_Overhead = 36 * n + 44 * \sum_{i=1}^{n} len(s_i) \tag{3.4}$$

where 36 and 44 are parameters calculated on 64 bit Linux kernel. Note that Linux Audit and *Ellipsis* utilize a memory buffer, kaudit buffer (Fig. 2.1). This buffer is not considered in Eq. 3.4 but is explored in Section 5.4. As evident from Eq. 3.3, applications with long repeating sequences benefit most from *Ellipsis*. For any sequence that is not included as a template, *Ellipsis* behaves identical to Linux Audit. In this work we use dynamic analysis only to find the most frequently occurring sequences. Parameters from the detailed case study presented in Section 5 are included in Table 3.1 as an example. Detailed templates from the case study are provided in Appendix A.

Extended Horizon for Log Reduction. Till now we have limited the horizon of log reduction to individual task *loop* instances. If the system is continually behaving within the parameters encoded in the templates, the horizon for log reduction can be extended to multiple instances of a task. Many contiguous instances of the task yield a single audit message which captures *(a)* the start time of the first instance *(b)* end time of the last instance and *(c)* the number of task instances reduced within this entry. The maximum number of instances that can be reduced together is configurable and can be as assigned by the system administrator or possibly the application hyper-period. Whenever a reduction check fails, a single log message is generated for previously reduced instances in addition to the complete log for the current instance. This higher performance system is henceforth referred to as *Ellipsis-HP*. At worst *Ellipsis-HP* behaves identical to *Ellipsis* while in the best case where all repetitions of a sequence happen contiguously, *Ellipsis-HP* will capture all sequence repetitions in one

Table 3.1: 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

entry.

$$L^{Best}_{Ellipsis\text{-}HP} = B_{Ellipsis} * n + B_{Audit} * I * \sum_{i=n+1}^{N}(p_i * len(s_i)) + f * B_{Audit} \quad (3.5)$$

$$L_{Audit} - L^{Best}_{Ellipsis\text{-}HP} = B_{Audit} * I * \sum_{i=1}^{n}(p_i * len(s_i)) - B_{Ellipsis} * n \quad (3.6)$$

Temporal Constraint. RTS are sensitive to time intervals between events, thus, *Ellipsis* also considers temporal checks in the template matching process. For *Ellipsis* each templatized sequence needs to finish within a pre-determined time interval for it to be reduced at runtime. This interval can be learned when templates are created or can be a constraint provided by the system administrator. *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. The impact of temporal constraints on log size reduction are explored further in Section 5.3.

3.2 TEMPLATE CREATION

There are two stages to template creation: *(a)* identification of sequences of system calls and their arguments and *(b)* profiling temporal behavior. Figure 3.1 provides an overview of this automated offline processs. In the first training stage, *Ellipsis* captures traces over multiple executions of the application. These traces can be obtained from a variety of sources, *e.g.*, `strace` [55] or from the audit logs themselves. To effectively delineate between threads, our tracer implementation modifies Linux Audit to include the Thread ID in log messages. The goal of the *Ellipsis* templates is to capture the cyclic system call behavior of periodic tasks. The *challenge* of identifying repetitive execution behavior is actually common in the auditing literature [49, 56, 57, 58] but past solutions require binary analysis, code annotations, stack analysis or a combination. Instead, we observe that periodic tasks typically end with calls to `sleep` or `yield` that translate to `nanosleep` and `sched_yield` system calls in Linux. Periodic behaviors can also be triggerd by polling timerfds to read events from multiple timers by using `select` and `epoll_wait` system calls. We leverage these system calls to identify boundaries of task executions and then extract a sequence

of system call invocations within each task execution. These system call sequences are then converted into intermediate thread-level templates, each entry of which includes the system call name along with the passed arguments. The existence or usage of boundary syscalls is not compulsory though. We only use them to identify loops without modifying the application. However Linux Audit supports custom audit messages that can be added to applications. So any application can be modified to explicitly mark the start and end of repeating loops.

This process yields the per task syscall sequences exhibited by the application and their properties i.e. length and probability of occurrence. Based on desired log reduction vs memory tradeoff (§3.1) a subset of sequences can be chosen to create corresponding templates. If there is a requirement to create templates for all possible valid syscall sequences, static analysis tools can be used to enumerate all code paths and create templates accordingly. We infer task runtimes by auditing the application using *Ellipsis* configured with the intermediate templates obtained previously. *Ellipsis* validates these learnt behaviors to selectively disable log reduction for task instances that experience delays to allow for investigations.

3.3 RUNTIME MATCHING

Given the template of system call sequences that periodically occur in an application, we designed a kernel module that extends Linux Audit to filter system calls that are described by the *Ellipsis* template. We extend the Linux Audit command-line utility `auditctl` to transmit template data into kernel space. The templates corresponding to an application's behavior are modeled as a finite state automaton (FSA) as shown in Fig 3.2. We implemented the template matching feature to complement the system call filtering performed by Linux Audit, thus performing log reduction down at the source of log events.

While the periodic task is executing, all system calls that are 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 log message onto the main kaudit buffer to denote the execution of a templatized task. However, *Ellipsis* enqueues the entire task-specific buffer to the main kaudit buffer if *(a)* a system call occurs that is not allowed by the automaton, *(b)* if the template is not fully described at the end of the task instance or *(c)* if the task instance does not adhere to the expected temporal behavior of the fully described template. Using this runtime matching routine, *Ellipsis* reduces the storage overhead of each task period to a single log message, thus dramatically improving the average case. However, if the application exhibits anomalous or suspicious behavior, *Ellipsis* ensures that forensic evidence of the events is retained in lossless fidelity.

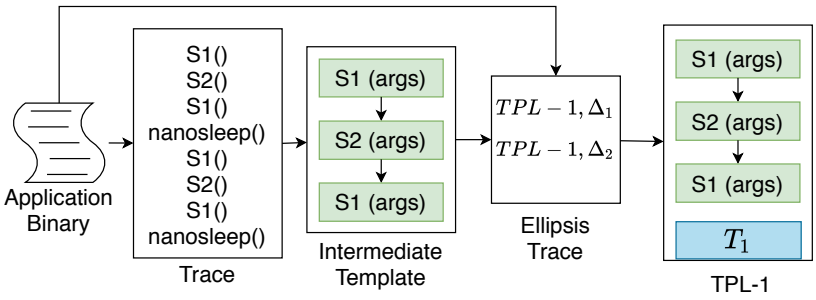

Figure 3.1: Overview of the *Ellipsis* template generation workflow. In the first stage the application's system calls are traced to identify repeating sequences. Syscalls are denoted by S1, S2, S3. In the second stage the application is audited with *Ellipsis* using the intermediate template. The second stage creates a profile for tasks execution time (Δ_1) using *Ellipsis* audit logs that is then used to enrich the template with temporal constraints (T_1) .

