SoK: Security in Real-Time Systems

MONOWAR HASAN, Washington State University, Pullman, USA
ASHISH KASHINATH, University of Illinois at Urbana-Champaign, Urbana, USA
CHIEN-YING CHEN, University of Illinois at Urbana-Champaign, Urbana, USA
SIBIN MOHAN, The George Washington University, Washington, USA

Security is an increasing concern for real-time systems (RTS). Over the last decade or so, researchers have demonstrated attacks and defenses aimed at such systems. In this article, we *identify*, *classify* and *measure* the effectiveness of the security research in this domain. We provide a high-level summary [*identification*] and a taxonomy [*classification*] of this existing body of work. Furthermore, we carry out an in-depth analysis [*measurement*] of scheduler-based security techniques — the most common class of real-time security mechanisms. For this purpose, we developed a common metric, "*attacker's burden*", used to measure the effectiveness of (existing as well as future) scheduler-based real-time security measures. This metric, built on the concept of "work factor" [1], is adapted for, and normalized across, various scheduler-based real-time security techniques.

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

Additional Key Words and Phrases: Embedded systems, cyber-physical systems, scheduling

ACM Reference Format:
Monowar Hasan, Ashish Kashinath, Chien-Ying Chen, and Sibin Mohan. 2024. SoK: Security in Real-Time Systems. *ACM Comput. Surv.* 56, 9, Article 218 (April 2024), 31 pages. https://doi.org/10.1145/3649499

1 INTRODUCTION

Real-time systems (RTS), such as avionics, nuclear power plants, automobiles, space vehicles, power generation and distribution systems, medical devices, industrial robots, and the like have been in existence for decades. Most of these systems have *safety-critical* properties, i.e., any problems could result in significant harm to humans, the system, or even the environment. Consider the case where a car's airbag, a real-time system with stringent timing constraints, fails to deploy *in time*— such failures can have disastrous results. Despite their importance, security has rarely been a consideration in the design of RTS, mainly due to beliefs such as: (a) real-time systems

A part of this work was conducted when M. Hasan and S. Mohan were affiliated with University of Illinois at Urbana-Champaign (UIUC). The material in this paper is based upon work supported in part by the U.S. National Science Foundation (NSF) under grant NSF CPS 2246937 and NSF CNS 2312006. Any findings, opinions, recommendations or conclusions expressed in the paper are those of the authors and do not necessarily reflect the views of sponsors.
Authors' addresses: M. Hasan, Washington State University, 355 NE Spokane Street, Pullman, WA 99164; e-mail: monowar.hasan@wsu.edu; A. Kashinath and C.-Y. Chen, University of Illinois at Urbana-Champaign, 201 North Goodwin Avenue, Urbana, IL 61801; e-mails: ashishk3@illinois.edu, cchen140@illinois.edu; S. Mohan, George Washington University, 800 22nd St NW, Washington, DC 20052; e-mail: sibin.mohan@gwu.edu.

ACM 0360-0300/2024/04-ART218
https://doi.org/10.1145/3649499

lack inherent value to adversaries (*"why would anyone attack them?"*), (b) the prevalence of custom hardware/software/protocols will deter attackers (*"these protocols/hardware/software are secret and so arcane that no one can decipher them"*) and also (c) the lack of computing power and memory in these systems will throttle potential adversarial actions (*"what can they do even if they get in?"*). In addition, RTS has *stringent timing requirements* for ensuring their correct operation. For instance, a typical window for an airbag deployment (time between detection of collision and final airbag operations) is around 50–60 ms [2] (less than the time it takes to blink human eyes once!). Such timing constraints severely inhibit how security solutions can be added to real-time systems; for instance, the protection methods should not cause timing problems in RTS.

With the advent of newer domains such as autonomous vehicles, drones, remote monitoring and control and Internet-of-Things (IoT), RTS find themselves front and center in modern society. Since many RTS now use commodity-off-the-shelf (COTS) components and are often connected to each other or even the Internet, they expose additional attack surfaces, often overturning all of the aforementioned beliefs. In fact, there has been a significant uptick in attacks against systems with real-time properties over the past decade [3–7].

