System Auditing for Real-Time Systems *

AYOOSH BANSAL, University of Illinois Urbana-Champaign, USA
ANANT KANDIKUPPA, University of Illinois Urbana-Champaign, USA
MONOWAR HASAN, Washington State University, USA
CHIEN-YING CHEN, University of Illinois Urbana-Champaign, USA
ADAM BATES, University of Illinois Urbana-Champaign, USA
SIBIN MOHAN, The George Washington University, USA

System auditing is an essential tool for detecting malicious events and conducting forensic analysis. Although used extensively on general-purpose systems, auditing frameworks have not been designed with consideration for the unique constraints and properties of Real-Time Systems (RTS). System auditing could provide tremendous benefits for security-critical RTS. However, a naïve deployment of auditing on RTS could violate the temporal requirements of the system while also rendering auditing incomplete and ineffectual. To ensure effective auditing that meets the computational needs of recording complete audit information while adhering to the temporal requirements of the RTS, it is essential to carefully integrate auditing into the real-time (RT) schedule.

This work adapts the Linux Audit framework for use in RT Linux by leveraging the common properties of such systems, such as special purpose and predictability. *Ellipsis*, an efficient system for auditing RTS is devised that learns the expected benign behaviors of the system and generates succinct descriptions of the expected activity. Evaluations using varied RT applications show that *Ellipsis* reduces the volume of audit records generated during benign activity by up to 97.55%, while recording detailed logs for suspicious activities. Empirical analyses establish that the auditing infrastructure adheres to the properties of predictability and isolation that are important to RTS. Furthermore, the schedulability of RT task sets under audit is comprehensively analyzed to enable the safe integration of auditing in RT task schedules.

CCS Concepts: • **Computer systems organization → Real-time systems**; • **Security and privacy → Systems security**.

Additional Key Words and Phrases: security auditing, model-based reduction, cyber-physical systems

*_Ellipsis_ was first introduced in a paper published in *ESORICS 2022* [15]. In this work we expand the capability of *Ellipsis* to efficiently audit different classes of RT Applications and the automated template generation (Section 3.4, Algorithm 1 and Figure 2). This is evaluated by expanding the experiments using the firm deadline RT application *Ardupilot* (Sections 5.3 and 5.5) and with new evaluations using a soft deadline video analysis application called *Motion* (Sections 4.4 and 6). With comprehensive empirical analysis, we establish the auditing framework's adherence to properties that are important to RTS (Section 7). Finally, we perform a thorough schedulability analysis for RT applications under audit (Section 8). Other major modifications/enhancements are: *(a)* we add a detailed discussion for RTS properties that *Ellipsis* depends on as Sections 2.2.1 and 2.2.2; *(b)* we provide additional details about *Ellipsis*' functionality in Section 3 and new Tables 2 and 4, summarizing notations and evaluation questions, respectively, for the reader's convenience and clarity; *(c)* we expand the discussion about *Ellipsis* in Sections 9.2, 9.5, and 9.6; *(d)* we expand the related work, Section 10, to include state of the art in System Auditing, RTS Security and Data Compression and address recent literature; *(e)* we provide detailed examples of templates in Appendix A; *(f)* other editorial changes to most sections, especially the abstract, introduction, and conclusion.

Authors' addresses: Ayoosh Bansal, University of Illinois Urbana-Champaign, Urbana-Champaign, USA, ayooshb2@illinois.edu; Anant Kandikuppa, University of Illinois Urbana-Champaign, Urbana-Champaign, USA, anantk3@illinois.edu; Monowar Hasan, Washington State University, Pullman, USA, monowar.hasan@wsu.edu; Chien-Ying Chen, University of Illinois Urbana-Champaign, Urbana-Champaign, USA, cchen140@illinois.edu; Adam Bates, University of Illinois Urbana-Champaign, Urbana-Champaign, USA, batesa@illinois.edu; Sibin Mohan, The George Washington University, Washington, D.C., USA, sibin.mohan@gwu.edu.

Manuscript submitted to ACM

ACM Reference Format:
Ayoosh Bansal, Anant Kandikuppa, Monowar Hasan, Chien-Ying Chen, Adam Bates, and Sibin Mohan. 2023. System Auditing for Real-Time Systems . *ACM Trans. Priv. Sec.* 37, 4, Article 111 (August 2023), 35 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Real-Time Systems (RTS) are an integral part of numerous safety and security-critical domains, including medical devices, autonomous vehicles, manufacturing automation and smart cities, among others [47, 66, 83, 93]. Attacks on these systems can lead to subversion of life-saving medical devices [106], vehicle hijacks [38, 55], manufacturing disruptions [98] and even IoT botnets [49]. Therefore, RTS have become attractive targets for attacks [41]. The correctness of an RTS depends on the computations being completed within a temporal constraint, which is referred to as a deadline.

Security auditing is a key component of intrusion detection and response in general-purpose systems. Auditing plays a crucial role in detecting, investigating, and provenance analysis of intrusions [28, 52, 53, 81]. System auditing takes place at the *kernel layer* and creates a new event for every syscall that is issued. Not only does this approach relieve the application developer from the responsibility of event logging, it provides a unified view of system activity in a way that application-specific logging simply cannot. In particular, systems logs can be parsed into a connected graph based on the shared dependencies of individual events, facilitating causal analysis over the history of events within a system [18, 58, 62, 65, 65, 72, 74, 78, 87, 90]. This capability is invaluable to defenders when tracing suspicious activities [52, 53, 81], to the point that the vast majority of cyber analysts consider audit logs to be the most important resource when investigating threats [28]. Hence, the deployment of system-level audit capabilities can help on multiple fronts: *(a)* fault detection/diagnosis and *(b)* understanding and detecting future security events.

However, RTS event logging today is limited to application layer event recorders [23, 25] or performance profiling [26]. The information recorded, *e.g.,* syscall occurrence not arguments, is insufficient to trace attacks. Without syscall argument values links between entities cannot be identified, *e.g.,* without a filename or file handle information accesses to a file cannot be linked together. There is a growing need for comprehensive system auditing in RTS [12, 39, 63]. However, auditing systems like Linux Audit are known to cause high overheads [76]. Therefore, a naïve inclusion of auditing in Real-Time (RT) application schedules can lead to violations of the temporal requirements of the applications.

To bring comprehensive system auditing to RTS, we present **Ellipsis**, a kernel-based auditing system that leverages unique properties of RTS to enable efficient system auditing in RTS. Adapting Linux Audit [101] for use in RTS, *Ellipsis* leverages predictable repeating execution paths, a property common to the vast majority of RTS. As part of the extensive pre-deployment analyses that are part of RTS development, using an automated iterative process *Ellipsis* learns these execution paths, generating *templates* representing the learned behaviors to reduce the audit event streams by matching these templates to succinct descriptions. Simultaneously, any anomalous behavior is recorded in full detail. *Ellipsis* minimizes the volume of audit events and logs generated without losing security-relevant information.

Reducing expected event streams in the kernel, before they are ever recorded, allows *Ellipsis* to minimize the auditing resource requirements for benign behaviors of RT applications, thus maximizing the auditing capacity available for anomalous activities. Despite the reduction of event streams, *Ellipsis* retains all relevant information, detecting stealthy attacks designed to work specifically against RTS. *Ellipsis* achieves high (> 90%) event generation reduction for different classes of RT applications, from soft deadline video processing applications to firm deadline autopilot systems. Furthermore, *Ellipsis* adheres to the properties of execution time predictability and does not introduce any significant blocking and therefore priority inversion [94], *i.e., Ellipsis* can be safely integrated within RT schedules. Encouraged by these findings, we conduct a detailed schedulability analysis for auditing RT tasks with *Ellipsis. Ellipsis* presents the

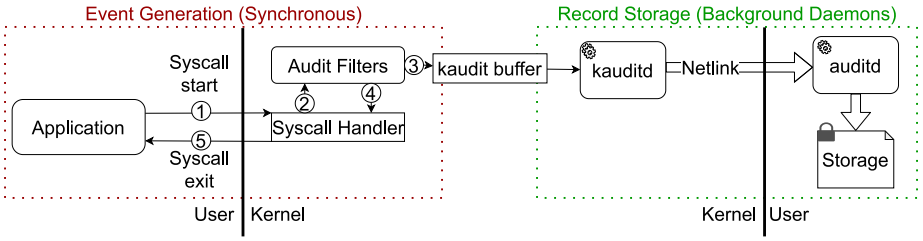


Fig. 1. Linux Audit Framework [1]. Audit logs are generated using auditing hooks in the kernel's syscall handler and temporarily stored in the `kauditd` buffer. Log maintenance is handled by two background daemons, `kauditd` and `auditd`.

first system for security auditing tailored specifically for RTS. With the results and analysis presented in this work, RTS can appropriately provision auditing tasks, including them as part of the RT schedules, enabling safe and effectual use of system-wide security auditing in RTS. The key **contributions** of this work are:

- *Ellipsis*[1], an audit framework, uniquely-tailored to RT environments (§3).
- Security analysis (§4) and performance evaluations (§5, §6) to demonstrate that *Ellipsis* retains relevant information while significantly reducing audit event generation and log volume.
- Temporal Properties (§7) and Schedulability (§8) analyses to establish the safe inclusion of auditing in RTS.

2 BACKGROUND AND SYSTEM MODEL

2.1 Linux Audit Framework

The Linux Audit system [101] provides a way to record system activities. As illustrated in Figure 1, on any syscall invocation ①, hook code invokes Linux Audit. If the event matches the `audit_filter` ②, as defined by the system administrator, a new event record is added to a `kaudit_buffer` ③. The control flow then returns back to syscall handler ④ and eventually to the application ⑤. The records in `kaudit_buffer` must be transferred to an eventual user, most commonly to the file system for persistent storage. Background daemons `kauditd` and `auditd`, running in kernel and user spaces respectively, transmit these records from the `kaudit_buffer` to the userspace for storage.

It is well-established that Linux Audit can incur large computational and storage overheads in traditional software [76]. However, its impacts on RT applications were unclear. Both components of auditing present unique challenges:

- *Event generation* hooks in the syscall path not only add additional latency to each syscall but also introduces the shared `kauditd` buffer whose access is coordinated using a spinlock. These changes could potentially wreak havoc on RT task sets as a result of changing execution profiles, resource contention, or priority inversion [94]. Encouragingly, upon conducting a detailed analysis (§7, §8) we observed that Linux Audit does not introduce significant issues of priority inversion or contention over auditing resources shared across applications. Further, except for limited outlier cases, the latency introduced by auditing syscalls can be bounded and schedulability determined. Hence it is a good candidate for firm and soft deadline RTS as supported by RT Linux [103].
- *Record storage* completeness depends on the background daemons being provided sufficient execution time, otherwise `kaudit_buffer` becomes full and any syscalls executed when the buffer is full are lost, compromising the integrity and utility of the audit log. This problem is the main focus of *Ellipsis*.

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

Table 1. RTS properties relevant to *Ellipsis*

Property	Relevance to *Ellipsis*	Sections
Periodic tasks	Most RT tasks are periodically activated, leading to repeating behaviors. *Ellipsis* templates describe the most common repetitions.	2.2.1, 3
Aperiodic tasks	The second most common form of RT tasks, Aperiodic tasks also lead to repeating behaviors, but with irregular inter-arrival times.	2.2.1, 3
Code Coverage	High code coverage analyses are part of existing RTS development processes, *Ellipsis*' automated template generation adds minimal cost.	2.2.2, 9.5
Special Purpose	RTS are special purpose machines, tasks are known at development *i.e.,* templates can be created before system deployment.	3
Temporal Predictability	A requirement for safety and correct functioning of RTS, naïvely enabling auditing can violate this by introducing overheads and variability.	5.5, 9.3
Longevity	Once deployed RTS can remain functional for years. *Ellipsis*' can save enormous log storage and transmission costs over the lifetime of the RTS.	5.4, 6

2.2 RTS Properties

Ellipsis leverages properties unique to RT environments, that we describe here. In contrast to traditional applications where determining all possible execution paths is often undecidable, knowledge about execution paths is an essential component of RT application development. RTS are special-purpose machines that execute well-formed tasksets to fulfill predetermined tasks. RT Applications structure commonly involves repeating loops that are excellent targets for conversion to templates. Various techniques are employed to analyze the tasksets with high code coverage *e.g.,* worst-case execution time (WCET) analysis for real-time tasks [27, 48, 54, 71, 92, 96, 120]. All expected behaviors of the system must be accounted for at design time in conjunction with the system designers. Any deviation is an unforeseen fault or malicious activity, which needs to be audited in full detail. Table 1 contains a summary of RTS features and constraints, with references to sections in this work that discuss, evaluate, or leverage these features. In this work we show that applications from two different classes of RTS follow this model; a control application (Ardupilot [14]) and a video analysis application (Motion [5]). We now discuss in further detail two RTS features that *Ellipsis* leverages.

2.2.1 Repetition of Sequences. In their seminal work on Intrusion Detection, Hofmeyr *et al.* [57] established that the normal behavior of an application can be profiled as sequences of syscall. This works exceptionally well for RTS as they feature limited tasks with limited execution paths on a system. Unlike general-purpose systems, RTS run limited predefined tasks. The requirements of reliability, safety, and timing predictability imply that RTS have limited execution paths which can be tested and analyzed to ensure the previously mentioned requirements. A recently published survey of industry practitioners in RTS [9] shows that 82% of the RTS contained tasks with periodic activation. Periodically activated tasks with limited execution paths will invariably lead to high repetitions of certain sequences. Yoon *et al.* [120] demonstrated the existence of repeating syscall sequences in an RTS. The reliable repetition of behaviors has also led to profile-driven techniques being successfully employed towards achieving predictable temporal behaviors in RTS [45].

2.2.2 Code Coverage. The survey [9] also noted that the five most important system aspects for industrial RTS were Functional Correctness, Reliability and Availability, System Safety, Timing predictability, and System security. The role of code coverage in software testing is well established [60, 113]. Prior works have established the correlation between code coverage and reliability of software [34, 35, 40]. Software safety standards include structural code coverage as a requirement [8, 24]. Timing predictability in RTS is ensured by coding standards, guidelines [54, 92, 105] and

worst-case execution time (WCET) analysis [48, 96]. Therefore, high code coverage is an integral component of the RTS development process. Template generation for *Ellipsis* therefore does not introduce a significant additional burden. Template sequences are determinable in the course of existing development processes for RTS.