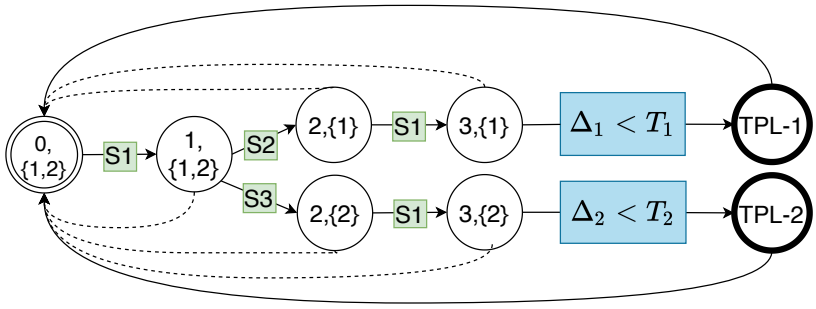

Figure 3.2: Runtime template matching is modelled as a Finite State Automaton (FSA). *Ellipsis* FSA states are numbered as x,{y} where x is the number of syscalls matched while y is the set of templates that are reachable from this state. This example shows TPL-1 from Fig. 3.1 and an additional TPL-2 that shares first syscall with TPL-1 but diverges by calling S3 after first S1. *Ellipsis* starts in state (0,{1,2}) prior to observing system calls present in the two configured templates. System call invocations trigger state transitions and following the entire sequence identified by a template while satisfying temporal constraints leads to an accept state (TPL-1,TPL-2). Any unexpected behavior causes *Ellipsis* to write stored audit logs and return to the start state, as shows with the dotted transitions.

CHAPTER 4: SCHEDULABILITY ANALYSIS

This section presents a real-time schedulability analysis for Linux Audit and *Ellipsis*. The base scenario in Fig. 4.1 shows execution timeline for two periodic tasks that are not under audit. Audit overheads can be divided into two parts: *(i)* $A(\cdot)$ represents the task execution with additional synchronous overhead of log generation and *(ii)* $B(\cdot)$ represents the processing time required to maintain the audit logs, transporting them from kaudit buffer to userspace and eventually to persistent storage (Fig. 2.1). Audit log maintenance (Task $B(\cdot)$) is composed of `kauditd` and `auditd` daemons that run with background priority. Task $B(\cdot)$ varies with the number of log events that need to be maintained. Any additional overheads of log maintenance, like transporting them to a remote server can also be trivially included in this component. We now analyze the schedulability when tasks are audited.

Let us represent the WCET of each task τ_i as $C_i^{'} = C_i + C_i^A$ where C_i^A is the additional (worst-case) computational time for log generation. The overhead C_i^A for Linux Audit has been analysed in §2. Let us represent the taskset as $\Gamma = \{\tau_i(C_i^{\mathrm{I}}, T_i, D_i)\}$ Auditing would not cause any deadline violations if the WCRT of each task $\tau_i \in \Gamma_{\mathrm{I}}$ (denoted by R_i) is less than or equal to its deadline ($R_i \leq D_i$). The response time calculation, hence the schedulability of task under audit, therefore can be obtained by standard multicore global fixed-priority scheduling analysis techniques [59, 60] and represented by: $R_i = \min\{x\}$ s.t.: $x \leq C_i^{\mathrm{I}}$ and $\sum_{\tau_k \in hp(\tau_i)} I_i^k(x) < M(x - C_i^{\mathrm{I}})$, where $hp(\tau_i)$ represents the set of tasks with a priority higher than τ_i and $I_i^k(\cdot)$ is the interference experienced by τ_i from a higher-priority task τ_k.

Recall that the audit buffer is filled by the real-time tasks at runtime and needs to be periodically cleared and saved in persistent storage. Task $B(\cdot)$ responsible for this behaves similar to garbage collectors that have been studied in prior works in literature [61]. We assume that the buffer is large enough to hold logs before consecutive invocations of the buffer draining task (see §5.4 for further analysis on buffer usage). We further assume that the buffer draining mechanism executes with the lowest priority (*i.e.*, it does not interfere with the existing real-time tasks). Let us now consider the augmented taskset $\Gamma \cup \{\tau_B\}$ where the task τ_B is responsible for draining the logs in audit buffer to the user space audit daemon (*kauditd*) and any post processing including persistent storage of the log (*auditd*). Recall that by assumption the buffer is large enough to hold logs for one hyperperiod. The hyperperiod is the least common multiple (LCM) of the periods of each task $\tau_i \in \Gamma$ and we denote this as $LCM(\Gamma)$. The buffer draining mechanism will be invoked once in each hyperperiod, *i.e.*, $T_B = LCM(\Gamma)$ and needs to complete its execution before its next periodic interval (*i.e.*, $D_B = T_B$). We can audit this system (*i.e.*, there will be no log loss) if the

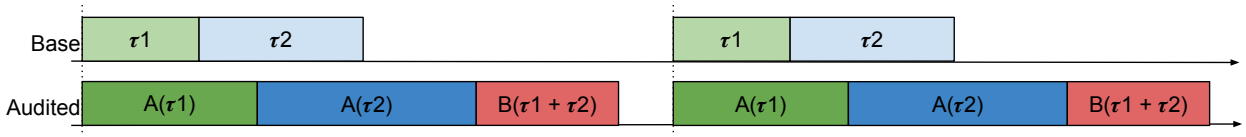

Figure 4.1: Sample timelines for two periodic tasks $\tau 1$ and $\tau 2$. $A(\cdot)$ is the increased computation time of the task including audit log generation overhead. $B(\cdot)$ represents the runtime of *kauditd* and *auditd* daemons as they handle the logs generated by the real-time tasks.

WCRT of the buffer draining task is less than its period, *i.e.*, $R_B \leq T_B$. We assume here that τ_B is the only background task in the system. However, if other such tasks exist their interference can be accounted for.

Let us now consider a window of length x (*e.g.*, a time interval $[t_1, t_2)$ such that t_1 is the arrival time of a job of τ_B, t_2 is a generic value less than or equal to $t_1 + T_B$ and $x = t_2 - t_1$). $I_B^i(x)$ is the interference from a higher-priority real-time task τ_i within the window x — this is the cumulative time in which the buffer draining task can not execute because of the execution of τ_i. Note that τ_B cannot execute during the collection of intervals when all M cores are occupied by the real-time tasks. The cumulative length[1] of this interval is $x - C_B$ [62]. Therefore a sufficient condition the schedulability of the audit draining task is given by: $\sum_{\tau_i \in \Gamma} I_B^i(x) < M(x - C_B)$ where $\sum_{\tau_i \in \Gamma} I_B^i(x)$ represents the total interference from real-time tasks. Let us assume that there exists a function $\mathcal{R}_B(\cdot)$ that returns the minimum value of x with following schedulability conditions:

$$x \geq C_B \text{ and } \sum_{\tau_i \in \Gamma} I_B^i(x) < M(x - C_B), \tag{4.1}$$

i.e., $\mathcal{R}_B(\cdot) = \min\{x\}$ if Eq. (4.1) holds; otherwise $\mathcal{R}_B(\cdot) = \infty$. By definition the value of $\mathcal{R}_B(\cdot)$ is also an upper bound for the WCRT of τ_B and the calculation of the functions $I_B^i(x)$ and $\mathcal{R}_B(\cdot)$ can be obtained by standard global multicore scheduling analysis [59, 60]. The logging mechanism therefore will work as expected if the following conditions hold: $\mathcal{R}_B(\cdot) \leq T_B = LCM(\Gamma_{\text{II}})$ and $R_i \leq D_i, \forall \tau_i \in \Gamma_{\text{II}}$. However, if $\mathcal{R}_B(\cdot) > T_B$ the accumulation of logs in buffer would eventually lead to $\mathcal{R}_B(\cdot) = \infty$, overflowing the buffer.