Security problems in RTS also differ when compared to general purpose systems. The requirement for tight timing guarantees brings up new vulnerabilities; for instance, if an attacker is able to introduce a minor delay (even a few milliseconds) into the critical pathway for the deployment of an airbag, then the passengers could be seriously injured or, worse, killed. In addition, RTS are typically designed with *safety* in mind, rather than security. Even in the presence of malicious actions, the prime focus of RTS designers would still be safe, i.e., ensure that the system or its operators do not come to harm.

Real-time security has received significant attention in recent years from both academia and industry, and now a significant body of work has started to appear in this domain. For instance, regular attack vectors are also finding use against RTS, e.g., the leakage of critical data [8] and even hostile actions due to lack of authentication [5, 6, 9]. In this article, we have *three major goals*:

(1) *surveying the knowledge* in this area so that researchers can gain an understanding of the domain, current solutions and challenges;
(2) *systemizing* this knowledge by establishing a high-level *taxonomy* of existing work and
(3) development of a common *metric* that can be used to compare and contrast solutions that are focused on scheduler-based techniques since they form the largest collection of works in this domain.

Systematization Approach. In this study, we investigate RTS security issues and summarize existing techniques (both from attack and defense perspectives) in a comprehensive manner. While there exists some limited survey (see Section 9 for details), to the best of our knowledge, there exists no prior comprehensive summary and taxonomy of real-time security research that (a) classify attack and defense mechanisms and (b) systematically compare real-time security solutions using a "unified metric". We have studied *over 250* papers from a variety of archived sources published in the last 27 years (1995–2022) and short-listed the related work. For instance, our search includes all major online archives *viz.,* ACM Digital Library, Google Scholar, IEEE Xplore, MDPI, ScienceDirect, Scopus, USENIX and Wiley Online Library. In addition, we manually parsed papers from major security, real-time, embedded systems and design automation conferences. We also crawled the related publications from the websites of the researchers we know who work in similar domains. Some of the major keywords used in our search include: "real-time systems", "security", "side-channel", "attacks", "timing analysis", "temporal guarantees", among others. Table 9 in Appendix A lists all the articles and their source communities. NOTE: We excluded articles that

are not directly connected to real-time security, for instance, security for broader cyber-physical/control systems, robotics, IoT/cloud/edge systems, and mobile devices.

Taxonomy. Our research identifies some well-defined categories and sub-categories (for attacks and defenses) that the majority of work in real-time security can be classified into. In this article, we present such a taxonomy (Figure 2) that can be easily used to identify and classify existing (and potentially future) work in RTS security.

Metric. A large class of security solutions in RTS are centered around resource management algorithms since they form the crux of RTS design. The **operating system (OS)** *scheduler* is considered to be the most important resource management algorithm and, hence, has received a lot of attention in terms of adding security, as is evident in Figure 2. Even with this diverse body of work, there exists no systematic framework or metric to analyze these systems or contrast them against each other.

In this article, we *present a metric*, named *attacker's burden*, inspired by *work factor* [1], a concept that captures the "cost" of circumventing a security mechanism with the resources available to an attacker. Our metric builds upon this concept by capturing the (increased) computational load on adversaries by translating it into something directly relevant to RTS: the ***time** that is available to attackers*. Unlike cryptographic algorithms where a direct correlation can be made between increased key size and the computational power required by adversaries, the computation of the attacker's burden in RTS security is not straightforward since, as shown in Section 6, even among the scheduler-driven techniques, there is significant diversity in how security mechanisms are designed. Hence, calculating the time available for would-be attackers is challenging. To demonstrate the use of this metric, we show how it can be computed for the various categories of scheduler-based security techniques in literature —thus providing an easy reference for comparing the state-of-the-art. We believe that this new metric, presented in Section 6, will allow designers to analyze and quantify the effect of their solutions and, in effect, improve the security guarantees in such systems.

In this article, we make the following contributions:

- A taxonomy of real-time security research (Figure 2) and a comprehensive study of related literature (see the list of articles in Table 9).
- A systematic review and qualitative comparison of various RTS security solutions, both from attack and defense perspectives (Section 4, Section 5 and Section 7).
- In-depth study of scheduler-level RTS security solutions (Section 5) and related analyses using our newly developed metric, "attacker's burden" (Section 6).

We start with a background on RTS (Section 2).