2.3 Threat Model

We consider an adversary that aims to penetrate and impact an RTS through exfiltration of data, corruption of actuation outputs, causing deadline violations, *etc.* This attacker may install modified programs, exploit a running process, or install malware on the RTS to achieve their objectives. To observe this attacker, our system adopts an aggressive audit configuration intended to capture all forensically-relevant events, as identified in prior works [44, 52, 72, 77, 85, 102, 116].[2] We assume that the underlying OS and the audit subsystem therein are trusted. This is a standard assumption in the system auditing literature [18, 52, 73, 79, 91]. Far from being impractical on RTS, prior works provide a secure kernel that meets both the trust and temporal requirements for hosting *Ellipsis* in RT Linux [31, 37, 108, 109].

Ellipsis' goal is to capture evidence of an attacker's intrusion and activity without losing relevant information and hand it off to a tamper-proof system. Although audit log integrity is an important security goal itself, it is commonly explored orthogonally to other audit research due to the modularity of security solutions, *e.g.,* [18, 85, 117]. Therefore, we assume that once recorded to `kaudit_buffer`, attackers cannot compromise the integrity of audit logs. Finally, we assume that the real-time applications can be profiled in a controlled benign environment prior to being the target of attack, such as pre-deployment testing and verification as in prior work [121].

3 *ELLIPSIS*

The volume of audit events is the major limiting factor for auditing RTS. High event volume can result in event loss, high log storage costs, and large maintenance overheads [76]. We present *Ellipsis*, an audit event reduction technique designed specifically for RTS. *Ellipsis* achieves this through *templatization* of the audit event stream. Templates represent learned expected behaviors of RT tasks, described as a sequence of syscalls with arguments and temporal profiles.[3] These templates are generated in an offline profiling phase, similar to common RTS analyses like WCET [27, 69, 121]. At runtime, the application's syscall stream is compared against its templates; if a contiguous sequence of syscalls matches a template, only a single record indicating the template match is inserted into the event stream (`kaudit_buffer`). Matched syscalls are never inserted into the event stream, reducing the number of events generated by the auditing system. While a sequence of audited syscall events is replaced by a single record, the relevant information is preserved (§4).

3.1 Model

Consider a system in which the machine operator wishes to audit a single RT task $\boldsymbol{\tau}$. An RT *task* here corresponds to a *thread* in Linux systems, identified by a combination of process and thread ids. We can limit this discussion to a single task, without losing generality, as *Ellipsis*' template creation, activation, and runtime matching is independent for each task. We modified Linux Audit to include thread ids in audit event records to support this independent handling.

RT tasks are commonly structured with a one-time *init* component and repeating *loops*. Let $\boldsymbol{s_i}$ denote a syscall sequence the task exhibits in a *loop* execution and $\boldsymbol{N}$ the count of different syscall execution paths τ might take (*i.e.,* $0 < i \leq N$). A

[2] 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`.

[3] Template examples are available as Appendix A

Table 2. Notations

Symbol	Description
S1, S2, S3	System Calls
$iTPL$	Intermediate Template, *i.e.,* a template without temporal information
$TPL - X$	Template-X where $X \in \mathbb{N}$
Δ_i	Observed runtime for ith instance of the templatized task
T_X	Temporal constraint of $TPL - X$
x,{y}	Unique state ID for *Ellipsis* FSA x: number of system calls matched; y: set of potential template matches.
τ	A RT task
s_i	ith syscall sequence exhibited by τ
N	Count of possible s_i for τ, $0 < i \leq N$
$len(s_i)$	number of syscalls in s_i
p_i	probability of occurrence of corresponding s_i

template describes these sequences (s_i), identifying the syscalls and arguments. As noted in Section 2.2, RT applications are developed to have limited code paths and bounded loop iterations. Extensive analysis of execution paths is a standard part of the RTS development process. Thus, for RTS, N is finite and determinable. Let function $\boldsymbol{len(s_i)}$ return the number of syscalls in the sequence s_i. Further, let $\boldsymbol{p_i}$ be the probability that an iteration of τ exhibits syscall sequence s_i.

3.2 Template Learning Phase

Templates are created during a *predeployment learning phase.* Identification of cyclic syscall behaviors has been addressed in prior works [67, 77], using binary analysis, code annotations, stack analysis, or a combination. While any technique that yields s_i and p_i can be employed here, including the prior mentioned ones, we developed an automated dynamic analysis, leveraging RT task structure and Linux Audit itself. Applications are executed and audited in a benign environment, as in prior work [120]. Stress tests and code coverage suits are ideal for this, to generate audit logs from all recurrent code paths. These audit logs contain the syscall sequence that the application exhibits. The learning phase duration is primarily dependent on the application's test suite's execution time and the number of times individual tests may need to be run (§3.4). Subsequent sections describe the template creation process.

3.3 Sequence Identification

Given the audit logs from the application test suite execution, we can identify syscalls sequences and their probability of occurrence. We observe that RT tasks typically end with calls to `sleep` or `yield` that translate to `nanosleep` and `sched_yield` syscalls in Linux. Periodic behaviors can also be triggered by polling `timerfds` to read events from multiple timers by using `select` and `epoll_wait` syscalls. We leverage these syscalls to identify boundaries of task executions within the audit log and then extract sequences of syscall invocations. Figure 2 provides an overview of this process. We also modified Linux Audit to include the Thread ID in log messages to disambiguate threads belonging to a process. This first step yields the per-task syscall sequences exhibited by the application and their properties: length, probability of occurrence, and arguments. These syscall sequences are then converted into intermediate templates, each entry of which includes the syscall name along with the arguments. Intermediate templates are identical to final

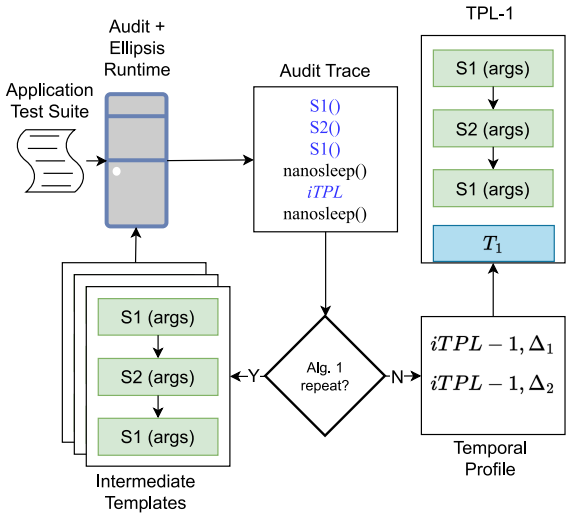


Fig. 2. *Ellipsis* template creation. Syscalls are denoted by S1/S2. Application is audited iteratively with *Ellipsis* to identify repeating syscall sequences, with each iteration using the intermediate templates identified previously. Final iterations yield no new templates, but rather yield the task's execution time profile (Δ_i), used to generate the temporal constraint (T_1). Intermediate templates enriched with temporal constraints are the final templates.

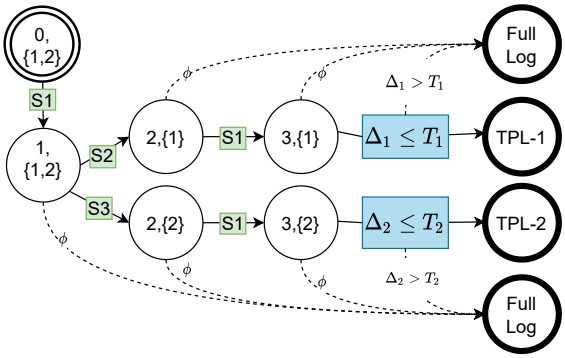


Fig. 3. Runtime template matching as an FSA with states as [syscalls matched count, {set of reachable templates}]. Syscall invocations trigger state transitions. TPL-1 (S1, S2, S1) and TPL-2 (S1, S3, S1) are shown as example. Template matches (TPL-1, TPL-2) emit a single record, failure leads to a full log store. Completing the template sequence and satisfying temporal constraints leads to an accept state (TPL-1, TPL-2) emitting a single record. Any divergence or failure causes *Ellipsis* to emit complete logs, shown as dotted transitions. The FSA then returns to the initial state [0, {1,2}] for the task to start capturing the next iteration.

templates, except that they do not have any temporal limits. Since audit events may be lost during this tracing, the sequences identified at this stage may be incomplete. Therefore an iterative procedure is required to ensure that all sequences exhibited by the application at runtime are identified by *Ellipsis*.

3.4 Iterative Procedure

Event loss may occur during the learning phase, *i.e.,* before *Ellipsis* can be used to reduce the audit event volume. Templates captured from such an audit log will be incomplete. An iterative process is required to capture complete syscall sequences, as described by Algorithm 1. In each iteration, all previously identified intermediate templates are loaded to memory (§3.7). *Ellipsis* uses these intermediate templates to reduce audit events (§3.8). Intermediate templates do not have temporal information and therefore temporal constraint is assumed to be infinity. Parts of the audit trace will be formed of records reduced by *Ellipsis*. This frees up auditing capacity (§5.3), leading to a reduction in the number of audit records lost. In each iteration, therefore, additional intermediate templates (sequences) are identified, further freeing up auditing capacity for the next iteration. This procedure terminates successfully when no audit events are lost, but unsuccessfully when audit events are lost but no new sequences could be identified. At the end of this process, we have a set of intermediate templates which correspond to all identified sequences.

3.5 Sequence Selection

A subset of intermediate templates is chosen to be converted to final templates. This choice is based on the trade-off between the benefit of audit event volume reduction and the memory cost as defined later by (3) and (7), respectively.

Algorithm 1 Iterative Sequences Identification

```
1:  iTPL ← {}                                          ▷ Initialize intermediate template set to empty
2:  repeat
3:      Clear templates from memory, audit stats, and audit logs   ▷ using auditctl and file truncation
4:      Load current iTPL to memory                                ▷ Described in Section 3.7
5:      Run application test suite, while being audited with Ellipsis.
6:      Wait till test suite finishes and kaudit_buffer is empty   ▷ using auditctl
7:      lost_cur ← count of audit events lost                      ▷ using auditctl
8:      iTPL_cur ← new sequences from audit trace   ▷ Ellipsis reduces all previously identified sequences
9:      iTPL ← {iTPL, iTPL_cur}
10:     if lost_cur == 0 then
11:         result ← done                          ▷ No loss so further iterations will not yield new patterns
12:     else if lost_cur == lost_prev & iTPL_cur == {} then
13:         result ← fail                          ▷ Stuck with log loss and no new iTPL to reduce loss further
14:     else
15:         lost_prev ← lost_cur
16:     end if
17: until result == done||result == fail
```

The security tradeoff is minimal, as demonstrated in Section 4. Let's assume $\boldsymbol{n}$ sequences are chosen to be reduced ($0 \leq n \leq N$). As noted earlier, *Ellipsis* treats each task independently, the value of n is also independent for each task.

3.6 Template Creation

For the next step, Figure 2 Temporal Profile, these n templates are loaded and the application profiled again to collect temporal profile for each template *i.e.,* the expected duration and inter-arrival intervals for each template. The intermediate templates are enriched with temporal information, yielding final templates. Templates are stored in the form of text files and occupy negligible disk space, *e.g.,* templates used for evaluation (§4.3.1) occupied 494 bytes. This whole process is automated, given an application binary with necessary inputs, using the template creation toolset.[1]

3.7 *Ellipsis* Activation

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

3.8 Runtime Matching

Given the template(s) of syscall sequences, an *Ellipsis* kernel module, extending from Linux Audit syscall hooks, filters syscalls that match a template. The templates are modeled as a finite state automaton (FSA), implemented as a collection of linked lists in kernel memory. While the RT task is executing, all syscall sequences allowed by the automaton are stored in a temporary task-specific buffer. If the set of events fully describes an automaton template, *Ellipsis* discards the contents of the task-specific buffer and enqueues a single record onto the `kaudit` buffer to denote the execution of a templatized activity. Alternatively, *Ellipsis* enqueues the entire task-specific buffer to the main `kaudit_buffer` if

Table 3. Parameters from evaluation (§4.3.1)

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

(a) a syscall occurs that is not allowed by the automaton, *(b)* the template is not fully described at the end of the task instance or *(c)* the task instance does not adhere to the expected temporal behavior of the fully described template. Thus, the behavior of each task instance is reduced to a single record when the task behaves as expected. For any abnormal behavior, the complete audit log is retained. Figure 3 demonstrates this procedure with simplified examples.

3.9 Audit Event Reduction

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

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

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

Ellipsis' Event reduction — Audit events for n sequences

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

Iterations — *Ellipsis* events for n sequences

As evident from (3), to maximize reduction, long sequences with large p_i values must be chosen as the n sequences for reduction. RT applications, like control systems, autonomous systems, and even video streaming, feature limited execution paths for the majority of their runtimes [64]. This property has been utilized by Yoon *et al.* in a prior work [120]. Therefore, for RT applications the distribution of p_i is highly biased *i.e.,* certain sequences s_i have a high probability of occurrence. Table 3 provides example values for the parameters used, determined during the *Sequence Identification* step in template creation for the later case study (§5).

3.10 Storage Size Reduction

Let $\boldsymbol{B_A}$ denote the average cost of representing a syscall event in the audit log and $\boldsymbol{B_E}$ denote the average cost of representing *Ellipsis*' template match record. Thus B_A represents the average size over all events in the Linux Audit log, whereas in *Ellipsis* syscall sequences that match a template will be removed and replaced with a template match event of an average size B_E. By design, $B_E <= B_A$; B_E is a constant 343 bytes, while B_A averaged 527 bytes (1220 bytes max) in our evaluation. Noting that the init events (f) are not reduced by *Ellipsis*, the disk size reduction *i.e.,* difference in sizes of $\boldsymbol{\tau}$'s audit log for Linux Audit $\boldsymbol{(L_A)}$ and *Ellipsis* $\boldsymbol{(L_E)}$ is:

$$L_A = I * (B_A * \sum_{i=1}^{N}(p_i * len(s_i))) + f * B_A \tag{4}$$

$$L_E = I * (B_E * \sum_{i=1}^{n} p_i + B_A * \sum_{i=n+1}^{N}(p_i * len(s_i))) + f * B_A \tag{5}$$

The reduction in log size is given by:

$$L_A - L_E = I * \left(B_A * \sum_{i=1}^{n} (p_i * len(s_i)) - B_E * \sum_{i=1}^{n} p_i \right) \quad (6)$$

Log size reduction; Audit log size for n sequences; Iterations; *Ellipsis* log size for n sequences

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

3.11 Memory Tradeoff

The tradeoff for *Ellipsis*' benefits are computational overheads (evaluated in §5.6 and §7.6) and the memory cost of storing templates ($\boldsymbol{M_\tau}$). Let $\boldsymbol{M_{fixed}}$ be the memory required per template, excluding syscalls, while $\boldsymbol{M_{syscall}}$ be the memory required for each syscall in the template. On 32 bit kernel $M_{fixed} = 116$ and $M_{syscall} = 56$ bytes, determined by *sizeof* data structures. As an example, 3 templates from evaluation occupied 2 KB in memory (Appendix A)

$$M_\tau = M_{fixed} * n + M_{syscall} * \sum_{i=1}^{n} len(s_i) \quad (7)$$

For reference, the parameters for the application detailed in Section 4.3.1 are provided in Table 3. Complete templates for the same can be found in Appendix A. The 3 templates used for the case study took 2 KB of memory space. It should be noted that (7) does not consider the potential to reduce the `kaudit_buffer` size, decreasing the memory space required for auditing. Section 5.3 evaluates the reduction in `kaudit_buffer` occupancy when using *Ellipsis*.