4.1 SYNTHETIC WORKLOAD

Experiment. To study the impact of auditing we perform a schedulability and auditing experiment, comparing Linux Audit and *Ellipsis-HP*. Tasksets (Γ) consist of upto 5 tasks

[1]This total interval length $x - C_B$ may not necessarily be contiguous.

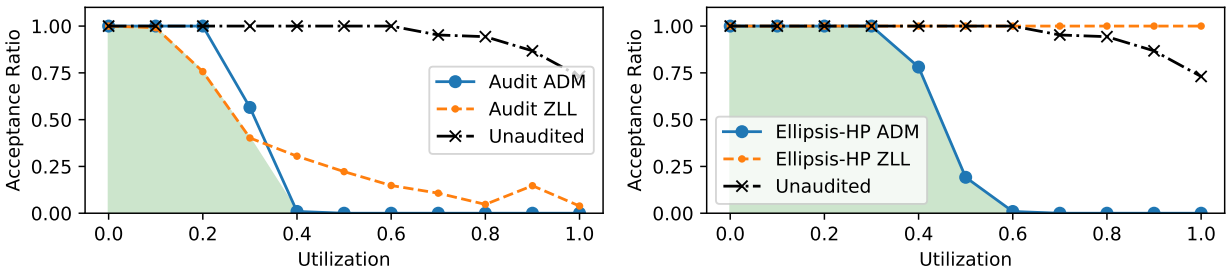

Figure 4.2: Temporal and Auditing constraint analysis for periodic tasksets. Taskset utilization (X axis) are plotted against the percentage of tasksets (Y-axis) for which All Deadlines are Met (ADM) or have Zero Log Loss (ZLL). Tasks have 5% system call and 95% non-system call workload. The shaded region shows the area where both ADM and ZLL requirements are satisfied. The schedulability or ADM for Unaudited case is provided as a reference.

(τ) with utilization chosen uniformly $\in [0, 1]$. Tasks have a period $\in [100, 1000]\mu s$ and have a utilization selected using UUniFast algorithm [63]. Each task's computation time is divided into 95% busy wait and 5% audited syscalls. The syscalls are $\in$ `[getpid, getppid, getgid32, getuid32, getpgrp]`. These syscalls were chosen for their stable execution profiles. However their small execution time means that the synchronous overhead component $A(\cdot)$ disproportionately impacts them. The evaluation setup as described in Sec 2.2 is used. Additionally, tasks are isolated to core 1, while interrupts and kernel processes (including `kauditd`) run on core 0. Other cores are disabled. The audit buffer is set to be large enough to hold exactly one hyperperiod worth of audit information. All tasks are started together and each τ_i runs for $2 \times LCM(\Gamma)/T_i$ iterations. Tasks are executed with rate monotonic priority. This ensures that if the second hyperperiod of the taskset starts with some audit logs in kaudit buffer from the previous hyperperiod, log loss will occur. For each taskset we monitor whether all tasks meet their temporal requirement *i.e.,* All Deadlines Met (ADM). We also monitor whether any audit logs are lost *i.e.,* Zero Log Loss (ZLL). 1000 tasksets are run, the results for tasksets are grouped together by utilization in groups of 0.1. Y axis measures fraction of the tasksets in each group for which all deadlines were met (ADM) and zero logs were lost (ZLL).

Observations. Figure 4.2 shows the results. The synchronous $A(\cdot)$ component impacts ADM. *Ellipsis-HP* performs better on ADM metric allowing all tasksets with upto 0.3 utilization to meet all deadlines whereas Linux Audit sees deadlines miss starting at 0.3 utilization. The main point of difference is the $B(\cdot)$ component that deals with maintenance of auditing information; log loss occurs when this component does not get enough computation time. Linux Audit starts experiencing log loss at very small utilizations. When deadlines

are missed, the total runtime of the taskset goes beyond 2 hyperperiods but Linux Audit is unable to avoid log loss despite the additional time. In stark contrast, *Ellipsis-HP* is able to *achieve lossless auditing for all workloads*.

Discussion. Although Linux Audit has been included in embedded Linux its impact on schedulability of real time tasksets requires careful analysis. In this experiment, despite the low ratio of the task workload being audited (5%), Linux Audit is able to meet deadlines for all tasksets with utilization upto 0.2 only. This is the trade-off for the security benefits of auditing. Surprisingly, Linux Audit starts seeing loss of auditing information as taskset utilization goes above 0.1. *Ellipsis-HP* improves this significantly by satisfying both ADM and ZLL requirements for tasksets of upto 0.3 utilization. Section 5.5 further explores these overheads for an autopilot application.

CHAPTER 5: PERFORMANCE EVALUATION

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 [64]. It has been installed in over a million vehicles and has been the basis for many industrial and academic projects. We chose the quadcopter variant of ArduPilot as it has the most stringent temporal requirements within the application suite. All evaluations are conducted on the setup described in Sec. 2.2. A Navio2 module board containing intertial sensors required by ArduPilot was attached on the Raspberry Pi board. The kaudit buffer size was set to 50K as it was found to be the highest stable configuration possible for the evaluation platform. Among the seven tasks spawned by Copter, we focus primarily on a task named Fast Loop for evaluating temporal overheads as it includes the Copter stability and control tasks that need to run at a high frequency to keep the QaudCopter stable and safe.

Among the system calls observed in the trace of Copter, we found that only a small subset of system calls were relevant to forensic analysis [48], *viz.,* `execve, openat, read, write, close` and `pread64`. Upon running the template generation script on the application binary, we obtained templates for three tasks 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 system calls as well as temporal constraints. Templates were loaded into the kernel before we evaluated *Ellipsis* or *Ellipsis-HP*. Linux Audit was set up to capture invocations of the system calls made by the Copter application as mentioned above. Complete templates are provided in Appendix A.

5.1 AUDIT COMPLETENESS

Experiment. We first evaluate whether complete audit information can be captured for this application. We ran the application for 100K iterations for task frequencies of 100, 200, 300 and 400 Hz as defined in the application source code.

Observations. Fig. 5.1 compares the logs lost count for Linux Audit, *Ellipsis* and *Ellipsis-HP* across multiple task frequencies. We observe that Linux Audit is unable to provide lossless auditing for the desired application periodic frequency of 400 Hz. Only when the application frequency was lowered to 100 Hz could Linux Audit capture complete logs.

Discussion. This task performs critical stability and control function so reducing the task frequency to accommodate auditing can have catastrophic results. On the other hand, there is no need for such a trade off with *Ellipsis* as it can provide lossless auditing for the entire

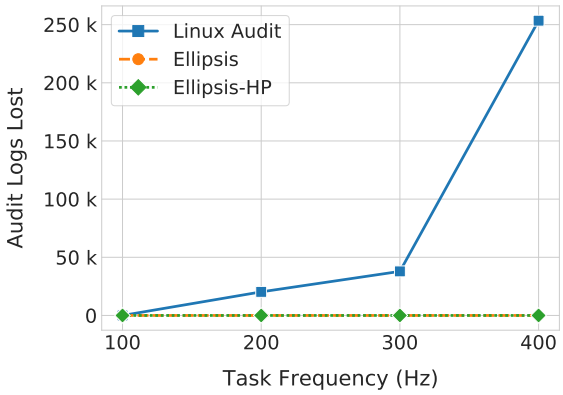

Figure 5.1: Number of audit logs lost for various frequencies of the primary periodic loop in the ArduPilot application over 100K iterations.

range of application frequencies, as shown in Figure 5.1.