2 REAL-TIME SYSTEMS

RTS are defined by their strong timing requirements. They need to function correctly, but *within their predefined timing constraints*, often termed as a *deadlines* —recall the airbag example from earlier where the deployment has to complete with a few tens of milliseconds. Some of the common properties and assumptions related to RTS are as follows: (i) implemented as a system of periodic tasks, (ii) worst-case bounds are known for most loops as well as the critical pieces of code, (iii) no dynamically loaded or self-modifying code, (iv) recursion is either not used or statically bounded, (v) memory and processing power often limited, and (vi) stringent timing and safety requirements.

2.1 Architecture and System Development Model

In Figure 1, we present a high-level illustration of an RTS. Each real-time application in the system (called *task* task) represents a time-critical function and a collection of such tasks are hosted on

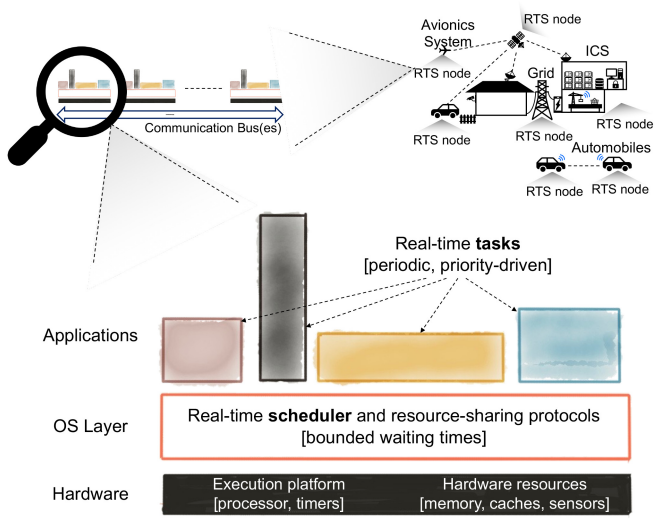


Fig. 1. Abstraction of a real-time node with common uses cases.

a hardware platform (mostly single-core systems). The concept of tasks in RTS can be trivially mapped with *processes* or *threads* in general-purpose OS. The scheduler in **real-time OS (RTOS)** uses timers and interrupt handlers to enforce timing guarantees at runtime. This ability of the scheduler to interrupt application processing at precise time instants is essential to ensure the "correctness" of the system. Access to shared platform resource (such as caches, buses, memory) is regulated using resource sharing protocols to ensure data consistency and bounds on waiting time so that deadlines can be met. The communication network in RTS is required to provide service with low jitters and meet end-to-end message deadlines for all messages.

2.2 Task and Scheduling Model

Tasks in RTS are generally characterized by their periods (inter-arrival times), constant, upper bounded execution times and temporal constraints (deadlines). Schedulability tests [10–12] are used to determine if all tasks in the system meet their respective deadlines. If this is the case, then the task set is deemed to be "schedulable" and the system, safe. In Listing 1 we present an abstraction of a real-time task. running on real-time Linux (RT_PREEMPT [13]). Line 4 specifies the priority and Line 11 performs the main task functionality. Timers are updated in Lines 14-19 for periodic invocations.

In real-time scheduling, task priorities can be either *fixed* or *dynamic*. Examples of commonly used fixed and dynamic priority algorithms are: (a) *rate monotonic (RM)*, where task priorities are assigned based on periods (i.e., shorter period implies higher priority) and (b) *earliest deadline first (EDF)*, where a job with shortest deadline is scheduled first [14]. Scheduling algorithms can be *(i) preemptive* (tasks can be preempted by higher priority tasks) or *(ii) non-preemptive* (when a task starts executing, it will not be preempted and execute until completion). A system is called mixed-criticality RTS if it has more that one criticality levels (say safety critical, mission critical, and low critical) [15].