3.12 Extended Reduction Horizon

Until now we have limited the horizon of reduction to individual task *loop* instances. We can further optimize by creating a single record that describes multiple consecutive matches of a template. This higher performance system is henceforth referred to as **Ellipsis-HP**. When a *Ellipsis-HP* match fails, a separate record is logged for each of the base template matches along with a complete log sequence for the current instance (*i.e.,* the base behavior of *Ellipsis*). *Ellipsis-HP* performs best when identical sequences occur continuously, capturing all sequence repetitions in one entry. By design, event reduction for *Ellipsis-HP* is always better than or equal to *Ellipsis*. The memory and computational costs of using *Ellipsis-HP* over *Ellipsis* are negligible. The potential drawback of *Ellipsis-HP* is in the increased uncertainty in the timing for individual events, if the complete audit log needs to be reconstructed, as described in Section 4.2.

$$E^{Best}_{Ellipsis\text{-}HP} = n + I * \sum_{i=n+1}^{N} (p_i * len(s_i)) + f \quad (8)$$

$$E_A - E^{Best}_{Ellipsis\text{-}HP} = I * \sum_{i=1}^{n} (p_i * len(s_i)) - n \quad (9)$$

Log size reduction; Audit log size for n sequences; Iterations; $E^{Best}_{Ellipsis\text{-}HP}$ log size for n sequences

Table 4. Evaluation and Analyses questions

Sections	Question
4.1, 4.2	Does *Ellipsis* create new vulnerabilities allowing attackers to evade detection?
4.3, 4.4	Is *Ellipsis* and Auditing in general capable of detecting attacks on RTS?
5, 6	How effective is *Ellipsis* in reducing audit event generation for real-world RT applications?
7	Do Linux Audit and *Ellipsis* adhere to the temporal properties important to RTS?
8	Can the schedulability of RT tasks under audit be determined? What changes must be considered?

3.13 Temporal Constraints

RTS are sensitive to time intervals between events, thus, *Ellipsis* also considers temporal checks in the template matching process (§3.6 and §3.8). *Ellipsis-HP* adds additional checks for inter-arrival times of different task instances. Note that the prior discussion on log sizes ((3), (6) and (9)), assumes that temporal constraints are always met. The impact of temporal constraints on log size is evaluated in Section 5.5.

3.14 Summary

Ellipsis leverages predictable benign behaviors and analyses of RT applications to limit the audit event volume generation. We now evaluate and analyze *Ellipsis*, compared to Linux Audit, to answer the questions summarised in Table 4.

4 SECURITY ANALYSIS

The security goal of *Ellipsis*, indeed auditing in general, is to record all forensically relevant information, thereby aiding in the investigation of suspicious activities. We now discuss the security implications of *Ellipsis*, demonstrating that *Ellipsis* fulfills the same security role as Linux Audit, even improving upon it by reducing benign information, simplifying the forensic analysis, and avoiding loss of records of malicious events.

4.1 Stealthy Evasion

If a malicious process adheres to the expected behavior of benign tasks, the associated logs will be reduced. The question, then, is whether a malicious process can perform meaningful actions while adhering to the benign templates. If *Ellipsis* exclusively matched against syscall IDs only, such a feat may be possible; however, *Ellipsis* also validates syscalls' arguments and temporal constraints, effectively validating both the *control flow* and *data flow* before templatization, to the extent possible using syscalls. Thus making it exceedingly difficult for a process to match a template while affecting the RTS in any meaningful way. For example, an attacker might try to substitute a read from a regular file with a read from a sensitive file; however, doing so would require changing the file handle argument, failing the template match. Thus, at a minimum, *Ellipsis* provides comparable security to commodity audit frameworks and may provide improved security by avoiding the common problem of log event loss. A positive side effect of *Ellipsis* is built-in partitioning of execution flows, benefiting provenance techniques that utilize such partitions [67, 77, 78].

4.2 Information Loss

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

Table 5. Platform Setup

Platform	Raspberry Pi 4 Model B [2] (RPi4)
RAM	4 GB
Kernel	RT Linux [103] 4.19 from raspberrypi/linux [3], patched with *Ellipsis*
Kconfig Enabled	`CONFIG_PREEMPT_RT_FULL`, `CONFIG_AUDIT`, `CONFIG_AUDITSYSCALL`
Performance and Isolation	Power Management Disabled, CPU Frequency Governor Performance [70], All kernel background tasks/interrupts to core 0 using the *isolcpu* kernel argument.
Audit Rules	Audit rules for capturing syscall events were configured to match against the evaluated application, *i.e.,* background process activity was not audited.
`kaudit_buffer` size	Set to 50K, unless otherwise specified. Larger values led to system panic/hangs.

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

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

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

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

4.3 Demonstration: Throttle Override Attack

4.3.1 Application Description. 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 [14]. It has been installed in over a million vehicles and has been the basis for many industrial and academic projects [10, 21]. We chose the quadcopter variant of ArduPilot, called ArduCopter, as it has the most stringent temporal requirements within the application suite. ArduPilot periodically updates actuation signals that control the rotary speed of motors that power rotors. The periodic updates are responsible for maintaining vehicle stability and responding, in real time, to any perturbations (*e.g.,* wind).

The platform setup is described in Table 5. The RPi4 board was equipped with a Navio2 Autopilot hat [6] for sensor/actuator interfaces. Among the syscalls observed in the trace of ArduPilot, we found that only a small subset of

syscalls were relevant to forensic analysis [44]: `execve, openat, read, write, close` and `pread64`. Upon running the template generation script on the application binary, we obtained the most frequently occurring templates for three tasks ($n = 1$, for each task), consisting of 14 `write`, 16 `pread64` and 1 `read`, respectively. These templates include expected values corresponding to the file descriptor and count arguments of the syscalls as well as temporal constraints. Templates were loaded into the kernel when evaluating *Ellipsis* or *Ellipsis-HP*. Auditing was set up to audit invocations of the syscalls made by the ArduPilot application as mentioned above. Complete templates are provided in Appendix A.

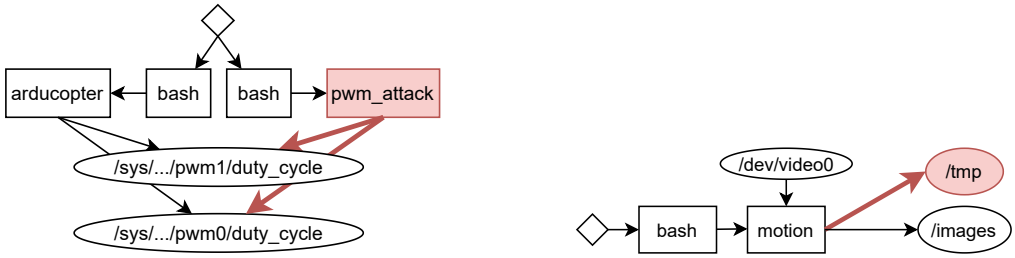
4.3.2 Attack Scenario. Let's consider a stealthy attacker who wants to destabilize or take control of unmanned drones. To achieve this, the attacker first gains control of a task on the system and attempts to override the control signals. An actuation signal's effect depends on the duration for which it controls the vehicle, therefore, naïvely overriding an actuation signal is not a very effective attack as the control task may soon update it to the correct value, reducing the attack's effect. The attacker instead leverages side-channel attacks such as Scheduleak [33] during the reconnaissance phase of the attack to learn when the control signals are updated. Armed with this knowledge, the attacker overrides the actuation signals immediately after the original updates, effectively taking complete control, with little computational overhead. Using tools provided with Scheduleak [33], a malicious task is able to override actuation signals generated by ArduPilot. This setup is run for 250 seconds and audit logs are collected with *Ellipsis*.

4.3.3 Results. Overriding throttle control signals involves writing to files in `sysfs`. This attack behavior can be observed in audit logs as sequences of `openat, write` and `close` syscalls. Combining templates with the obtained audit log yields the attack graph in Figure 4a. *Ellipsis* correctly identifies that ArduPilot is only exhibiting benign behaviors, reducing its audit logs. *Ellipsis* preserves detailed attack behaviors for the malicious syscall sequences. *Ellipsis* did not lose audit events throughout the application runtime. In contrast, Linux Audit loses audit events (§5.2), due to buffer overflows (§5.3), potentially losing critical forensic evidence. Linux Audit would also have captured the complete log from the benign ArduPilot task, complicating the eventual forensic analysis.

4.3.4 Discussion. Scheduleak [33] invokes `clock_gettime` syscall frequently to infer task activation times. Such syscalls are irrelevant for commonly used forensic analysis as they don't capture critical information flows. Despite the lack of visibility in the reconnaissance phase of the attack, auditing can capture evidence of attacker interference that creates new information flows, as shown in Figure 4a. We have demonstrated that when a process deviates from the expected behaviors, *e.g.,* due to an attack, *Ellipsis* provides the same security as Linux Audit. *Ellipsis* all but eliminates the possibility of losing portions of the malicious activity due to `kaudit` buffer overflow. However, it is impossible to guarantee that no events will ever be lost with malicious activities creating unbounded new events. *Ellipsis* improves upon Linux Audit by *(a)* freeing up auditing resources which can then audit malicious behaviors, and *(b)* reducing the audit records from benign activities that must be analyzed as part of forensic provenance analysis. Stealthy attacks like this also show the role of auditing in improving vulnerability detection and forensic analysis on RTS.

4.4 Demonstration: Data Exfiltration Attack

4.4.1 Application Description. Motion [5] is a soft real-time video analysis application. It monitors camera images and detects motion by tracking pixel changes between consecutive image frames. It is primarily used for surveillance and stores images when movement is detected. Images are stored at a location specified by the system administrator. Using the platform setup described in Table 5, we ran Motion v4.3.2 using a webcam as a video source. While not commonly considered forensically relevant, we include `ioctl, rt_sigprocmask` and `gettimeofday` in our audit



(a) (§4.3) Throttle Override Attack

(b) (§4.4) Data exfiltration attack

Fig. 4. Attack graphs created using *Ellipsis* audit logs.

ruleset as these syscalls are used to capture frames from video devices and maintain video frame rates. By running the template generation tools we obtained two templates that describe how Motion *(i)* captures an image frame and *(ii)* captures an image frame with movement and saves it to the file system.

4.4.2 Attack Scenario. The attacker inserts malicious code into the victim application to save images to an attacker-controlled location when motion is detected, as shown in Figure 4b. The attacker can exploit another process running on the system to exfiltrate these images out of the system at a later point in time, successfully leaking sensitive information. Both Motion and the malicious application are audited by *Ellipsis* for 5 minutes. We introduce movement in the camera's field of view to trigger image stores. Images get stored in both benign and malicious locations in the system. The attack can be realized using the following code snippet, developed by Yoon *et al.* [120] :

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

4.4.3 Results. *Ellipsis* correctly reduces audit logs that correspond to the capture of image frames where motion is not detected because that behavior matches the templates and 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` syscalls that differ from the behavior described by the templates, therefore retaining complete audit logs generated in response to observed movement.

4.4.4 Discussion. Since the attack continued throughout the experiment, we observe only a 25% reduction in audit log size with *Ellipsis* compared to 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 regular levels of reduction, *i.e.,* $\approx$ 90% (§6). We also evaluated this stealthier version of the attack where the attackers only switch the destination directory for a brief time to hide evidence of malicious activity. As motion detection drives the creation of images on disk, it may not be immediately apparent that an attack has occurred without observing audit logs. *Ellipsis* compares syscall arguments against expected values and hence can identify changes to the filename argument passed to the `openat` system call in the attack trace.

4.5 Summary

Owing to the rigorous syscall order and argument checking, *Ellipsis* captures the same information as lossless auditing by Linux Audit. Its audit event volume reduction does not lose relevant information, rather reducing the benign activity records helps avoid loss of records of malicious activity. The reduced audit log volume also simplifies eventual forensic analysis. However, it should be noted that *Ellipsis*'s security goal is to match the protection and forensic analysis capabilities provided by Linux Audit. Any malicious attacks that evade Linux Audit, evade *Ellipsis* too. Any forensically relevant information, not expressed by audit events, is not recorded by Linux Audit or *Ellipsis* alike. Having established *Ellipsis*' security impacts, we now present case studies, using applications *i.e.,* ArduPilot (§5) and Motion (§6), investigating in detail the audit record volume reduction and overheads of *Ellipsis*.

5 CASE STUDY: ARDUPILOT

5.1 Application Description

We use the same setup as described in Table 5 and Section 4.3.1. Additionally, we instrumented the application for measuring the runtime overheads introduced by auditing. Among the seven tasks spawned by ArduPilot, we focus primarily on a task named FastLoop for evaluating temporal overheads as it includes the stability and control tasks that need to run at a high frequency to keep the QuadCopter stable and safe.

5.2 Audit Completeness

Experiment. We ran the application for 100K iterations at task frequencies of 100 Hz, 200 Hz, 300 Hz and 400 Hz[4], measuring audit events lost. The fast dynamics of a quadcopter benefit from the lower discretization error in the ArduPilot's PID controllers at higher frequencies [110] leading to more stable vehicle control.

Observations. Figure 5 compares the log event loss for Linux Audit, *Ellipsis*, and *Ellipsis-HP* across multiple task frequencies. We observe that Linux Audit lost log events at all task frequencies above 100 Hz. In contrast, *Ellipsis* and *Ellipsis-HP* did not lose the audit event log at any point in the experiment.

Discussion. Because this ArduCopter task performs critical stability and control functions, reducing task frequency to accommodate Linux Audit may have considerable detrimental effects. Further investigation (§5.3) revealed that Linux Audit dropped log events due to `kaudit` buffer overflow, despite the buffer size being 50K. In contrast, *Ellipsis* can provide auditing for the entire frequency range without suffering log event loss.