To explore the trade-off of audit completeness vs an application's temporal requirements, the task was run again at 400 Hz. Auditing was allowed to insert waits whenever the kaudit buffer is full by utilizing a configuration provided by Linux Audit, *audit_set_backlog_wait_time*. These waits did allow Linux Audit to log all audit information but led to *missed deadlines*. Hence, Linux Audit would force a trade-off between ArduCopter's temporal requirements and lossless auditing, whereas *Ellipsis* achieves both.

5.2 AUDIT LOG REDUCTION

Experiment. For measuring the volume of audit logs written to disk, we ran the application over multiple iterations in 10 to 100K range to simulate application behavior over varying runtimes. Figure 5.2 compares the storage costs in terms of log files size on disk in bytes.

Observations. 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 tends to dominate over the periodic behavior of tasks. Over a 250 second runtime the growth of logs 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 reduction option by lowering storage costs by 860MB, or **93%**, compared to Linux Audit. *Linux Audit Lossless* estimates the log size had there been no log loss.

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.

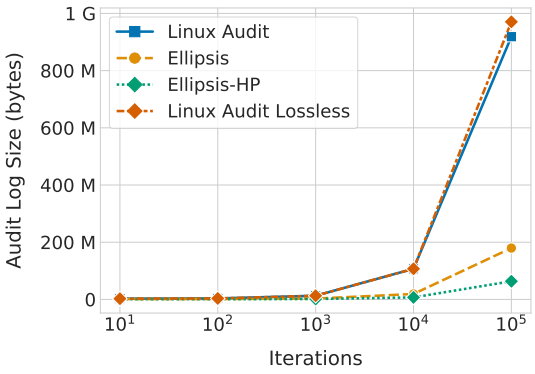

Figure 5.2: Total size of the audit log, as size on disk (Y-axis), captured for different number of iterations (X-axis).

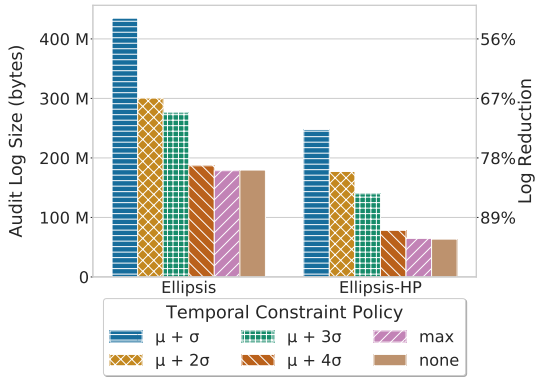

Figure 5.3: Log size (Y-axis left) under varying temporal constraint policies. The right Y axis shows the % reduction in log size compared to Linux Audit (§5.2). $\mu + 4\sigma$ covers 99.5% of the total 100k iterations.

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 both longevity of the system and reducing the performance costs.

5.3 TEMPORAL CONSTRAINT POLICY

Experiment. In this experiment we explore the impact of different policies for temporal constraints. As described in Chapter 3, 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. The *max* policy is used in all other experiments in this section. 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. Fig. 5.3 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* policy yield the same log size, which is expected given that temporal determinism is a design feature of RT applications.

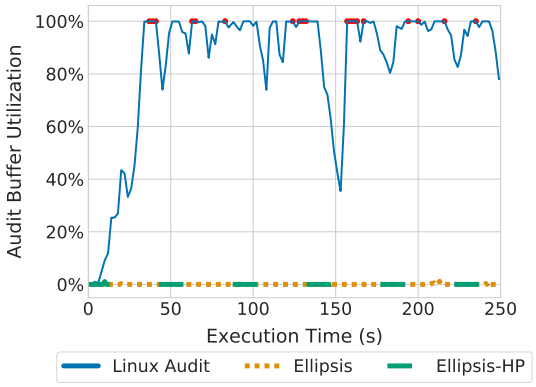

Figure 5.4: Audit buffer utilization over time. Additional red annotations signify all times where buffer is completely filled. *Ellipsis* and *Ellipsis-HP* did not use more than 2% buffer space.

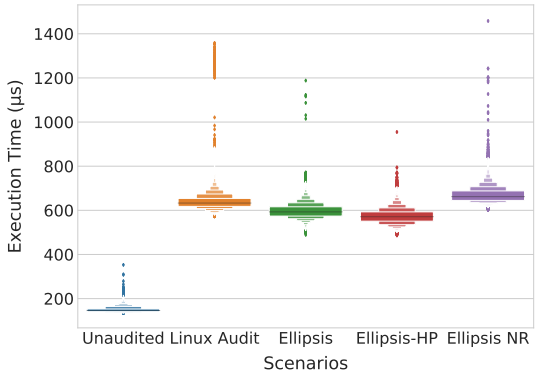

Figure 5.5: Comparison of runtime overheads. Task period and deadline is 2500 μs.

Discussion. The timing constraints are decided based on the observed values from the learning phase. Learning phase behavior is considered *correct* as this phase is a controlled execution. Hence we believe that *Ellipsis* should be used at runtime to record unexpected behaviors i.e. not seen during learning phase while eliminating audit logs for all expected behaviors. The policies *max* and $\mu + 4\sigma$ most closely correspond with this recommendation. Further is it notable that since *max* and *none* policy yield almost the same log size, under normal operation, the *max* constraint comes with negligible overheads.

5.4 AUDIT BUFFER UTILIZATION

Experiment. 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 5.4 shows the comparison of the percentage utilization of the audit buffer by Linux Audit, *Ellipsis* and *Ellipsis-HP* over time.

Observations. From Fig. 5.4, we see that for Linux Audit, the utilization of the kaudit buffer rises quickly and remains close to 100% for the majority of the running time, resulting in loss of audit messages as measured earlier (§5.1). In contrast, *Ellipsis* and *Ellipsis-HP* ensure that the buffer utilization remains negligible throughout the execution.

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 audit logs being generated and thus reducing the number of outstanding audit logs buffered in the system.

The variations that we see in the plots can be attributed to the scheduling of the non real-time *kauditd* thread that is responsible for sending the outstanding audit messages to user-space for retention on disk. We observe that the backlog builds with time when *kauditd* isn't scheduled and drops sharply when *kauditd* eventually gets CPU time.

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. 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 [65].

5.5 RUNTIME OVERHEADS

Experiment. Finally, we measure the runtime overheads. Figure 5.5 shows the execution time in microseconds, for the Fast Loop task of Copter under various conditions. The execution time for each scenario was measured by running the task for 1000 iterations with a kaudit buffer size of 50K to facilitate collection of all audit logs. The 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. In *Ellipsis* NR (No Reduction), we instrumented the template to force the reductions to always fail, due to a forced mismatch at the very last entry in the template. In this extreme, synthetic scenario, *Ellipsis* is forced to do the longest possible match before eventually failing to reduce the log. When all reductions fail, *Ellipsis* and *Ellipsis-HP* are equivalent and hence *Ellipsis* NR is the worst case scenario for both systems.