3 TAXONOMY

One of the contributions of this article is the *identification and classification* of research in the area of real-time security. Figure 2 illustrates the various categories of RTS security research that

```
int main()
{
   struct timespec t; struct sched_param param;
   param.sched_priority = _PRIORITY_; // set priority

   /* enable real-time scheduling */
   sched_setscheduler(0, SCHED_FIFO, &param)
   clock_gettime(0, &t); // get current time
   /* main real-time loop */
   while(1) {
      main_task_function();  // do the stuff
      /* update timer (nanosecond and second fields)
         for next period */
      t.tv_nsec += _PERIOD_;
      while (ts->tv_nsec >= NANOSEC_PER_SEC) {
        ts->tv_nsec -= NANOSEC_PER_SEC; ts->tv_sec++;
      }
      /* wait for next period */
      clock_nanosleep(0, TIMER_ABSTIME, &t, NULL);
   }
   return 0;  // end of code (never reaches here)
}
```

Listing 1. Code abstraction of a periodic real-time task.

we have identified. We find that RTS security research can be broadly classified into *attack* and *defense* mechanisms. We categorize attacks into two classes: (i) reconnaissance (where an attacker passively infers system information and attempts to break confidentiality—this reconnaissance can be used to launch further attacks, see Section 4.1) and (ii) targeted attacks (where an adversary tampers with temporal constraints of other real-time tasks, Section 4.2).

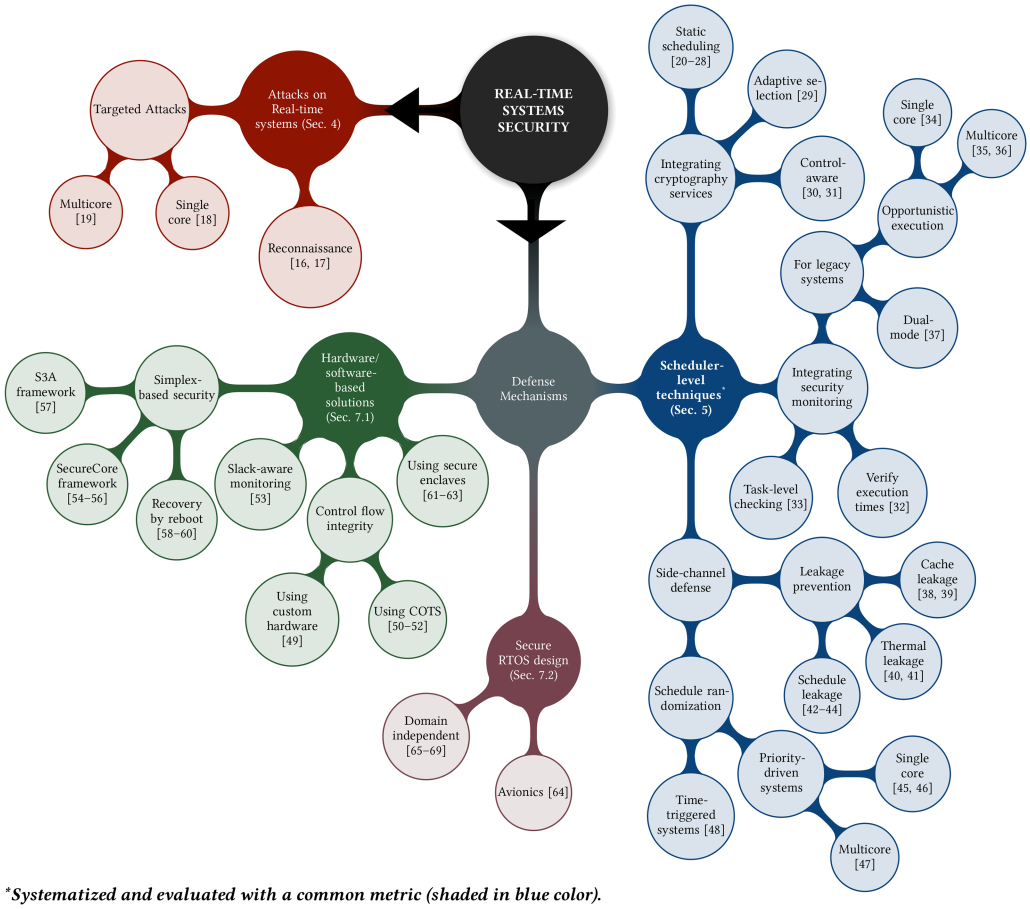
The literature on defense mechanisms can be divided into three major categories: (i) scheduler-based techniques (Section 5), (ii) hardware/software-based architectural solutions (Section 7.1) and (iii) research on secure RTOS design (Section 7.2). We further classify scheduler-based defenses, that has the most number of articles, into three categories: (a) integration of cryptography services (Section 5.1); (b) integration of monitoring techniques (Section 5.2), and (c) side-channel defense techniques (Section 5.4-Section 5.3).