5.3 Audit Buffer Utilization

Experiment. The size of the `kaudit` buffer is determined by a "backlog limit" configuration, that controls the number of outstanding audit messages allowed in the kernel [4]. The default configuration is 8192 but as noted before (Table 5) we set it to 50K. The `kaudit_buffer` state was sampled periodically, once every 2 seconds, by querying the audit command-line utility *auditctl* during the execution of the application for 100K iterations. Figure 6 shows the comparison of the percentage utilization of the audit buffer by Linux Audit, *Ellipsis*, and *Ellipsis-HP* over time.

Observations. From Figure 6, we see that for Linux Audit, the utilization of the `kaudit_buffer` rises quickly and remains close to 100% for the majority of the runtime, resulting in loss of audit messages, as measured earlier (§5.2). In

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

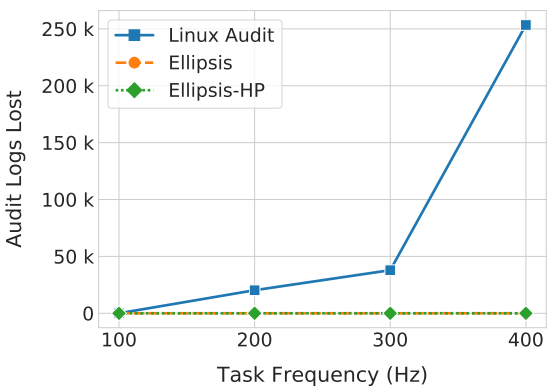


Fig. 5. (§5.2) Number of audit events lost vs. frequencies of the primary loop in ArduPilot, for 100K iterations. The frequency of the Fast Loop task is varied from 100 Hz to 400 Hz and the number of logs lost are plotted. *Ellipsis* and *Ellipsis-HP* suffer no log loss at any frequencies and hence their lines overlap exactly with the X-axis.

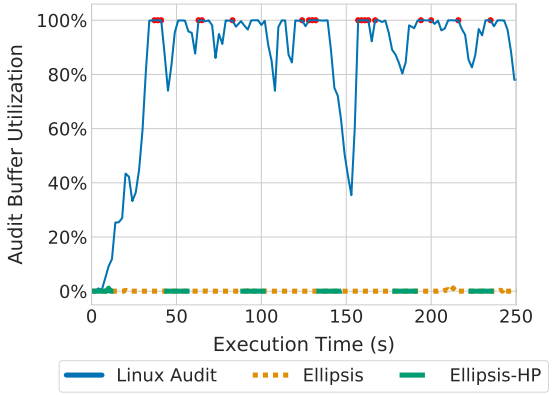


Fig. 6. (§5.3) Audit buffer utilization. Additional red annotations signify times when the buffer is filled. *Ellipsis* and *Ellipsis-HP* did not use more than 2% buffer space.

contrast, *Ellipsis* and *Ellipsis-HP* ensure that the buffer utilization remains negligible throughout the execution. As noted before (Table 5), the buffer size is already set to the largest value the platform can support without panics or hangs.

Discussion. When the `kaudit_buffer` is full, new audit messages are lost; hence, to ensure that suspicious events are recorded, it is essential that *the buffer is never full. Ellipsis* can 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 the application profile. So we ran further experiments to determine the minimum `kaudit_buffer` size with which *Ellipsis* and *Ellipsis-HP* can still achieve complete auditing. These further experiments are free from any sampling limitation. We find that a buffer of 2.5K for *Ellipsis* and 1.5K for *Ellipsis-HP* was enough to support lossless auditing under normal operation. This reduced memory requirement is valuable for RTS that run on resource-constrained platforms. The reduced time that the buffer holds audit logs, reduces the attack window for recently identified race condition attacks on the audit buffer [86]. While the buffer utilization under normal operation is vastly reduced, the buffer limit should still be kept larger than the observed minimum utilization to capture anomalous behavior without loss.

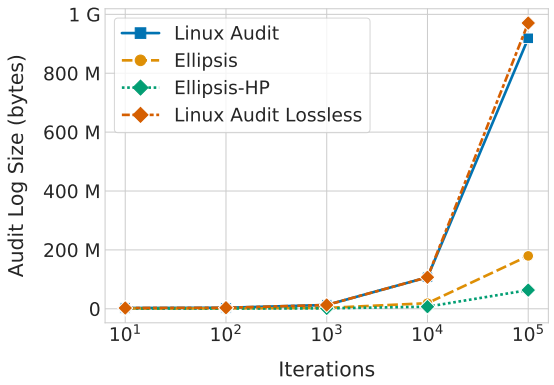


Fig. 7. (§5.4) Total size on disk of the audit log (Y-axis), captured for different numbers of iterations (X-axis). The size of the log is measured as file size on disk in bytes. 10^5 iterations take $\approx$ 250 seconds to complete. For Linux Audit, we measure the actual size of the log on disk albeit logs are lost. Linux Audit Lossless provides an estimate of the size of the log if the auditing was lossless.

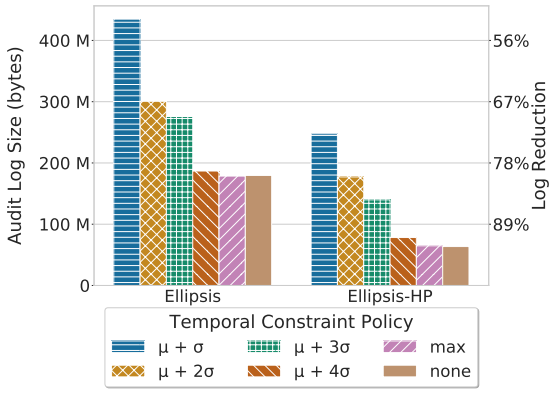


Fig. 8. (§5.5) Log size (Y-axis left) under varying temporal constraint policies. The right Y axis shows the % reduction in log size compared to Linux Audit. $\mu + 4\sigma$ covers 99.5% of the total 10^5 iterations. *none* policy result herer is same as 10^5 results in Figure 7.

5.4 Audit Log Size Reduction

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

Observations. Figure 7 compares the storage costs in terms of file size on disk in bytes. The storage costs for all systems over shorter runs were found to be comparable, as the cost of auditing the initialization phase of the application ($B_A * f$) tends to dominate over the periodic loops. Over a 250 second runtime (10^5 iterations) the growth of log size in *Ellipsis* was drastically lower compared to vanilla Linux Audit, with storage costs reducing by 740 MB, or **80%**, when using *Ellipsis*. *Ellipsis-HP* provides a more aggressive log size reduction option by lowering storage costs by 860MB, or **93%**, compared to Linux Audit. *Linux Audit Lossless* estimates the log size had Linux Audit not lost any log events using the number of logs lost and the average size of each log entry.

Discussion. The observations line up with our initial hypothesis that the bulk of the audit logs generated during a loop iteration would exactly match the templates. Thus, in *Ellipsis* by reducing all the log messages that correspond to a template down to a single message, we see a vast reduction in storage costs while ensuring the retention of all the audit data. *Ellipsis-HP* takes this idea further by eliminating audit log generation over extended periods of time if the application exhibits expected behaviors only. For RTS that are expected to run for months or even years without failing, these savings are crucial for continuous and complete security audit of the system.

5.5 Temporal Constraint Policy

Experiment. We explore here the impact of different temporal constraint policies. Temporal constraints are applied, intra-task, for *Ellipsis* and additionally inter-task for *Ellipsis-HP*. While the constraint values on expected runtimes and expected inter-arrival times of task instances are learned and applied separately for each task, a common policy can be enforced. For example the policy *max* implies that all timing constraints are set to the maximum value that was observed for them during the learning phase. Other policies explored in this experiment are based on the average

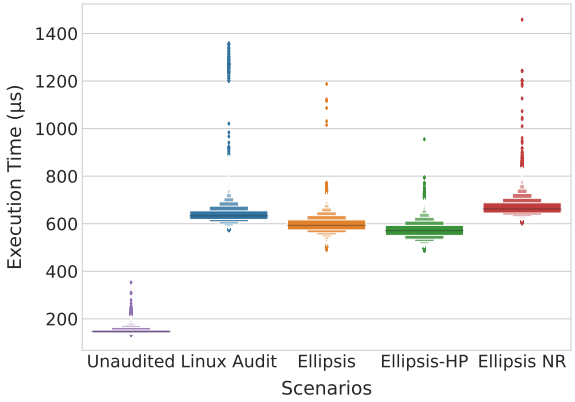


Fig. 9. (§5.6) Comparison of runtime overheads of ArduPilot main loop. The task period and deadline is 2500 μs. Ellipsis NR (No Reduction) refers to a forced scenario where each template match fails, leading to no log reduction.

(μ) and standard deviation (σ) of the time intervals observed during the learning phase. The *none* policy disables all temporal constraints and represents the best case in this experiment.

Observations. Figure 8 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 RTS.

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 *Ellipsis* should be used at runtime to record unexpected behaviors, *i.e.*, not seen during the learning phase, while eliminating audit logs for expected behaviors. The policies *max* and $\mu + 4\sigma$ most closely correspond with this. Further is it notable that since *max* and *none* policy yielded almost the same log size, the *max* constraint provided temporal violation checks with negligible cost.

5.6 Runtime Overheads

Experiment. This evaluation measures the execution time in microseconds (μs), for the Fast Loop task of ArduPilot, for 1000 iterations, under various auditing setups. The small number of iterations kept the generated log volume within `kaudit_buffer` capacity, avoiding overflows and audit events loss in any scenario. This avoids polluting the overhead data with instances of event loss. The time measurement is based on the monotonic timer counter. This process was repeated 100 times to capture the distribution of these measurements over longer application runs. *Ellipsis* and *Ellipsis-HP* refers to the normal execution of the application with their respective reduction techniques. To evaluate the absolute worst case for *Ellipsis*, a synthetic worst-case *Ellipsis* NR (No Reduction) is also included. In *Ellipsis* NR, the last syscall in each template is manually modified to an invalid syscall. All syscalls, before the last, match normally. The last syscall match, however, always fails, due to the modification. Therefore, all additional overheads to match a template are incurred, without any eventual gain of successful event reduction. *Ellipsis* NR is also the worst case for *Ellipsis-HP*.

Observations. Figure 9 shows the distribution of 100 execution time samples for each scenario. *Ellipsis*, *Ellipsis-HP*, and *Ellipsis* NR have nearly the same overhead as Linux Audit. On average, *Ellipsis*'s overhead is *0.93*x and *Ellipsis-HP*'s overhead is *0.90*x of Linux Audit. The observed maximum overheads show a greater improvement. *Ellipsis*'s observed

Table 6. Motion Application Log Reduction for different configurations

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

maximum overhead is *0.87*x and *Ellipsis-HP*'s *0.70*x of Linux Audit. *Ellipsis* NR shows a *1.05*x increase in average overhead and *1.07*x increase in maximum observed overhead compared to Linux Audit.

Discussion. Ellipsis adds additional code to syscall auditing hooks, which incurs small computational overheads. When template matches fail (*Ellipsis* NR), this additional overhead is visible, although the overhead is not significantly worse than the baseline Linux Audit. However, in the common case where audit events are reduced by *Ellipsis*, this cost is masked by reducing the total amount of log collection and transmission work performed by Linux Audit. This effect is further amplified in *Ellipsis-HP* owing to its greater reduction potential (§5.4). Thus, *Ellipsis*'s runtime overhead depends on the proportion of audit information reduced in the target application. Thus, while reducing the runtime overhead of auditing is not *Ellipsis*' primary goal, it nonetheless enjoys a modest performance improvement by reducing the total work performed by the underlying audit framework.

5.7 Summary of Results

Ellipsis provides complete audit events retention while meeting temporal requirements of the ArduPilot application, with significantly reduced storage costs. *Ellipsis-HP* improves this further. The temporal constraint allows additional temporal checks, detecting anomalous latency spikes with effectively no additional log size overhead during normal operation. Under normal operation, *Ellipsis* and *Ellipsis-HP* also reduce the computational overheads of auditing. Linux Audit and *Ellipsis*' synchronous overheads, which can potentially interfere with temporal properties important to RTS, are analyzed in detail in Section 7.

6 CASE STUDY: MOTION

6.1 Application Description

We use the same setup as described in Table 5 and Section 4.4.1, equipped with a camera [13]. Motion [5] is a soft real-time video analysis application where the application execution paths and behavior vary depending on configuration and inputs. Motion application detects movement in camera inputs and saves images when movement is detected within frames. We point a camera to a screen showing a still image (Input = Still) or a video[5] with random motion (Input = *Motion*). Motion's behavior is determined via its configuration file. *emulate_motion* when set to on, it causes an image to be saved at a fixed rate of 2 Hz. When *emulate_motion* is set to off, an image is stored only when motion is detected in the input stream. Thus Motion Detection is *Enabled* when *emulate_motion=off* and Disabled when *emulate_motion=on*. Application logging verbosity is set to minimum level for Limited and maximum level for *Verbose*.

[5] https://www.youtube.com/watch?v=cElhIDdGz7M

6.2 Audit Log Size Reduction

Experiment. In this experiment, we run Motion with varying configurations and report the log reduction percentage. For each configuration, green colored option increases execution paths variability, while red colored option decreases the variability. For each combination templates are learned over a 120 second execution. The application is then audited with Linux Audit and *Ellipsis* for 300 seconds each.

Observations. We provide the rate of audited syscall events and size on disk for audit logs of both Linux Audit and *Ellipsis* with the log size reduction percentage in Table 6. Reduction is the percentage reduction in log size generated by *Ellipsis* from that generated by Linux Audit. In most cases, a log reduction of $> 90\%$ is achieved. The lowest reduction occurs when the application only processes the camera feed, never saving any images. The resultant log, with a low number of syscalls and lowest size, contains disproportionately high events from the setup phase of the application, leading to a lower reduction by *Ellipsis*, which is still quite high at 81.44%. We also experimented with a doubled rate of storing images (4 Hz) but no differences were observed, as is expected. Log loss was not observed in any scenario.

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

7 SYNCHRONOUS OVERHEAD ANALYSIS

We now analyze the auditing overhead added to each syscall, *i.e.,* Figure 1 ②, ③, ④. The primary purpose of this analysis is to determine if these synchronous overheads violate properties important to RTS, *e.g.,* execution time variability, resource contention, and overhead scaling.