Observations. *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. *Ellipsis* NR shows an *1.05*x increase in average overhead compared to Linux Audit. The observed maximum overheads show a greater improvement. *Ellipsis*'s observed

maximum overhead is *0.87*x and *Ellipsis-HP*'s observed maximum overhead is *0.70*x of Linux Audit. *Ellipsis* NR shows an *1.07*x increase in maximum observed overhead.

Discussion. When audit logs are successfully reduced by *Ellipsis*, the total work done to collect audit information is reduced. This in turn reduces the overhead of auditing. This effect is further amplified in *Ellipsis-HP* owing to its greater log reduction potential (§5.2). However, when the reduction fails, there is a small increase in overhead due to the extra comparative checks done to determine whether the logs can be reduced validly. This effect is visible in the *Ellipsis* NR case where all reduction attempts are instrumented to fail. For any application the actual runtime overhead change will depend on what proportion of the audit information is reducible. However, as evident from *Ellipsis* NR the worst case of no reduction is still not significantly worse than Linux Audit.

5.6 SUMMARY OF RESULTS

Ellipsis provides complete audit log retention while meeting temporal requirement of the ArduPilot application, using minimal memory and without any significant increase in the overhead of auditing. *Ellipsis-HP* further improves this by increasing the log reduction ratio.

CHAPTER 6: SECURITY ANALYSIS

We adopt a threat model similar to that used in related work [32, 66, 67, 68, 69], i.e., we include the underlying OS and the audit subsystem in the trusted computing base (TCB). We focus on attackers that aim to exfiltrate sensitive data from or degrade the performance of RT systems. For example, an attacker may install modified target programs, exploit a running process or install malware on the target system to achieve their objectives. Additionally, we require that attacks leverage syscalls so that they can be observed by the audit subsystem.

Further, we assume that attackers cannot compromise the integrity of audit logs. Although audit integrity is an important security goal, it is orthogonal to our objectives and can be enabled by building upon existing secure auditing frameworks [50]. Applications are assumed to be benign during the offline *Ellipsis* training phase. Target applications are the only processes of interest running on the system and are the only processes audited by the audit subsystem. Specifically, we leverage an aggressive audit configuration identified in prior work that is intended to capture all forensically-relevant events [29, 32, 48, 49, 50, 51, 52].[1]

6.1 DATA EXFILTRATION ATTACK

To demonstrate that *Ellipsis* retains critical forensic information in lossless fidelity while reducing benign application behavior, we reproduce a data exfiltration attack on Motion [70], an open-source video surveillance application that was presented in prior work [54].

Attack Scenario Motion [70] monitors camera images and detects motion by tracking changes between image frames. It is used for surveillance and can be configured to capture video or still images when motion is detected. The application contains a main loop that captures an image frame and saves it to a user configured location on detecting motion.

In this attack, the attacker inserts malicious code shown in Fig. 6.1 into the target application to save image frames to an attacker controlled location when motion is detected. The attacker can exploit another process running on the system in order to exfiltrate these sensitive images out of the system at a later point in time. Fig. 6.2 visualizes the entire attack scenario.

[1] 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`.

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

Figure 6.1: Attacker code that copies image frames to a malicious location. `event(...)` is a library function that internally uses `conf.target_dir` to determine where the images are to be written.

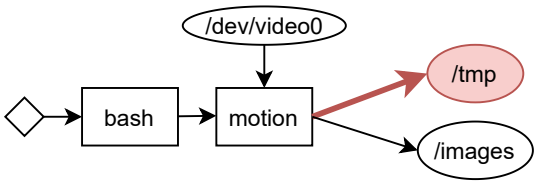

Figure 6.2: Attack graph describing the data exfiltration attack on Motion. Audit logs capture information flows from the video source (/dev/video0) to the expected location (/images) and the adversary controlled location (/tmp).

Experimental Setup We use the same setup as described in Sec 2.2, running Motion v4.3.2 using an external USB webcam as a video source. 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. Upon running the *Ellipsis* template generation script we obtained two templates that describe how Motion *(i)* captures an image frame and *(ii)* captures an image frame describing movement and saves it to the file system.

For this experiment, *Ellipsis* is configured with templates learned from the benign target application. We capture system audit logs while the malicious application executes for 5 minutes and introduce movement in the camera's field of view to trigger writes 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 remains unchanged in the malicious application. As the attacker inserts code to copy image frames describing movement, *Ellipsis* observes additional occurrences of `openat, write` and `close` system calls that differ from previously observed benign behavior. Therefore, *Ellipsis* retains all audit logs generated in response to observed movement in lossless fidelity. As *Ellipsis* matches only one template and the attack is ongoing throughout the experiment, we observe only a 25% reduction in

audit log size with *Ellipsis* compared to vanilla Linux Audit when the same number of images are written to disk. However, if the attack was made stealthier by selectively exfiltrating images of interest, we would have observed close to optimal levels of reduction.

6.2 THROTTLE OVERRIDE ATTACK

We now reproduce an attack on throttle signals on ArduPilot [64] presented in prior work [71] to show that *Ellipsis* helps in capturing attack behavior without log loss on resource constrained systems by aggressively reducing known benign behaviors.

Attack Scenario Autopilot applications such as ArduPilot are responsible for ensuring the safe operation of unmanned autonomous vehicles such as drones. The autopilot manages the rotary speed of motors using actuation signals that are periodically updated to ensure a fast and consistent response required for controlled flight.

Let's consider a stealthy attacker who wants to destabilize or take over control of unmanned drones. To achieve this objective, the attacker first gains control of a task on the system and attempts to override the control signals. A brute force approach of continuously overriding the actuation signals is not very effective in this case as the computational overhead of the attack can cause other tasks to miss deadlines, potentially leading to a system crash. Therefore, the attacker leverages side channel attacks such as Scheduleak [71] 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, thus effectively taking control with low overhead.

Experimental Setup We use the same setup described in Chapter 5 and choose the quadcopter variant of ArduPilot *i.e.,* Copter as the target application, for our experiment. We use attack scripts provided by authors of Scheduleak [71] to generate override signals on the same interface as used by Copter to send actuation signals. Furthermore, we configure the audit subsystem to audit forensically relevant syscalls from the target application and the attacker controlled task. *Ellipsis* is configured with 3 templates describing expected behavior for the Copter application. We execute the attack script for 250 seconds while Copter is running and capture audit logs for analysis.

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. *Ellipsis* preserves detailed attack behaviors for these syscall sequences as they do not match

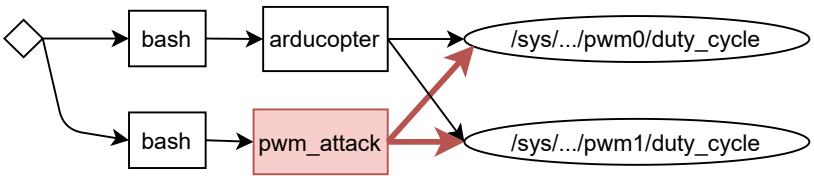

Figure 6.3: Attack graph describing the throttle override attack. Audit logs capture benign information flows from ArduCopter as well as malicious flows from the attacker controlled process (pwm_attack) to the PWM interface.

known templates while reducing expected benign behavior from the Copter application. Thus, *Ellipsis* supports lossless auditing throughout the application runtime. In contrast, vanilla Linux Audit loses audit logs potentially resulting in loss of forensic evidence.