Research on architectural solutions includes: (a) control flow monitoring (Section 7.1.1), (b) monitoring of task execution time by computing slack (Section 7.1.2), (c) techniques based on Simplex [16]—a real-time design framework to provide fault-tolerance (Section 7.1.3), and (d) techniques using off-the-shelf processor extensions such as ARM TrustZone [17] (Section 7.1.4).

4 ATTACKS ON RTS

We now start with a discussion on attack mechanisms that target RTS. We classified the attack methodologies on RTS based on the level of control over tasks in the system and the functional objective of the attack. An adversary can violate the integrity of the system via malicious code injections and/or by triggering a logic bomb that is otherwise silent/not detected during system design (attacks on *integrity* and/or *confidentiality*). Since many modern RTS communicate over unreliable mediums such as the Internet, the system is also vulnerable to network-based attacks. Threats to message communications are usually dealt with by integrating cryptographic protection mechanisms. In Section 5.1, we present solutions proposed in the literature to integrate cryptographic mechanisms.

Since many real-time applications are running on embedded platforms, they are vulnerable to DoS attacks (i.e., attacks on *availability*) due to inherent resource constraints (low memory/storage capabilities, limited computation resource, limited energy). For example, an attacker may perform system-level resource (e.g., CPU, disk, memory, I/O) exhaustion and force a critical task to miss



*Systematized and evaluated with a common metric (shaded in blue color).

Fig. 2. Taxonomy of various attacks and defense techniques proposed in the real-time literature.

the deadline due to resource unavailability. The DoS defense mechanisms developed for generic personal/enterprise computing platforms or embedded systems do not consider timing, safety, and resource limitations of RTS and are not easily adopted without significant modifications.

Other than trying to aggressively crash the system, the adversary may silently lodge itself in the system and extract sensitive information. Such side-channel attacks manipulate unexpected channels (e.g., cache usage [44], thermal traces [39] and timing behavior [40]) to acquire useful information from the victim. These channels are particularly effective for attacking RTS due to their deterministic behaviors. Side channel attacks break the confidentiality and could be used by the adversary to launch further attacks (that may jeopardize integrity and/or availability of the system). As we shall see in this article, researchers have found various side-channels for RTS (Section 4.1) and also proposed techniques to mitigate them (Section 5.4-Section 5.3).

We now present various attack techniques demonstrated by the researchers.

4.1 Reconnaissance

In order for many attacks to succeed, reconnaissance is typically one of the early steps in the process. It is in the interest of the attacker to stay undetected during this time to both, (a) collect necessary and sufficient information to enable their attacks and (b) not alert system operators

and their defensive actions. In the following, we illustrate two reconnaissance techniques in RTS identified by the researchers.

Chen et al. [44] developed techniques to reconstruct the task schedule of RTS so targeted attacks can be launched against critical tasks. In particular, they develop an algorithm (ScheduLeak) to deconstruct constituent jobs of the tasks from "busy periods" (i.e., a block of time when one or more tasks are executing). This allows the adversary to determine *what* tasks are running *when* by just observing the busy periods. The inferred, ordered, job set output by the algorithm allows the attacker to reconstruct the schedule (with up to 99% success rate if tasks have fixed execution times) and pinpoint the possible start time of any required victim task for the foreseeable future.

Liu et al. [45] showed that attackers can infer the engine speed of a vehicle by observing the real-time scheduling sequences on the engine control unit (ECU). They showed that the ECU schedule (consisting of the core engine task and other regular real-time tasks) and task periods can be obtained by observing the electromagnetic radiation using a near-field probe and hardware signal analyzer. The deduced period was then used to obtain the speed profile of the engine.

4.2 Targeted Attacks

We now present work on targeted attacks where adversaries tailor their attack strategy to leverage a specific RTS property.

4.2.1 Manipulating Control Tasks. For a priority-driven, periodic real-time application, changing the parameters (period, priority) of one task may alter the temporal properties (i.e., response time) and, hence, the safety of other tasks. Mahfouzi et al. [46] show that, by interfering with an entry task (say a low-priority, less-protected one that interacts with the outside world), an adversary can manipulate the response time of other, low-priority tasks (called *Butterfly attack*). This attack demonstrates how an adversary can destabilize single core systems. We now discuss timing perturbations of critical tasks from multicore context.