7.1 Setup

The evaluation platform with static configurations is detailed in Table 5. We configured audit rules to only match the applications we are running, *i.e.,* background process activity was not audited. Buffer size is set to 50K to avoid overflows, however, in some evaluations smaller buffer sizes are used. The values and reasoning are explained where applicable. `isolcpu` kernel argument is used to remove background activity from interfering with the evaluation applications. However, auditing background daemons, `kauditd` and `auditd` are scheduled on the same cores as the evaluation application where asynchronous `kaudit_buffer` drainage is part of the evaluation. Due to the large number of data points recorded, we show the distribution of latency values as BoxenPlots [56].

Since Linux Audit and *Ellipsis* share much of their infrastructure, we focus primarily on Linux Audit here, evaluating *Ellipsis* only where differences exist. For *Ellipsis* we evaluate two configurations, the best case of *Ellipsis-HP*, named ***Ellipsis B***, where all reduction attempts succeed, and the worst case ***Ellipsis NR*** where all reduction attempts fail, as also in Section 5.6. Since the success of *Ellipsis* is based on how predictable the execution paths of the taskset are, in RTS, *Ellipsis* performs closer to *Ellipsis B* than *Ellipsis NR*, as evidenced by high reduction percentages in Sections 5.4 and 6.2.

7.2 Benchmarks

(a) μ − bench. We execute a microbenchmark to observe variations in syscall execution time in the presence of external factors such as auditing, RT scheduling priorities, background stress, and parallel execution. We primarily use `getpid` syscall, a low latency non-blocking syscall. Since the goal of this analysis is to measure the properties of auditing

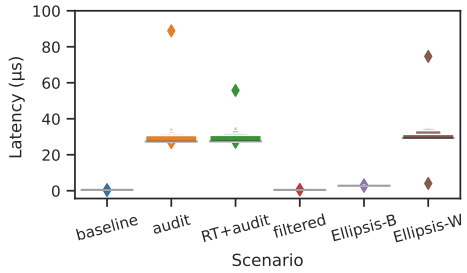


Fig. 10. Latency of `getpid` for various auditing scenarios.

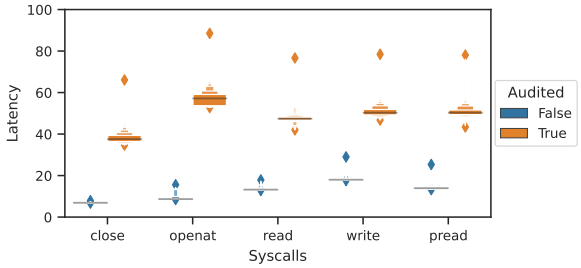


Fig. 11. Execution times of various syscalls, when audited and not.

hooks in the syscall execution path, it is necessary to minimize the latency of the syscall itself. Other syscalls were also evaluated to ensure the generality of observations as shown in Figure 11.

```
for (i = 0; i < 1000; i++) {
  clock_gettime(CLOCK_MONOTONIC, start_data);
  syscall(); // Replaced with specific syscalls
  clock_gettime(CLOCK_MONOTONIC, stop_data);
  // Get latency of reading the timer.
  clock_gettime(CLOCK_MONOTONIC, empty_start);
  clock_gettime(CLOCK_MONOTONIC, empty_stop);
  latency = timespec_subtract(start_data, stop_data) - timespec_subtract(empty_start, empty_stop);}
```

(b) Cyclictest. Cyclictest [104] measures the latency between a thread's scheduled and actual wake-up time. These threads only use `clock_gettime` and `clock_nanosleep` syscalls, the latter of which is added to auditing rules.

(c) Stress. Stress[6] application creates computational, file io, and virtual memory loads on the system. We audit the `sync`, `mmap2` and `munmap` syscalls used by this application.

(d) Scaling. We use synthetic tasksets to show that *Ellipsis*' overhead per syscall scales independent of the size of the template. The application setup is described in detail in Section 7.6.

7.3 Latency per Syscall

Experiment. We measure here the auditing overhead to a single syscall. Figure 10 shows the latency to execute the $\mu-bench$ issuing a `getpid` syscall. Each column shows the distribution of `getpid` execution latency over 1000 iterations. The *baseline* scenario has auditing disabled and no other application running. For the *audit* scenario, the baseline is repeated but the benchmark application is under audit with Linux Audit. For the *RT+audit* scenario, we execute the previous scenario with the benchmark application running at an RT priority. In the *filtered* scenario, the benchmark application is still under audit but `getpid` is no longer in the auditing filters. *Ellipsis B* and *Ellipsis NR* (§7.1) are equivalent to *RT+audit*, with auditing system changed. Figure 10 shows the results of this evaluation. Figure 11 shows the latencies for *baseline* (Audited = False) and *RT + audit* (Audited = True) scenarios for different syscalls.

Observations: The observed maximum overhead was just under 100 μs (*audit*), reducing to 60 μs when the application under audit is assigned RT priority. The overhead of auditing is limited to only the syscalls being audited.

[6]https://packages.ubuntu.com/bionic/stress

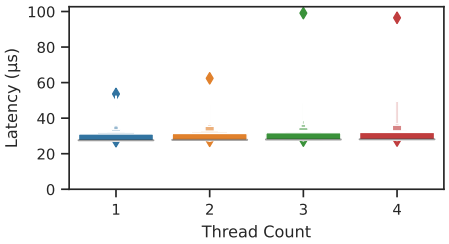


Fig. 12. Latency of a syscall execution with same priority parallel threads.

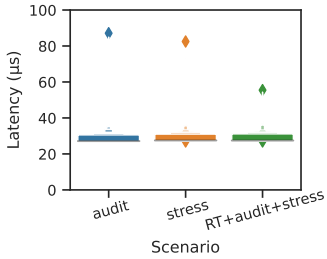


Fig. 13. The overhead of auditing `getpid` with stress.

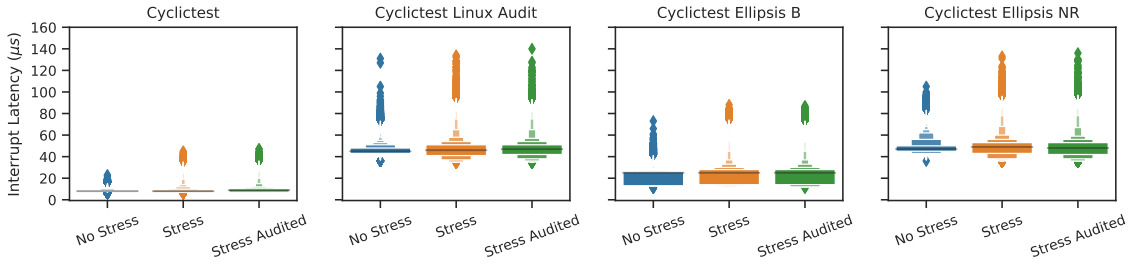


Fig. 14. Interrupt latency from Cyclictest. Cyclictest is run for 5 min with 1 ms interrupt intervals, at high priority, under different auditing conditions. Background priority Stress (§7.2) status is indicated by X-Axis.

The overheads for different syscalls vary due to differences in auditing work for each syscall, *e.g.,* `openat` creates a new access interface for the application, requiring more audit information to be recorded. *Ellipsis NR* behaves similarly to Linux Audit, while *Ellipsis B* shows significant overhead reduction.

7.4 Resource Contention

Experiment. Auditing introduces a resource shared among the RT tasks being audited, the `kaudit_buffer`, protected by a spinlock. Parallel accesses can lead to contention and blocking, even between tasks with the same RT priority. To test the presence of contention we run the $\mu - bench$ issuing `getpid` syscalls. While measuring the latency of one thread, we introduce an increasing number of additional threads running the same $\mu - bench$ at the same RT priority. The threads are synchronized via a barrier to start executing syscalls at the same time. $\mu - bench$ is a tight loop that runs a single `getpid` syscall in each thread, hence ruling out the cache or memory bandwidth as sources of contention.

Observations: The execution times for the syscalls from the thread under observation are shown in Fig. 12. In the average case, we observe only a small difference in the latency of `getpid` regardless of the parallel workloads. While the observed worst-case overhead is greater with 3 or 4 threads, it is still under 100 μs even when the tasks on all 4 cores are being audited. Delays due to contention would occur if multiple threads try to access the spinlock at the same time; but even with the fast `getpid` call the threads minimally contend on the spinlock. This result intuitively follows as the shared spinlock covers a small critical section containing fast pointer manipulations only, making contention uncommon even in this unfavorable scenario with repeated calls to a fast syscall. For brevity and because *Ellipsis* does not modify this record insertion behavior, only volume, *Ellipsis* is not evaluated separately here.

7.5 Priority Scheduling and Blocking

Experiment. An important concern is that a low-priority task's usage of the auditing system could block a higher-priority task by virtue of shared usage of the auditing framework. To investigate this, we run cyclictest (§7.2), at high priority, under different auditing and background Stress (§7.2) conditions. In Figure 14, *Cyclictest* is the baseline, followed by Cyclictest being audited by Linux Audit, *Ellipsis B*, and *Ellipsis NR*. Stress application state is noted in the X-Axis, being absent, present, and present + audited by Linux Audit, respectively. *Ellipsis* does not introduce any additional shared resources and is hence not used to audit the Stress. Each column shows the distribution of interrupt latency as reported by cyclictest. A similar evaluation for $\mu - bench$, for Linux Audit only, is shown in Figure 13.

Observations: From Figure 14 it is evident that the impact of the Stress application on cyclictest's interrupt latency is not affected by the Stress application being audited. Therefore we can conclude that auditing itself is not creating any significant new blocking of the high-priority cyclictest. This is an expected, albeit reassuring, result because the insertion of new events into the `kaudit` buffer does not block (§7.4) and audit daemons are not run at RT priority. Additionally, *Ellipsis NR* again performs the same as Linux Audit while *Ellipsis B* has significantly reduced overhead.

7.6 Synthetic Tasks: Overhead Scaling

Experiment. Because *Ellipsis* adds template matching logic in the critical execution path of syscalls, a potential concern is the overhead growth for tasks with long syscall sequences. In this experiment, we measure execution time for tasks that execute varying counts of `getpid` syscalls (10, 20, 30 ... 300). `getpid` is a low latency non-blocking syscall, which allows us to stress-test the auditing framework. As the max template length (*i.e.,* syscall count) observed in real application loops was 29, we analyze workloads of roughly 10 times that amount, *i.e.,* 300. The execution time for each task is measured 100 times. Temporal constraints are not used. Since the tasks have a single execution path *i.e.,* a fixed count of `getpid` syscalls, *Ellipsis*' audit events reduction always succeeds. For *Ellipsis* NR (No Reduction) we force template matches to fail at the last entry (same as §5.6). This represents the worst-case scenario.

Observations. Figure 15 shows the average syscall response time as the number of syscalls in the task loop increases. The primary observation of interest is that the *time to execute a syscall is roughly constant*, independent of the number of syscalls in the task and template. The higher value at the start is due to the non-syscall part of the task that quickly becomes insignificant for tasks with a higher number of syscalls. The maximum variance is negligible at $< 1.3\mu s$.

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

7.7 Remarks

Linux Audit and *Ellipsis* do not introduce any significant issues of blocking or contention. 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 limited outlier cases, *the latency introduced by syscall auditing can be measured and bounded.* This works well for the latency-sensitive RT systems that RT Linux is intended for, hence Linux Audit and *Ellipsis* are good candidates for auditing RTS based on RT Linux [103].

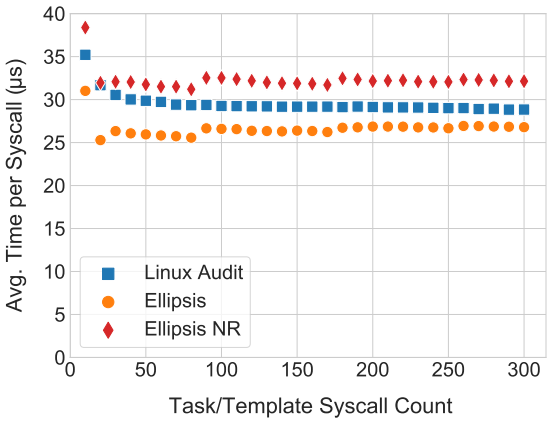


Fig. 15. (§7.6) Avg. execution latency of `getpid` syscall (Y-axis) with varying task/template lengths (X-axis)

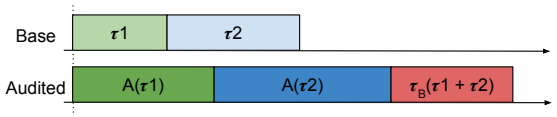


Fig. 16. Sample timelines for two periodic tasks $\tau 1$ and $\tau 2$. $A(\cdot)$ is the increased computation time of the task including audit log generation overhead, *i.e.,* $A(\tau_i) = C_i + C_i^A$. τ_B represents `kauditd` and `auditd` daemons as they handle the logs generated by the RT tasks $\tau 1$ and $\tau 2$. For brevity, only one instance per task is shown.

8 SCHEDULABILITY ANALYSIS

Having established the suitability of auditing for RTS, we now present the schedulability analysis for Linux Audit and *Ellipsis*. The base scenario in Figure 16 shows the execution timeline for two periodic tasks that are not under audit. Audit overheads can be divided into two parts, as shown in the Audited scenario in Figure 16: *(i)* $A(\cdot)$ represents the task execution with the synchronous overhead of log generation, C_i^A, (Figure 1 ②, ③, ④), and *(ii)* τ_B is a task representing the background daemons that maintain the audit logs, transporting them from `kaudit_buffer` to userspace and eventually to persistent storage, *i.e.,* `auditd` and `kauditd` in Figure 1. C_i^A depends on the number of audited events in a task and the number of events recorded, while τ_B's computational time, C_B, varies only with the number of events recorded. As *Ellipsis* reduces the recorded events, it also reduces the computation times C_i^A (§5.6) and C_B (§5.3).

8.1 Real-Time Task Model

We consider a multi-core system with M identical cores, running RT applications in a preemptive operating system (*e.g.,* Linux). The system consists of N real-time tasks and is scheduled using a fixed-priority scheduling policy. An RT task, τ_i, can be periodic or sporadic and is characterized by a tuple (T_i, D_i, C_i) where T_i is the period (or the minimum inter-arrival time if τ_i is a sporadic task), D_i is the relative deadline and C_i is the worst-case execution time (WCET). Let us represent the taskset as $\Gamma = \{\tau_i(C_i, T_i, D_i)\}\forall i$. The hyperperiod of this taskset is the least common multiple (LCM) of the periods of each task $\tau_i \in \Gamma$ and we denote this as $LCM(\Gamma)$. We assume that without auditing Γ is schedulable, *i.e.,* the worst-case response time (WCRT) for each task is less than or equal to its deadline.