Timing based side channel attacks such as Scheduleak [71] rely on frequent invocations of the `clock_gettime` system call to infer task activation times. Such system calls are irrelevant for forensic analysis as they don't capture critical information flows and are excluded from audit rules to reduce overheads. Despite the lack of visibility in the reconnaissance phase of the attack, *Ellipsis* and Linux Audit can capture evidence of attacker interference that creates new information flows relevant for forensic investigations as shown in Fig. 6.3.

6.3 SUMMARY

We observe that when a malicious process deviates from the expected behavior of a benign task, *Ellipsis* provides the same security as Linux Audit by aborting log reduction and recording all events. Additionally, *Ellipsis* all but eliminates the possibility of losing portions of the malicious activity due to log buffer overflow. However, if the malicious process adheres to the expected behavior of the benign task, the associated logs will be reduced.

The question, then, is whether the malicious process can perform meaningful work for the attacker while adhering to the benign template. If *Ellipsis* exclusively matched against system call IDs, such a feat may be possible; however, *Ellipsis* carefully inspects the arguments and temporal constraints of each system call when generating templates, making it highly unlikely that they could match against malicious actions. For example, an attacker might try to substitute a read from a non-sensitive file with a read from a sensitive file; however, doing so would require changing the file path argument of an `open` system call, which would violate the template. As a result, we argue that *Ellipsis* provides comparable security to commodity auditing frameworks at greatly reduced costs. Finally, we note that the original contents of the system log can be largely reconstituted. We refer interested readers to Appendix B for a demonstration.

CHAPTER 7: DISCUSSION

System Scope. While the analysis and evaluation are based on specific systems and scheduling policies, it should be noted that *Ellipsis* is useful for any application that has predictable repeating patterns. Further any real-time applications that can run on Linux can leverage Linux Audit and *Ellipsis*. This does not include safety-critical applications as Linux itself is not suitable for them.

Log Storage. Auditing can create large volumes of log (§5.2). This is specially concerning for RTS that have low storage capacity. Linux Audit supports remote log storage with auditd-remote plugin [72] or rsyslog [73]. However such support does not avoid log loss when the log buffer overflows (§5.4). Hence *Ellipsis* is a vital technique to audit RTS without losing any audit events, regardless of where logs are stored.

External Triggers for Aborting Log Reduction. *Ellipsis* aborts log reduction when an unexpected system call behavior is detected, but could also be adapted to support additional failure conditions. For example, log reduction could be aborted if the scheduler detects a deadline violation, or if a dedicated fault detection system raises an alert [2, 74].

Ellipsis as an Intrusion Detection Mechanism. *Ellipsis* compares the runtime behavior of real-time applications to learned templates; it is interesting to consider the possibility of *Ellipsis* templates being used as a security indicator for intrusion detection. As *Ellipsis* can detect variations in both timing behavior and system call usage, it may prove an effective complement to existing RTS intrusion detection mechanisms. For example consider a side channel attack like Scheduleak [75]. In this attack, the malicious task monitors its runtime by repeated system calls to get the current time. This behavior can be readily identified within the audit log and will present as a significant deviation from the template for a task.

CHAPTER 8: 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 [23, 27, 69, 76] and taint tracking [25, 77], 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 [24, 26, 28, 29, 30, 31, 32, 33, 34, 49, 78, 79, 80, 81, 82, 83, 84]; in this work, we make the application of such techniques applicable to RTS through the design of an 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 [28, 32, 34, 49, 57], 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, the well-formed nature of real-time tasks ensures the correctness of our execution units (*i.e.*, templates).

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 [51, 52, 58, 77, 80, 85, 86, 87, 88]. Of these, Ma et al.'s KCAL [37] and ProTracer [25] 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 [88]), but it is unclear whether these assumptions hold for cyber-physical environments. For example, KCAL would reduce multiple identical read syscalls to a single entry. However, a large number of extra reads may be cause deadline misses in an RTS and hence should not be reduced for RT applications under audit. To our knowledge, *Ellipsis* is the first system to explicitly address the challenges of auditing RTS.

Auditing RTS. Although auditing has been widely acknowledged as an important aspect of securing embedded devices [39, 40, 41], challenges unique to auditing RTS have received only limited attention. Wang et al. present ProvThings, an auditing framework for monitoring IoT smart home deployments [89], 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 [90]. Their embedded device emulates a USB flash drive, but does not consider system call auditing of real-time applications. Wu et al. present a network-

Table 8.1: Comparison of Event Tracing Systems

Features → **System ↓**	Primary Use case	Application Agnostic	Log Volume	Forensic Validity
Linux Audit [92]	Forensic Analysis	✓	✗	✓
C'Mon [2]	Fault Detection	✓	✗	-
VxWorks [4]	Event Tracing	✓	✗	-
QNX [3]	Event Tracing	✓	✗	-
Zeno [93]	Temporal Provenance	✗	✗	✗
Feather-Trace [36]	Event Tracing	-	✗	-
Ellipsis	**Forensic Analysis**	✓	✓	✓

layer auditing platform that captures the temporal properties of network flows and can thus detect temporal interference [91]. 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. In contrast, our work directly addresses the challenges of RTS auditing by incorporating the audit system into the real-time task schedule.

8.1 COMPARISON OF EVENT LOGGING SYSTEMS FOR RTS

While, to the best of our knowledge, there are no existing systems that address RTS Security Auditing, we present a comparison of related event tracing systems, as summarized in Table 8.1.

Application Agnostic. Linux, *Ellipsis*, C'mon, VxWorks and QNX are fully integrated into their respective systems and trace system events with no application changes. Feather-Trace events can be inserted in syscall path to trace them, though such support does not currently exist. Lack of support for ARM architectures also limits the applications that can utilize Feather-Trace. Zeno is based on Zipkin [94] which relies on application instrumentation.

Log Volume. Only *Ellipsis* features a log reduction mechanism which is critical for contiguous tracing of frequently occurring events, as required for security auditing of long running RTS, while minimizing storage and dissemination costs.

Forensic Validity. Linux Audit supports various security compliance standards [44, 95, 96, 97]. *Ellipsis* modifies auditing source code and thus does not inherit the certifications. However, *Ellipsis* maintains the same auditing information, except timestamps and inter-

process ordering (Refer Appendix B). C'Mon, VxWorks and QNX tracing systems are capable of tracing user-system interface events but the systems are not intended for security auditing. Zeno requires application modifications and hence cannot be used for auditing. Feather-Trace is flexible in what information it can log, however, the feasibility and impact of gathering large amounts of information using Feather-Trace is unclear.

Overhead. The final point of comparison is the overhead added by the tracing system but this is made difficult by limited available information. C'Mon system was built for RTS, having low predictable overheads. On a 2.4 GHz Intel i7-2760QM machine the average per event overhead was evaluated to be 1.3 μs. We could not find any publicly available information about temporal overheads of VxWorks and ONX tracing. Wu *et al.* provide latency impact of Zeno on macrobenchmarks suitable to their usecase. However, a direct inference of per event overhead is not available. Brandenburg and Anderson designed Feather-Trace to have exceptionally low overheads, with only one instruction added to events that are not traced and 61 instructions to added to traced events.