8.2 Schedulability

Auditing adds synchronous overheads to applications under audit, *i.e.,* the computational time to generate audit record and log it to `kaudit_buffer` (§2.1). Let C_i^A represent the additional (worst-case) computational time for log generation, as evaluated in Section 7. The WCET of each audited task τ_i is then $C_i^{\mathrm{I}} = C_i + C_i^A$. Let us represent the taskset as $\Gamma_{\mathrm{I}} = \{\tau_i(C_i^{\mathrm{I}}, T_i, D_i)\}, \forall \tau_i \in \Gamma$. 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, can therefore be obtained by standard multicore global fixed-priority scheduling analysis techniques [46, 99] 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 `kaudit_buffer` is filled by the RT tasks at runtime and needs to be cleared periodically (§2.1). Let τ_B represent the buffer draining mechanisms, *i.e.,* background daemons `auditd` and `kauditd` (§2.1). τ_B is invoked once in each hyperperiod, *i.e.,* $T_B = LCM(\Gamma)$ and needs to complete its execution before its next periodic interval (*i.e.,* $D_B = T_B$). τ_B is set to execute with the lowest priority and therefore does not interfere with the RT tasks. We assume here that τ_B is the only background task in the system. This simplification is based on the fact that only `auditd` and `kauditd`, which combined are τ_B, are the only background tasks that participate in auditing. However, if other unrelated background tasks exist their interference can be accounted for. We assume that the `kaudit_buffer`, the size of which is defined by the system administrator, is large enough to hold all audit records generated between consecutive invocations of the buffer draining task. At worst, this size is all the records that are generated over 2× the taskset hyperperiod.

Let's now define the augmented taskset $\Gamma_{\mathrm{II}} = \Gamma_{\mathrm{I}} \cup \{\tau_B\}$. This taskset can be audited with no log loss if the WCRT of the buffer draining task is less than its period, *i.e.,* $R_B \leq T_B$. Note that the taskset hyperperiods $LCM(\Gamma) = LCM(\Gamma_{\mathrm{I}}) = LCM(\Gamma_{\mathrm{II}})$. Further, τ_B behaves similarly to garbage collectors that have been studied in prior work [97].

Let's 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 RT task τ_i within the window x, *i.e.,* 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 RT tasks. The cumulative length[7] of this interval is $x - C_B$ [16], where C_B represents the cumulative computational time of τ_B. A sufficient condition of 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 RT tasks. Let us assume that there exists a function $\mathcal{R}_B(\cdot)$ that returns the minimum value of x for schedulability conditions

$$x \geq C_B \text{ and } \sum_{\tau_i \in \Gamma} I_B^i(x) < M(x - C_B), \tag{10}$$

i.e., $\mathcal{R}_B(\cdot) = \min\{x\}$ if Eq. (10) 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 [46, 99]. The logging mechanism therefore will work as expected if the following conditions hold: $\mathcal{R}_B(\cdot) \leq T_B$, $T_B = LCM(\Gamma_{\mathrm{II}}) = LCM(\Gamma)$ and $R_i \leq D_i, \forall \tau_i \in \Gamma_{\mathrm{II}}$. However, if $\mathcal{R}_B(\cdot) > T_B$ the accumulation of logs in the buffer would eventually lead to $\mathcal{R}_B(\cdot) = \infty$, overflowing the buffer.

8.3 Malicious Attacks

Prior analyses (§8.2, [97]) assume that the buffer usage is predictable, or at minimum bound-able. However, this assumption is valid during the expected benign activity only. Despite the schedulability conditions established in Section 8.2 being met, malicious attacks on the RTS, generating an unpredictable and arbitrarily high amount of audit events can cause `kaudit_buffer` overflow. This is further complicated by the fact that there is no clear priority scheme for the global `kaudit_buffer` usage, unlike prior work [36]. Audit events from low-priority or even background tasks can contain information critical to the security of the high-priority tasks.

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

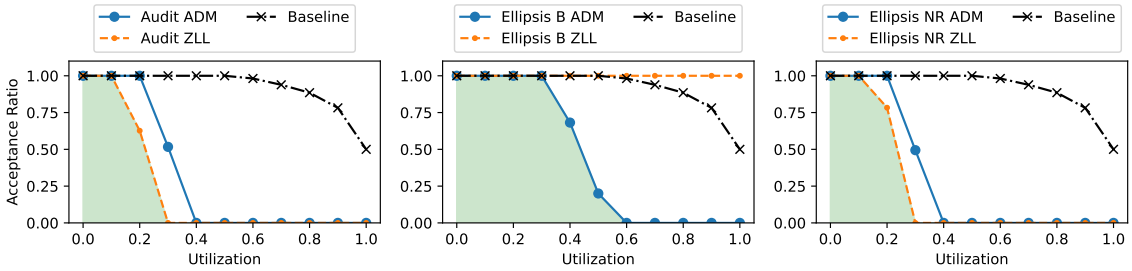


Fig. 17. Temporal and Auditing constraint analysis for periodic tasksets. Taskset utilization (X-axis) is plotted against the percentage of tasksets (Y-axis) for which All Deadlines are Met (ADM) or have Zero Log Loss (ZLL). Tasks have 5% syscall and 95% non-syscall workload. The shaded region shows the area where both ADM and ZLL requirements are satisfied. The ADM for Unaudited case is provided as a Baseline since utilization computation is based on it.

In such conditions, while it remains impossible to provide deterministic guarantees, *Ellipsis*' ability to severely reduce the audit record volume generated from benign activity maximizes the potential to fully retain the record of malicious activities (§5.4). Similarly, while the maximum `kaudit_buffer` demand cannot be bounded when the system is under attack, *Ellipsis* maximizes the buffer capacity available to audit the malicious events (§5.3).

8.4 Evaluation

Experiment. To study the impact of auditing we perform a schedulability evaluation. Table 5 details the platform setup. Tasksets (Γ) consist of upto 5 tasks (τ) with taskset utilization ($\sum_{i=0}^{4} \frac{C_i}{T_i}$) chosen uniformly $\in [0, 1]$ and hyperperiod ($LCM(\Gamma)$) of 1000μs. Tasks have a period between 100μs – 1000μs and have a utilization selected using UUniFast algorithm [22]. 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. Additionally, RT tasks and auditing daemons `kauditd` and `auditd` are isolated on core 3 (M = 1). All tasks are started together and each τ_i runs for $2 \times LCM(\Gamma)/T_i$ iterations. Tasks are executed with a rate monotonic priority.

`kaudit_buffer` is set to be large enough to hold exactly one hyperperiod worth of audit records. This ensures that if the second hyperperiod of the taskset starts with some audit logs in `kaudit_buffer` from the previous hyperperiod, records will be lost. For each taskset we monitor whether all tasks meet their temporal requirement *i.e.,* **All Deadlines Met (ADM)**. We also monitor whether any audit records/logs are lost *i.e.,* **Zero Log Loss (ZLL)**. 1000 tasksets are run, and the results for tasksets are grouped by utilization in groups of width 0.1.

Observations. Figure 17 shows the results. The Y-axis measures the fraction of the tasksets in each group for which ADM / ZLL requirements were satisfied. The synchronous $A(\cdot)$ component impacts ADM. *Ellipsis B* (§7.1) performs better on ADM metric allowing all tasksets with up to 0.3 utilization to meet all deadlines, compared to 0.2 for Linux Audit and *Ellipsis NR* (§7.1). The main point of difference is the τ_B task that deals with the maintenance of auditing information; log loss occurs when this task does not get enough computation time. Linux Audit and *Ellipsis NR* start experiencing log loss at a very small utilization of 0.1. In contrast, *Ellipsis B* achieves lossless auditing for all workloads.

Discussion. Although Linux Audit has been included in embedded Linux and *Ellipsis* has been developed to reduce auditing volume for RTS, their impact on the schedulability of RT tasksets required careful analysis. In this experiment, despite the low ratio of the task workload being audited (5%), tasksets under audit can meet deadlines for tasksets with

low utilization only. This is the trade-off for the security benefits of auditing. *Ellipsis*' performance is dependent on the success of the reduction attempts. In the worst case, *Ellipsis NR* performs identically to Linux Audit. However, in this worst-case, the taskset is exhibiting behaviors, previously unseen during testing and analyses. Since RTS with well-formed tasksets would not have such a condition occur during valid operations of the system, the actual performance of *Ellipsis* would align closer to *Ellipsis B*. *Ellipsis* is the better auditing system for RTS.

9 DISCUSSION

9.1 System Scope

Ellipsis is useful for any application that has predictable repeating patterns. When sequence counts are too numerous with no high-probability sequences, it may be possible that too much system memory would be required to achieve significant log reduction. That said, a large number of possible sequences is not detrimental to *Ellipsis* as long as there exist some high-probability sequences. *Ellipsis*'s efficacy is also not dependent on specific scheduling policies unless tasks share process and thread ids; if task share process/thread ids and the scheduler can reorder them, *Ellipsis* cannot distinguish between event chains, leading to unnecessary template match failures. Lastly, we note that while we have motivated our design by discussing periodic tasks, *Ellipsis* is able to work effectively on any predictable execution profiles; *e.g.*,, *Ellipsis* would also be effective for aperiodic or timetable-triggered tasks, that are significantly prevalent in industrial RTS [9], or even non-RT applications that share the properties of predictability and repetition.

9.2 Limitations

Ellipsis is primarily an enhancement over Linux Audit, therefore, it shares many limitations of Linux Audit. Linux Audit suffers from *(i)* high runtime overheads; *(ii)* loss of events leading to incomplete and ineffectual security auditing, even in the absence of malicious activities; *(iii)* recording large amounts of extraneous audit data; *(iv)* limited scope of visible events, *e.g.,* syscalls. *Ellipsis* does not meaningfully change the runtime overheads, for better or worse (§5.6, §7.6). *Ellipsis* all but eliminates event loss during typical benign operation, however, under malicious attacks, recording of all events cannot be guaranteed, as the malicious activity may create audit events at unbounded rates. It is worth noting that *Ellipsis* vastly increases the auditing capacity available for auditing malicious events (§5.3). Any attack on Linux Audit that relies on overflowing the `kaudit_buffer` is therefore still possible on *Ellipsis*, made harder by the high performant reduction of benign expected behaviors (> 90%). *Ellipsis* systemically addresses the problem of recording extraneous audit data (§5.4). *Ellipsis* shares the visibility and scope of Linux Audit. Any malicious control or data flow modification, that does not involve syscalls, cannot be recorded with either Linux Audit or *Ellipsis*. However, it is noteworthy that without the use of syscalls, *i.e.,* by introducing additional syscalls or modifying the arguments of existing ones, the ability of malicious actors to cause harm is severely limited [42].

9.3 Auditing Hard Deadline RTS

Ellipsis, like Linux Audit and Linux itself, is unsuitable for hard-deadline RTS. All synchronous audit components must meet the temporal requirements for Hard RTS with bounded WCET, including syscall hooks and *Ellipsis* template matching. Additionally, the `kaudit_buffer` occupancy must have a strict upper bound. In this paper *Ellipsis* takes a long step forward, deriving high confidence empirical bounds (§5.6) to enable *Ellipsis*' use in firm- or soft-deadline RTS, which are prolific [9]. However, the strict bounds required for Hard RTS are a work in progress. Similarly, security auditing for specialized Real-Time Operating Systems, *e.g.,* FreeRTOS [7], will be investigated in future work.

9.4 Unfavorable Conditions

We consider here the impact of using *Ellipsis* to audit hypothetical RTS where our assumptions about RTS properties do not hold (§2.2). If the RTS may execute previously unknown syscall sequences, extra events would exist in the audit log. The audit log recorded by *Ellipsis* would thus be larger. Since safety, reliability, and timing predictability are important requirements for RTS [9] the gaps in code coverage can only be small. Hence the unknown syscall sequences will not have a major impact on audit events and log size. If known syscall sequences have a near uniform probability of occurrence, simply using templates for them all achieves high reduction ($n = N$). The tradeoff is additional memory required to store templates which is a small cost (Eq. (7)). Finally, if the above are combined, sequences with a substantial probability of occurrence would remain untested during the RTS development. For such a system, functional correctness, reliability, safety or timing predictability cannot be established, making this RTS unusable.

9.5 Code Coverage

Ellipsis is extraordinarily effective for RT applications tested with high code coverage to ensure reliability [35]. However, *Ellipsis* can be deployed on any system for any application under audit. The resultant log reduction benefit is proportional to the ratio of runtime spent in previously analyzed execution paths that are included as templates. Therefore, perfect code coverage is not a requirement for the use of *Ellipsis*.

9.6 Deployment Considerations

The mechanisms for template use are fully flexible. Any sequence for any task can be independently reduced with *Ellipsis*. However, to use *Ellipsis* beneficially, sequences with a high probability of occurrence (p_i) should be chosen *i.e.,* top n sequences by high p_i out of total N. The primary trade-off is the memory cost of storing templates, as in (7). For an RTS with limited memory, using (2) and (7), n value can be chosen for each task independently to minimize the *Ellipsis* events generated. The parameter n is chosen independently for each task, allowing highly optimized use of the main memory available for storing templates. A second trade-off is security. As the information lost by *Ellipsis* is minimal (§4), the trade-off on security is also minimal.

10 RELATED WORK

10.1 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 [18, 87, 89, 91] and taint tracking [20, 78], 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 [43, 50–53, 58, 65, 72, 74, 77, 81, 88, 90, 100, 112]; in this work, we make the use of such techniques applicable to RTS through the design of a system audit framework that is compatible with temporally constrained applications. Our approach to template generation in *Ellipsis* shares similarities with the notion of *execution partitioning* of log activity [52, 53, 65, 67, 77], which decomposes long-lived applications into autonomous units of work to reduce false dependencies in forensic investigations. Unlike past systems, however, our approach requires no application instrumentation to facilitate. Further, the well-formed nature of real-time tasks ensures the correctness of our execution units *i.e.,* templates.

10.2 Auditing RTS

Although auditing has been widely acknowledged as an important aspect of securing embedded devices [12, 39, 63], challenges unique to auditing RTS have received limited attention. Wang *et al.* present ProvThings, an auditing framework for monitoring IoT smart home deployments [111], but rather than audit low-level embedded device activity their system monitors API-layer flows on the IoT platform's cloud backend. Tian et al. present a block-layer auditing framework for portable USB storage that can be used to diagnose integrity violations [107]. Their embedded device emulates a USB flash drive, but does not consider syscall auditing of RT applications. Wu et al. present a network-layer auditing platform that captures the temporal properties of network flows and can thus detect temporal interference [114]. Whereas their system uses auditing to diagnose performance problems in networks, the presented study considers the performance problems created by auditing within RT applications. Zeno [115] provides event tracing that requires application instrumentation. Feather-Trace [26] events can be inserted in the syscall path to trace them, though such support does not currently exist, nor does it support ARM architectures. Our work directly enables efficient system-level auditing in RTS and incorporates the auditing system into the real-time task schedule.

10.3 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 [17, 19, 20, 32, 51, 59, 68, 75, 100, 102, 116, 122]. However, these approaches address the log storage overheads and not the voluminous event generation that is prohibitive to RTS auditing (§5.2). KCAL [76] and ProTracer [78] 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 [68]), but it is unclear whether these assumptions hold for real-time cyber-physical environments, *e.g.,* KCAL or ProTracer would reduce multiple identical read syscalls to a single entry. However, a large number of extra reads can cause catastrophic deadline misses. Forensic reduction in RTS, therefore, needs to be cognizant of the characteristics of RTS or valuable information can be lost. Our approach to template generation in *Ellipsis* shares similarities with the notion of *execution partitioning* of log activity [52, 53, 65, 67, 77], which decomposes long-lived applications into autonomous units of work to reduce false dependencies in forensic investigations. Unlike past systems, however, our approach requires no instrumentation to facilitate. Further, leveraging the well-formed nature of real-time tasks ensures the correctness of our execution units *i.e.,* templates. To our knowledge, this work is the first to address the need for forensic reduction of system logs in RTS.

10.4 RTS Security

RTS are an especially attractive target for malicious activity due to the potential to cause physical harm, as demonstrated by remotely disabling the engine of a car while it was running on the highway [84]. Prior works have analyzed the need and complexity of security in RTS [11, 29, 30, 61, 80, 95]. Security in RTS often involves a trade-off between system safety and functionality [11]. RTS being computing systems suffer from all the vulnerabilities of any components that they use, like sensors, operating systems, networking *etc.* There exists an additional class of attacks that impacts RTS by disrupting the execution just enough to make tasks miss deadlines [33]. In this work, we do not focus on a specific attack type but rather all attacks that leave identifiable information in logs as discussed in Section 4. Many solutions have been proposed to bolster the security of RTS. These solutions exploit the specific nature of RTS which can include predictable repetition of execution [118], memory access patterns [119] or static timing analysis [123]

to detect unexpected malicious attacks. However, due to their reliance on repetitive behavior, these techniques are primarily suitable for simpler RTS which use a small set of known tasks. Sporadic tasks that activate at random times in response to external stimuli further limit the utility of these techniques. In a survey of intrusion detection techniques for cyber-physical systems various detection techniques and the auditing materials are discussed [82]. We focus on Audit Log collection and usage which enables provenance techniques to investigate and detect intrusions. These techniques do not depend on the specific nature of real-time systems. They are known to be generally applicable as discussed in Section 4. Thus Audit logs and Provenance-based techniques can help secure RTS and CPS of all sizes and complexity.

10.5 Data Compression

Prius [100] proposed a trace compression technique for extremely resource-constrained devices like microsensor motes. Prius and *Ellipsis* share some of the constraints and goals, like limited resources and log reduction. However, key differences exist. *Ellipsis* achieves log reduction by filtering and coalescing events at the point of event generation, while Prius reduces an existing log. As shown in Section 5.2, audit events can be lost unless the events are reduced at the point of event creation itself. *Ellipsis* is complimentary to any data compression technique, *e.g.,* logs from Table 6 Index 1 could be reduced further by using gzip, from 7.6 MB to 237 KB for Linux Audit log, from 191 KB to 13 KB for *Ellipsis* log. The final point of difference is that *Ellipsis* removes unnecessary information in perpetuity not just for cold storage and transmission. Although reconstruction is possible (§4.2), the removed information has a high probability of being benign. The removal of extraneous information simplifies forensic analysis/investigation.

11 CONCLUSION

This work presents *Ellipsis*, a security auditing system designed for Real-Time Systems (RTS). *Ellipsis* is built to satisfy the unique requirements of RTS, which it does by leveraging the unique properties of special-purpose design, predictable execution, and extensive analyses before deployment. The successful application-aware co-design of *Ellipsis* and the increasing need for securing RTS motivates the adaptation of other general-purpose security tools for RTS.

ACKNOWLEDGMENTS

The material presented in this paper is based on work supported by the Office of Naval Research (ONR) under grant number N00014-17-1-2889, National Aeronautics and Space Administration (NASA) under grant number 80NSSC22M0070, and the National Science Foundation (NSF) under grant numbers CNS 1750024, CNS 1932529, CNS 1955228, CNS 2055127, CNS 2152768, and CNS 2246937. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] 2018. System Auditing. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security_guide/chap-system_auditing
[2] 2019. Raspberry Pi 4 Model B. https://www.raspberrypi.org/products/raspberry-pi-4-model-b/
[3] 2019. Raspberry Pi Linux 4.19 Preempt RT. https://github.com/raspberrypi/linux/tree/rpi-4.19.y-rt
[4] 2020. auditctl. https://manpages.debian.org/buster/auditd/auditctl.8.en.html
[5] 2021. Motion. https://motion-project.github.io/
[6] 2021. Navio2 board. https://navio2.emlid.com/
[7] 2023. FreeRTOS: Real-time operating system for microcontrollers. https://freertos.org/
[8] RTCA (Firm). SC 167. 1992. *Software considerations in airborne systems and equipment certification.* RTCA, Incorporated.
[9] Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert Ian Davis. 2020. An empirical survey-based study into industry practice in real-time systems. In *2020 IEEE Real-Time Systems Symposium (Proceedings).* IEEE.

[10] Azza Allouch, Omar Cheikhrouhou, Anis Koubâa, Mohamed Khalgui, and Tarek Abbes. 2019. MAVSec: Securing the MAVLink protocol for ardupilot/PX4 unmanned aerial systems. In *2019 15th International Wireless Communications & Mobile Computing Conference (IWCMC)*. IEEE.

[11] R. Altawy and A. M. Youssef. 2016. Security Tradeoffs in Cyber Physical Systems: A Case Study Survey on Implantable Medical Devices. *IEEE Access* 4 (2016), 959–979. https://doi.org/10.1109/ACCESS.2016.2521727

[12] Mike Anderson. 2020. SECURING EMBEDDED LINUX. https://elinux.org/images/5/54/Manderson4.pdf

[13] ArduCam. 2009. Arducam 5MP OV5647 1080p Mini Camera Module for Raspberry Pi 4/3B+/3. https://www.arducam.com/product/arducam-ov5647-standard-raspberry-pi-camera-b0033/. [Online; accessed 1-Nov-2022].

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

[15] Ayoosh Bansal, Anant Kandikuppa, Chien-Ying Chen, Monowar Hasan, Adam Bates, and Sibin Mohan. 2022. Towards Efficient Auditing for Real-Time Systems. In *European Symposium on Research in Computer Security*. Springer, 614–634.

[16] Sanjoy Baruah. 2007. Techniques for multiprocessor global schedulability analysis. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. IEEE, 119–128.

[17] Adam Bates, Kevin R. B. Butler, and Thomas Moyer. 2015. 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* (Edinburgh, Scotland) *(TaPP'15)*.

[18] Adam Bates, Dave Tian, Kevin R.B. Butler, and Thomas Moyer. 2015. Trustworthy Whole-System Provenance for the Linux Kernel. In *Proceedings of 24th USENIX Security Symposium* (Washington, D.C.).

[19] Adam Bates, Dave Tian, Grant Hernandez, Thomas Moyer, Kevin R.B. Butler, and Trent Jaeger. 2017. Taming the Costs of Trustworthy Provenance through Policy Reduction. *ACM Trans. on Internet Technology* 17, 4 (sep 2017), 34:1–34:21.

[20] Y. Ben, Y. Han, N. Cai, W. An, and Z. Xu. 2018. T-Tracker: Compressing System Audit Log by Taint Tracking. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. 1–9. https://doi.org/10.1109/PADSW.2018.8645035

[21] He Bin and Amahah Justice. 2009. The design of an unmanned aerial vehicle based on the ArduPilot. *Indian Journal of Science and Technology* 2, 4 (2009), 12–15.

[22] Enrico Bini and Giorgio C Buttazzo. 2005. Measuring the performance of schedulability tests. *Real-Time Systems* 30, 1-2 (2005), 129–154.

[23] Klaus Böhm, Tibor Kubjatko, Daniel Paula, and Hans-Georg Schweiger. 2020. New developments on EDR (Event Data Recorder) for automated vehicles. *Open Engineering* 10, 1 (2020), 140–146.

[24] Matteo Bordin, Cyrille Comar, Tristan Gingold, Jérôme Guitton, Olivier Hainque, Thomas Quinot, Julien Delange, Jérôme Hugues, and Laurent Pautet. 2009. Couverture: an innovative open framework for coverage analysis of safety critical applications. *Ada User Journal* 30, 4 (2009).

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

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

[27] Claire Burguiere and Christine Rochange. 2006. History-based schemes and implicit path enumeration. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[28] Carbon Black. 2018. Global Incident Response Threat Report. https://www.carbonblack.com/global-incident-response-threat-report/november-2018/. Last accessed 04-20-2019.

[29] Alvaro Cardenas, Saurabh Amin, Bruno Sinopoli, Annarita Giani, Adrian Perrig, Shankar Sastry, et al. 2009. Challenges for securing cyber physical systems. In *Workshop on future directions in cyber-physical systems security*, Vol. 5.

[30] A. A. Cardenas, S. Amin, and S. Sastry. 2008. Secure Control: Towards Survivable Cyber-Physical Systems. In *2008 The 28th International Conference on Distributed Computing Systems Workshops*. 495–500. https://doi.org/10.1109/ICDCS.Workshops.2008.40

[31] António Casimiro, Pedro Martins, and Paulo Verissimo. 2000. How to build a timely computing base using real-time linux. In *2000 IEEE International Workshop on Factory Communication Systems. Proceedings (Cat. No. 00TH8531)*. IEEE, 127–134.

[32] Chen Chen, Harshal Tushar Lehri, Lay Kuan Loh, Anupam Alur, Limin Jia, Boon Thau Loo, and Wenchao Zhou. 2017. Distributed provenance compression. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 203–218.

[33] Chien-Ying Chen, Amiremad Ghassami, Stefan Nagy, Man-Ki Yoon, Sibin Mohan, Negar Kiyavash, Rakesh B Bobba, and Rodolfo Pellizzoni. 2015. *Schedule-based side-channel attack in fixed-priority real-time systems*. Technical Report.

[34] Mei-Hwa Chen, Michael R Lyu, and W Eric Wong. 1996. An empirical study of the correlation between code coverage and reliability estimation. In *Proceedings of the 3rd International Software Metrics Symposium*. IEEE, 133–141.

[35] M-H Chen, Michael R Lyu, and W Eric Wong. 2001. Effect of code coverage on software reliability measurement. *IEEE Transactions on reliability* 50, 2 (2001), 165–170.

[36] Rodrigo Coelho, Gerhard Fohler, and Jean-Luc Scharbarg. 2017. Upper bound computation for buffer backlog on AFDX networks with multiple priority virtual links. In *Proceedings of the Symposium on Applied Computing*. 586–593.

[37] Miguel Correia, Paulo Veríssimo, and Nuno Ferreira Neves. 2002. The design of a COTS real-time distributed security kernel. In *European Dependable Computing Conference*. Springer, 234–252.

[38] Casey Crane. 2020. Automotive Cyber Security: A Crash Course on Protecting Cars Against Hackers. https://www.thesslstore.com/blog/automotive-cyber-security-a-crash-course-on-protecting-cars-against-hackers/

[39] Robert Day and Michael Slonosky. 2020. Securing connected embedded devices using built-in RTOS security. http://mil-embedded.com/articles/securing-connected-embedded-devices-using-built-in-rtos-security/

[40] Fabio Del Frate, Praerit Garg, Aditya P Mathur, and Alberto Pasquini. 1995. On the correlation between code coverage and software reliability. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95.* IEEE, 124–132.

[41] Department of Homeland Security. 2020. Cyber Physical Systems Security. https://www.dhs.gov/science-and-technology/cpssec

[42] Stephanie Forrest, Steven Hofmeyr, and Anil Somayaji. 2008. The evolution of system-call monitoring. In *2008 annual computer security applications conference (acsac).* IEEE, 418–430.

[43] Peng Gao, Xusheng Xiao, Ding Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, Chung Hwan Kim, Sanjeev R. Kulkarni, and Prateek Mittal. 2018. SAQL: A Stream-based Query System for Real-Time Abnormal System Behavior Detection. In *27th USENIX Security Symposium (USENIX Security 18).* USENIX Association, Baltimore, MD, 639–656. https://www.usenix.org/conference/usenixsecurity18/presentation/gao-peng

[44] Ashish Gehani and Dawood Tariq. 2012. SPADE: Support for Provenance Auditing in Distributed Environments. In *Proceedings of the 13th International Middleware Conference* (Montreal, Quebec, Canada) *(Middleware '12).*

[45] Golsana Ghaemi, Dharmesh Tarapore, and Renato Mancuso. 2021. Governing with Insights: Towards Profile-Driven Cache Management of Black-Box Applications. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021).* Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[46] Nan Guan, Meiling Han, Chuancai Gu, Qingxu Deng, and Wang Yi. 2015. 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, 11–20.

[47] Levent Gurgen, Ozan Gunalp, Yazid Benazzouz, and Mathieu Gallissot. 2013. Self-aware cyber-physical systems and applications in smart buildings and cities. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE, 1149–1154.

[48] Jan Gustafsson and Andreas Ermedahl. 2007. Experiences from applying WCET analysis in industrial settings. In *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07).* IEEE, 382–392.

[49] Mounir Hahad. 2020. IoT proliferation and widespread 5G: A perfect botnet storm. https://www.scmagazine.com/home/opinion/executive-insight/iot-proliferation-and-widespread-5g-a-perfect-botnet-storm/

[50] Xueyan Han, Thomas Pasqueir, Adam Bates, James Mickens, and Margo Seltzer. 2020. Unicorn: Runtime Provenance-Based Detector for Advanced Persistent Threats. In *27th ISOC Network and Distributed System Security Symposium (NDSS'20).*

[51] Wajih Ul Hassan, Nuraini Aguse, Mark Lemay, Thomas Moyer, and Adam Bates. 2018. Towards Scalable Cluster Auditing through Grammatical Inference over Provenance Graphs. In *Proceedings of the 25th ISOC Network and Distributed System Security Symposium.* San Diego, CA, USA.