Linux Audit, *Ellipsis* and *Ellipsis-HP* share similar behaviors in per event overheads (§5.5). On our embedded platform (§2.2), syscalls that are not audited incur minimal overheads (avg. 0.8 μs, max. 1.0 μs, Fig. 2.2 filtered scenario). The larger latency to audited syscalls (avg. 27 μs, Fig. 2.2 audit scenario) is due to the rich information captured by Linux Audit that is critical for forensic analysis. For brevity, we now provide short summary of some additional experiments. Analyses of audit information generated by Linux Audit and *Ellipsis* could detect RTS relevant attacks used in prior works in literature [54, 75]. We then modified Linux Audit to record minimal information on each syscall *i.e.,* syscall number and timestamp (For comparison, Appendix B contains complete original Linux Audit entries). These changes reduced the average latency overhead to 12 μs. While the latency reduction is desirable, these changes compromised the ability to detect the attacks. The higher overhead of Linux Audit and *Ellipsis* is a direct result of their security critical functionality and configurability.

CHAPTER 9: CONCLUSIONS

This work presents an analysis of Linux Audit and proves its suitability for use in RTS. *Ellipsis* is a novel log reduction system that exemplifies the synergistic application aware co-design of security and fault analysis mechanisms for RTS. The analysis and the development of *Ellipsis* allows real-time 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 further. Similarly, other security mechanisms from general purpose systems warrant a deeper analysis and redesign for their use in RTS.

REFERENCES

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

[2] J. Song and G. Parmer, "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*. IEEE, 2015, pp. 247–258.

[3] "The instrumented microkernel." [Online]. Available: https://www.qnx.com/developers/docs/6.4.1/neutrino/sys_arch/trace.html

[4] "Tracealyzer for vxworks." [Online]. Available: https://percepio.com/docs/VxWorks/manual/

[5] 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.

[6] I. Lee, O. Sokolsky, S. Chen, J. Hatcliff, 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.

[7] L. Gurgen, O. Gunalp, 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.

[8] L. Monostori, B. Kádár, T. Bauernhansl, 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.

[9] U. Bose, "The black box solution to autonomous liability," *Wash. UL Rev.*, vol. 92, p. 1325, 2014.

[10] K. Böhm, T. Kubjatko, 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.

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

[12] A. Martyn, "Tesla blames drivers who wreck its cars but won't hand over crash data without a court order." [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

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

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

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

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

[17] M. Hahad, "Iot proliferation and widespread 5g: A perfect botnet storm." [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." [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." [Online]. Available: https://www.machinedesign.com/medical-design/article/21128232/step-up-cyber-hygiene-secure-access-to-medical-devices

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

[21] N. Perlroth and D. E. Sanger, "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.

[22] 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. [Online]. Available: http://doi.acm.org/10.1145/945445.945467 pp. 223–236.

[23] 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.

[24] K. Pei, Z. Gu, B. Saltaformaggio, 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. [Online]. Available: http://doi.acm.org/10.1145/2991079.2991122 pp. 583–595.

[25] 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.

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

[27] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, 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. [Online]. Available: http://doi.acm.org/10.1145/3127479.3129249 pp. 405–418.

[28] 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. Yegneswaran, "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.

[29] 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, USA, February 2018.

[30] S. M. Milajerdi, B. Eshete, R. Gjomemo, and V. N. Venkatakrishnan, "Propatrol: Attack investigation via extracted high-level tasks," in *Information Systems Security*, V. Ganapathy, T. Jaeger, and R. Shyamasundar, Eds. Cham: Springer International Publishing, 2018, pp. 107–126.

[31] C. Tan, L. Zhao, W. Liu, L. Xu, and L. Wang, "Capt: Context-aware provenance tracing for attack investigation," *China Communications*, vol. 15, no. 2, pp. 153–169, Feb 2018.

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

[33] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "Holmes: Real-time apt detection through correlation of suspicious information flows," in *2019 2019 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2019. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP.2019.00026

[34] W. U. Hassan, M. Noureddine, 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.

[35] 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.

[36] B. Brandenburg and J. Anderson, "Feather-trace: A lightweight event tracing toolkit," in *Proceedings of the third international workshop on operating systems platforms for embedded real-time applications*, 2007, pp. 19–28.

[37] S. Ma, J. Zhai, Y. Kwon, K. H. Lee, X. Zhang, G. Ciocarlie, A. Gehani, V. Yegneswaran, D. Xu, and S. Jha, "Kernel-supported cost-effective audit logging for causality tracking," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/ma-shiqing pp. 241–254.

[38] National Highway Traffic Safety Administration, "Air bags." [Online]. Available: https://www.nhtsa.gov/equipment/air-bags

[39] K. Kohei, "Recent security features and issues in embedded systems." [Online]. Available: https://elinux.org/images/e/e2/ELC2008_KaiGai.pdf

[40] M. Anderson, "Securing embedded linux." [Online]. Available: https://elinux.org/images/5/54/Manderson4.pdf

[41] R. Day and M. Slonosky, "Securing connected embedded devices using built-in rtos security." [Online]. Available: http://mil-embedded.com/articles/securing-connected-embedded-devices-using-built-in-rtos-security/

[42] "Embedded linux." [Online]. Available: https://elinux.org/Main_Page

[43] R. Rajkumar, L. Sha, and J. P. Lehoczky, "Real-time synchronization protocols for multiprocessors," in *Proceedings. Real-Time Systems Symposium*, 1988, pp. 259–260.

[44] "System auditing." [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security_guide/chap-system_auditing

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

[46] "Raspberry Pi 4 Model B." [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/

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

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

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

[50] R. Paccagnella, P. Datta, W. U. Hassan, A. Bates, C. W. Fletcher, A. Miller, and D. Tian, "Custos: Practical Tamper-Evident Auditing of Operating Systems Using Trusted Execution," in *27th ISOC Network and Distributed System Security Symposium*, ser. NDSS'20, February 2020.

[51] 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. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243763 pp. 1324–1337.

[52] Z. Xu, Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang, "High fidelity data reduction for big data security dependency analyses," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978378 pp. 504–516.

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

[54] M.-K. Yoon, S. Mohan, J. Choi, M. Christodorescu, and L. Sha, "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, pp. 191–196.

[55] "strace." [Online]. Available: https://strace.io/

[56] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001.* IEEE, 2000, pp. 144–155.

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

[58] S. Ma, K. H. Lee, C. H. Kim, J. Rhee, X. Zhang, and D. Xu, "Accurate, low cost and instrumentation-free security audit logging for windows," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC 2015. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2818000.2818039 pp. 401–410.

[59] Y. Sun and M. Di Natale, "Assessing the pessimism of current multicore global fixed-priority schedulability analysis," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, 2018, pp. 575–583.

[60] N. Guan, M. Han, C. Gu, Q. Deng, and W. Yi, "Bounding carry-in interference to improve fixed-priority global multiprocessor scheduling analysis," in *2015 IEEE 21st International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, 2015, pp. 11–20.

[61] M. Schoeberl, "Real-time garbage collection for java," in *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*. IEEE, 2006, pp. 9–pp.

[62] S. Baruah, "Techniques for multiprocessor global schedulability analysis," in *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. IEEE, 2007, pp. 119–128.

[63] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, 2005.

[64] A. D. Team and Community, "Ardupilot," 2020. [Online]. Available: https://ardupilot.org/

[65] R. Paccagnella, K. Liao, D. Tian, and A. Bates, "Logging to the danger zone: Race condition attacks and defenses on system audit frameworks."