[52] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. 2019. NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage. In *26th ISOC Network and Distributed System Security Symposium (NDSS'19).*

[53] Wajih Ul Hassan, Mohammad Noureddine, Pubali Datta, and Adam Bates. 2020. OmegaLog: High-Fidelity Attack Investigation via Transparent Multi-layer Log Analysis. In *27th ISOC Network and Distributed System Security Symposium (NDSS'20).*

[54] Les Hatton. 2004. Safer language subsets: an overview and a case history, MISRA C. *Information and Software Technology* 46, 7 (2004), 465–472.

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

[56] Heike Hofmann, Karen Kafadar, and Hadley Wickham. 2011. *Letter-value plots: Boxplots for large data.* Technical Report. had.co.nz.

[57] Steven A Hofmeyr, Stephanie Forrest, and Anil Somayaji. 1998. Intrusion detection using sequences of system calls. *Journal of computer security* 6, 3 (1998), 151–180.

[58] Md Nahid Hossain, Sadegh M. Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R. Sekar, Scott Stoller, and V.N. Venkatakrishnan. 2017. SLEUTH: Real-time Attack Scenario Reconstruction from COTS Audit Data. In *26th USENIX Security Symposium (USENIX Security 17).* USENIX Association, Vancouver, BC, 487–504. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/hossain

[59] Md Nahid Hossain, Junao Wang, R. Sekar, and Scott D. Stoller. 2018. Dependence-preserving Data Compaction for Scalable Forensic Analysis. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18).* USENIX Association, Berkeley, CA, USA, 1723–1740. http://dl.acm.org/citation.cfm?id=3277203.3277331

[60] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2019. Code coverage at Google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 955–963.

[61] Md E. Karim and Vir V. Phoha. 2014. Cyber-physical Systems Security. In *Applied Cyber-Physical Systems*, Sang C. Suh, U. John Tanik, John N. Carbone, and Abdullah Eroglu (Eds.). Springer New York, New York, NY, 75–83.

[62] Samuel T. King and Peter M. Chen. 2003. Backtracking Intrusions. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03).* ACM, New York, NY, USA, 223–236. https://doi.org/10.1145/945445.945467

[63] KaiGai Kohei. 2020. Recent security features and issues in embedded systems. https://elinux.org/images/e/e2/ELC2008_KaiGai.pdf

[64] Sascha Konrad and Betty HC Cheng. 2005. Real-time specification patterns. In *Proceedings of the 27th international conference on Software engineering.* 372–381.

[65] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela Ciocarlie, Ashish Gehani, and Vinod Yegneswaran. 2018. MCI: Modeling-based Causality Inference in Audit Logging for Attack Investigation. In *Proc. of the 25th Network and Distributed System Security Symposium (NDSS'18).*

[66] Insup Lee, Oleg Sokolsky, Sanjian Chen, John Hatcliff, Eunkyoung Jee, BaekGyu Kim, Andrew King, Margaret Mullen-Fortino, Soojin Park, Alexander Roederer, et al. 2011. Challenges and research directions in medical cyber–physical systems. *Proc. IEEE* 100, 1 (2011), 75–90.

[67] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition. In *Proceedings of NDSS '13* (San Diego, CA).

[68] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. LogGC: Garbage Collecting Audit Log. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security* (Berlin, Germany) *(CCS '13)*. ACM, New York, NY, USA, 1005–1016. https://doi.org/10.1145/2508859.2516731

[69] Yau-Tsun Steven Li and Sharad Malik. 1995. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*. 88–98.

[70] Linux Kernel Organization, Inc. 2020. CPU frequency and voltage scaling code in the Linux kernel. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt

[71] C. L. Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (Jan. 1973), 46–61. https://doi.org/10.1145/321738.321743

[72] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. 2018. Towards a Timely Causality Analysis for Enterprise Security. In *Proceedings of the 25th ISOC Network and Distributed System Security Symposium (NDSS'18)*. San Diego, CA, USA.

[73] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. 2018. Towards a Timely Causality Analysis for Enterprise Security. In *NDSS*.

[74] Sadegh M. Milajerdi, Birhanu Eshete, Rigel Gjomemo, and Venkat N. Venkatakrishnan. 2018. ProPatrol: Attack Investigation via Extracted High-Level Tasks. In *Information Systems Security*, Vinod Ganapathy, Trent Jaeger, and R.K. Shyamasundar (Eds.). Springer International Publishing, Cham, 107–126.

[75] Shiqing Ma, Kyu Hyung Lee, Chung Hwan Kim, Junghwan Rhee, Xiangyu Zhang, and Dongyan Xu. 2015. Accurate, Low Cost and Instrumentation-Free Security Audit Logging for Windows. In *Proceedings of the 31st Annual Computer Security Applications Conference* (Los Angeles, CA, USA) *(ACSAC 2015)*. ACM, New York, NY, USA, 401–410. https://doi.org/10.1145/2818000.2818039

[76] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. 2018. Kernel-Supported Cost-Effective Audit Logging for Causality Tracking. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 241–254. https://www.usenix.org/conference/atc18/presentation/ma-shiqing

[77] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2017. MPI: Multiple Perspective Attack Investigation with Semantic Aware Execution Partitioning. In *26th USENIX Security Symposium*.

[78] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In *Proceedings of NDSS '16* (San Diego, CA).

[79] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In *NDSS*.

[80] D. W. McKee, S. J. Clement, J. Almutairi, and J. Xu. 2017. Massive-Scale Automation in Cyber-Physical Systems: Vision amp;amp; Challenges. In *2017 IEEE 13th International Symposium on Autonomous Decentralized System (ISADS)*. 5–11. https://doi.org/10.1109/ISADS.2017.56

[81] S. Momeni Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan. 2019. HOLMES: Real-Time APT Detection through Correlation of Suspicious Information Flows. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA. https://doi.org/10.1109/SP.2019.00026

[82] Robert Mitchell and Ing-Ray Chen. 2014. A Survey of Intrusion Detection Techniques for Cyber-physical Systems. *ACM Comput. Surv.* 46, 4, Article 55 (March 2014), 29 pages. https://doi.org/10.1145/2542049

[83] László Monostori, Botond Kádár, Thomas Bauernhansl, Shinsuke Kondoh, S Kumara, Gunther Reinhart, Olaf Sauer, Gunther Schuh, Wilfried Sihn, and Kenichi Ueda. 2016. Cyber-physical systems in manufacturing. *Cirp Annals* 65, 2 (2016), 621–641.

[84] PBS NewsHour. 2015. Hacking researchers kill a car engine on the highway to send a message to automakers. https://www.pbs.org/newshour/show/hacking-researchers-kill-car-engine-highway-send-message-automakers

[85] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher W. Fletcher, Andrew Miller, and Dave Tian. 2020. Custos: Practical Tamper-Evident Auditing of Operating Systems Using Trusted Execution. In *27th ISOC Network and Distributed System Security Symposium*.

[86] Riccardo Paccagnella, Kevin Liao, Dave Tian, and Adam Bates. 2020. Logging to the Danger Zone: Race Condition Attacks and Defenses on System Audit Frameworks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1551–1574.

[87] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. 2017. Practical Whole-system Provenance Capture. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. ACM, New York, NY, USA, 405–418. https://doi.org/10.1145/3127479.3129249

[88] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, , and Margo Seltzer. 2018. Runtime Analysis of Whole-System Provenance. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[89] T. F. J. . Pasquier, J. Singh, D. Eyers, and J. Bacon. 2017. Camflow: Managed Data-Sharing for Cloud Services. *IEEE Transactions on Cloud Computing* 5, 3 (July 2017), 472–484. https://doi.org/10.1109/TCC.2015.2489211

[90] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, Shiqing Ma, Fei Wang, Zhiwei Zhang, Luo Si, Xiangyu Zhang, and Dongyan Xu. 2016. HERCULE: Attack Story Reconstruction via Community Discovery on Correlated Log Graph. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications* (Los Angeles, California, USA) *(ACSAC '16)*. ACM, New York, NY, USA, 583–595. https://doi.org/10.1145/2991079.2991122

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

[92] Peter Puschner and Alan Burns. 2002. Writing temporally predictable code. In *Proceedings of the Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems.(WORDS 2002)*. IEEE, 85–91.

[93] Ragunathan Rajkumar, Insup Lee, Lui Sha, and John Stankovic. 2010. Cyber-physical systems: the next computing revolution. In *Design Automation Conference*. IEEE, 731–736.

[94] Ragunathan Rajkumar, Lui Sha, and John P Lehoczky. 1988. Real-time synchronization protocols for multiprocessors. In *Proceedings. Real-Time Systems Symposium*. 259–260.

[95] A. Sadeghi, C. Wachsmann, and M. Waidner. 2015. Security and privacy challenges in industrial Internet of Things. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2744769.2747942.

[96] Daniel Sandell, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. 2004. Static timing analysis of real-time operating system code. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 146–160.

[97] Martin Schoeberl. 2006. Real-time garbage collection for Java. In *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*. IEEE, 9–pp.

[98] David Shepherd. 2020. Industry 4.0: the development of unique cybersecurity. https://www.manufacturingglobal.com/technology/industry-40-development-unique-cybersecurity

[99] Youcheng Sun and Marco Di Natale. 2018. Assessing the pessimism of current multicore global fixed-priority schedulability analysis. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 575–583.

[100] Vinaitheerthan Sundaram, Patrick Eugster, and Xiangyu Zhang. 2012. Prius: Generic hybrid trace compression for wireless sensor networks. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*. 183–196.

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

[102] Yutao Tang, Ding Li, Zhichun Li, Mu Zhang, Kangkook Jee, Xusheng Xiao, Zhenyu Wu, Junghwan Rhee, Fengyuan Xu, and Qun Li. 2018. NodeMerge: Template Based Efficient Data Reduction For Big-Data Causality Analysis. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. ACM, New York, NY, USA, 1324–1337. https://doi.org/10.1145/3243734.3243763

[103] The Linux Foundation. 2018. Real-Time Linux. https://wiki.linuxfoundation.org/realtime/start

[104] The Linux Foundation. 2022. RT-Tests. https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/rt-tests

[105] The MISRA Consortium Limited. 2021. MISRA. https://www.misra.org.uk/

[106] The MITRE Corporation. 2018. Medical Device Cybersecurity. https://www.mitre.org/sites/default/files/publications/pr-18-1550-Medical-Device-Cybersecurity-Playbook.pdf

[107] Dave (Jing) Tian, Adam Bates, Kevin R. B. Butler, and Raju Rangaswami. 2016. ProvUSB: Block-level Provenance-Based Data Protection for USB Storage Devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, NY, USA.

[108] Paulo Veríssimo and António Casimiro. 2002. The timely computing base model and architecture. *IEEE Trans. Comput.* 51, 8 (2002), 916–930.

[109] Paulo Veríssimo, António Casimiro, and Christof Fetzer. 2000. The timely computing base: Timely actions in the presence of uncertain timeliness. In *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*. IEEE, 533–542.

[110] Liuping Wang. 2020. *PID control system design and automatic tuning using MATLAB/Simulink*. John Wiley & Sons.

[111] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. 2017. Fear and Logging in the Internet of Things. In *Proceedings of the 25th ISOC Network and Distributed System Security Symposium (NDSS'18)*.

[112] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Jungwhan Rhee, Zhengzhang Zhen, Wei Cheng, Carl A. Gunter, and Haifeng chen. 2020. You Are What You Do: Hunting Stealthy Malware via Data Provenance Analysis. In *27th ISOC Network and Distributed System Security Symposium (NDSS'20)*.

[113] W Eric Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. 2007. Effective fault localization using code coverage. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, Vol. 1. IEEE, 449–456.

[114] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. 2019. Zeno: Diagnosing Performance Problems with Temporal Provenance. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 395–420. https://www.usenix.org/conference/nsdi19/presentation/wu

[115] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. 2019. Zeno: Diagnosing Performance Problems with Temporal Provenance.. In *NSDI*. 395–420.

[116] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. 2016. High Fidelity Data Reduction for Big Data Security Dependency Analyses. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. ACM, New York, NY, USA, 504–516. https://doi.org/10.1145/2976749.2978378

[117] Carter Yagemann, Mohammad Noureddine, Wajih Ul Hassan, Simon Chung, Adam Bates, and Wenke Lee. 2021. Validating the Integrity of Audit Logs Against Execution Repartitioning Attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*.

[118] M. Yoon, S. Mohan, J. Choi, J. Kim, and L. Sha. 2013. SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 21–32. https://doi.org/10.1109/RTAS.2013.6531076

[119] M. Yoon, S. Mohan, J. Choi, and L. Sha. 2015. Memory Heat Map: Anomaly detection in real-time embedded systems using memory behavior. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2744769.2744869

[120] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Mihai Christodorescu, and Lui Sha. 2017. 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*.

[121] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Jung-Eun Kim, and Lui Sha. 2013. SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 21–32.

[122] Tiantian Zhu, Jiayu Wang, Linqi Ruan, Chunlin Xiong, Jinkai Yu, Yaosheng Li, Yan Chen, Mingqi Lv, and Tieming Chen. 2021. General, efficient, and real-time data compaction strategy for apt forensic analysis. *IEEE Transactions on Information Forensics and Security* 16 (2021), 3312–3325.

[123] Christopher Zimmer, Balasubramanya Bhat, Frank Mueller, and Sibin Mohan. 2010. Time-based Intrusion Detection in Cyber-physical Systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems* (Stockholm, Sweden) *(ICCPS '10)*. ACM, New York, NY, USA, 109–118. https://doi.org/10.1145/1795194.1795210

A TEMPLATES FOR ARDUPILOT

Table 7. Columns of this table describe the three template files 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

ArduPilot yielded 3 templates. System call numbers and their corresponding arguments, a0 - a4, were extracted from the audit logs. `read`, `write`, `pread64` have syscall numbers 3,4 and 180 respectively. Argument values of -1 and temporal constraint values of 0 denote that these arguments are ignored. **4:3:-1:1:-1** then indicates a *write syscall* with *a0 as 3, a2 as 1. a1 and a3* are *not forensically relevant*. Table 7 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)
```

Some fields in *Ellipsis* log are distinct from standard Linux Audit logs; *(i) template*: the name of the template, the first line of a template file *e.g.,arducopter, ap-rcin, ap-spi-0*; *(ii) rep*: the number of consecutive repetitions of the template this entry represents when using *Ellipsis-HP*; *(iii) stime*: timestamp of the first syscall in this reduced sequence, the unit is nanoseconds; *(iv) etime*: timestamp of the last syscall in this reduced sequence, the unit is nanoseconds.

Received 5 March 2023; revised 16 July 2023; accepted 12 September 2023