[66] 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 *NDSS*, 2018.

[67] A. Bates, D. Tian, K. R. B. Butler, and T. Moyer, "Trustworthy whole-system provenance for the linux kernel," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 319–334.

[68] S. Ma, X. Zhang, and D. Xu, "Protracer: Towards practical provenance tracing by alternating between logging and tainting," in *NDSS*, 2016.

[69] D. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, "Hi-Fi: Collecting High-Fidelity Whole-System Provenance," in *Proceedings of the 2012 Annual Computer Security Applications Conference*, ser. ACSAC '12, Orlando, FL, USA, 2012.

[70] "Motion." [Online]. Available: https://motion-project.github.io/

[71] C.-Y. Chen, A. Ghassami, S. Nagy, M.-K. Yoon, S. Mohan, N. Kiyavash, R. B. Bobba, and R. Pellizzoni, "Schedule-based side-channel attack in fixed-priority real-time systems," Tech. Rep., 2015.

[72] "audisp-remote(8) - linux man page." [Online]. Available: https://linux.die.net/man/8/audisp-remote

[73] "Central audit logging: Configuration and collecting linux audit events," 2019. [Online]. Available: https://linux-audit.com/central-audit-logging-configuration-collecting-linux-audit-events/

[74] W. Jiang, L. Wen, J. Zhan, and K. Jiang, "Design optimization of confidentiality-critical cyber physical systems with fault detection," *Journal of Systems Architecture*, vol. 107, p. 101739, 2020.

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

[76] T. F. J. . Pasquier, J. Singh, D. Eyers, and J. Bacon, "Camflow: Managed data-sharing for cloud services," *IEEE Transactions on Cloud Computing*, vol. 5, no. 3, pp. 472–484, July 2017.

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

[78] F. Capobianco, R. George, K. Huang, T. Jaeger, S. Krishnamurthy, Z. Qian, M. Payer, and P. Yu, "Employing Attack Graphs for Intrusion Detection," in *New Security Paradigms Workshop*, ser. NSPW'19, September 2019.

[79] P. Gao, X. Xiao, D. Li, Z. Li, K. Jee, Z. Wu, C. H. Kim, S. R. Kulkarni, and P. Mittal, "SAQL: A stream-based query system for real-time abnormal system behavior detection," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/gao-peng pp. 639–656.

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

[81] X. Han, T. Pasqueir, A. Bates, J. Mickens, and M. Seltzer, "Unicorn: Runtime Provenance-Based Detector for Advanced Persistent Threats," in *27th ISOC Network and Distributed System Security Symposium*, ser. NDSS'20, February 2020.

[82] Y. Kwon, D. Kim, W. N. Sumner, K. Kim, B. Saltaformaggio, X. Zhang, and D. Xu, "Ldx: Causality inference by lightweight dual execution," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872395 pp. 503–515.

[83] T. Pasquier, X. Han, T. Moyer, A. Bates, O. Hermant, D. Eyers, J. Bacon, , and M. Seltzer, "Runtime Analysis of Whole-System Provenance," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. ACM, Oct 2018.

[84] 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.

[85] A. Bates, K. R. B. Butler, and T. Moyer, "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*, ser. TaPP'15, July 2015.

[86] A. Bates, D. Tian, G. Hernandez, T. Moyer, K. R. Butler, and T. Jaeger, "Taming the Costs of Trustworthy Provenance through Policy Reduction," *ACM Trans. on Internet Technology*, vol. 17, no. 4, pp. 34:1–34:21, sep 2017.

[87] 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. [Online]. Available: http://dl.acm.org/citation.cfm?id=3277203.3277331 pp. 1723–1740.

[88] K. H. Lee, X. Zhang, and D. Xu, "LogGC: Garbage Collecting Audit Log," in *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516731 pp. 1005–1016.

[89] 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.

[90] D. J. Tian, A. Bates, K. R. B. Butler, and R. Rangaswami, "Provusb: 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.

[91] 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. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/wu pp. 395–420.

[92] S. Grubb, "Linux audit." [Online]. Available: https://people.redhat.com/sgrubb/audit/

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

[94] "Zipkin." [Online]. Available: https://zipkin.io/

[95] N. S. Agency, "Information systems security organization." [Online]. Available: https://www.commoncriteriaportal.org/files/ppfiles/capp.pdf

[96] N. S. Agency, "Labeled security protection profile." [Online]. Available: https://www.commoncriteriaportal.org/files/ppfiles/lspp.pdf

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

APPENDIX A: TEMPLATES FOR ARDUPILOT

Thread/Task	arducopter	ap-rcin	ap-spi-0
Syscall Count	14	16	1
Expected runtime (ns)	1303419	671567	0
Expected inter-arrival time (ns)	5012313	20029121	2010477
Syscall List	4:3:-1:1:-1 4:4:-1:1:-1 4:5:-1:1:-1 4:6:-1:1:-1 4:7:-1:1:-1 4:8:-1:1:-1 4:9:-1:1:-1 4:10:-1:1:-1 4:11:-1:1:-1 4:12:-1:1:-1 4:13:-1:1:-1 4:14:-1:1:-1 4:15:-1:1:-1 4:16:-1:1:-1	180:17:-1:11:-1 180:18:-1:11:-1 180:19:-1:11:-1 180:20:-1:11:-1 180:21:-1:11:-1 180:22:-1:11:-1 180:23:-1:11:-1 180:24:-1:11:-1 180:25:-1:11:-1 180:26:-1:11:-1 180:27:-1:11:-1 180:28:-1:11:-1 180:29:-1:11:-1 180:30:-1:11:-1 180:31:-1:11:-1 180:32:-1:11:-1	3:55:-1:8:-1

Table A.1: Columns of this table describe the content of three template files for ArduCopter.

We identified three templates in ArduPilot using the template generation process defined earlier (§3.2). System call numbers and their corresponding arguments, a0 - a4, were extracted from the audit logs. `read`, `write`, `pread64` have system call 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 A.1 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

```
type=SYSCALL msg=audit(1601405431.612391356:5893330): arch=40000028
    per=800000 template=arducopter rep=10 stime=1601405431589320747
    etime=1601405431612287042 ppid=1208 pid=1261 tid=1261 auid=1000 uid
    =0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=pts0 ses=3
    comm="arducopter" exe="/home/pi/ardupilot/build/navio2/bin/
    arducopter" key=(null)
```

APPENDIX B: 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(1601405431.612391356:5893330): arch=40000028
    syscall=4 per=800000 success=yes exit=8 a0=3 a1=126aa4 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)
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)
type=SYSCALL msg=audit(1601405431.612391367:5893334): arch=40000028
    syscall=4 per=800000 success=yes exit=7 a0=5 a1=126ab8 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)
```

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(1601405431.612391370:5893335): arch=40000028  per
    =800000 template=arducopter rep=1 stime=1601405431612391356 etime
    =1601405431612391367 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
    )
```

Using the template and the compressed line of log, following three lines can be reconstructed. ∅ 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:∅): arch=40000028 syscall=4 per
    =800000 success=yes exit=8 a0=3 a1=∅4 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)
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)
type=SYSCALL msg=audit(1601405431.612391367:∅): arch=40000028 syscall=4 per
    =800000 success=yes exit=7 a0=5 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)
```

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.