4.2.2 Cache-Level DoS Attacks. Many multicore processors use non-blocking caches [47] to support concurrent memory accesses from multiple cores. Bechte et al. [48] show that an adversary can cause significant timing influence to the real-time tasks by accessing shared caches. They show that shared cache blocking can occur in both out-of-order and in-order processors and can increase execution times significantly, e.g., up to 346X on a quad-core in-order architecture (Raspberry Pi). Recall from the airbag example, such attacks can delay the airbag deployment and threaten safety of the passengers.

4.3 Summary of Our Findings

While there exists some recent work on defensive techniques for RTS (see Section 5 and Section 7), there is considerably less focus on *attack* mechanisms. There is a lack of articles that demonstrate the techniques to extract important information and/or disrupt the normal operation of the system while still remaining undetected. We find that researchers study attack mechanisms from two contexts: (i) stealthy attacks (infer sensitive information by leveraging side-channels [44, 45]) and (ii) targeted attacks (increase in execution time of the victim tasks by modify tasks parameters such as periods [46] or blocking shared cache [48]). Out of four attack techniques we reviewed, three of them [44–46] explicitly consider single core, fixed-priority systems and only one mechanism (DoS attacks on shared caches) [48] is applicable to multicore platforms. Three of them [44, 45, 48] evaluate/demonstrate the attacks on real hardware/embedded platforms while one (Butterfly attack [46]) is evaluated using synthetic case studies.

Table 1. Threat Models Used in Literature

Problem Focus	Assumptions on Adversarial Capabilities
Integrating cryptography services [18–27]	The attacker has: (i) knowledge of the real-time tasks and intends to break confidentiality, authentication, and/or integrity of the existing tasks [18–20, 22–24, 26, 27]; *(ii)* access hardware to perform brute force attacks against symmetric cryptographic schemes [21]; *(iii)* knowledge of application vulnerabilities [25]
Control-aware data/message integrity [28, 29]	The attacker has: (i) knowledge of the schedule of control tasks [28]; (ii) access to low-level network messages and knowledge of when integrity checks (e.g., MAC) are performed [29]. The attacker wants to remain stealthy and (i) manipulate the system to an unsafe state [28]; (ii) inject false messages [29]
Monitor temporal variations [30]	No specific assumptions on attacker's capabilities; the adversary can leverage known vulnerabilities and execute malicious code
Task-level integrity checking [31]	Attacker can exploit buffer overflow vulnerabilities and launch ROP (return-oriented programming) attacks
Periodic monitoring for legacy RTS [32–35]	No specific assumptions on attacker's capabilities; model can be used for scenarios where periodic monitoring of system events is required
Leakage prevention by clearing shared cache [36, 37]	Vendor-based system development model; a compromised task (perhaps from a less trusted vendor) can snoop information from security sensitive (i.e., victim) tasks; no specific assumptions on how a task could be compromised
Robustness analysis of AES keys against differential power analysis attacks [38]	The adversary aims to obtain the AES key used in the system. The attacker (i) has physical access to the system; (ii) can accurately measure the power consumption; (iii) knows the periods of all tasks but does not know their actual execution times
Thermal leakage prevention [39]	Attacker (i) knows prior thermal profile of the task schedule; (ii) can have access to on-chip thermal sensors and obtain runtime measurements
Reduce determinism by schedule obfuscation [40–43]	The attacker can hijack one or more tasks in the system and wants to determine which task is running at any point in time (by observing execution traces)

5 SCHEDULER-LEVEL DEFENSES FOR RTS SECURITY

We now present scheduler-level defense mechanisms proposed in the literature (a total of 28 articles). These techniques (a) are software-based approaches (often integrated at design time), (b)can be applied by enforcing scheduler-level constraints and/or placing a hook within the scheduler, and (c) do not require any custom hardware and/or architectural support. We categorize these techniques into three major classes: (i) integrating cryptographic primitives (Section 5.1), (ii) integrating security monitoring techniques (Section 5.2) and (iii) side-channel defense mechanisms (Section 5.3 and Section 5.4). In Table 1, we summarize the threat models and assumptions considered by the authors. In particular, for different techniques we summarize the attacker's capabilities and knowledge of the system as well as the types of attacks the each of the schemes intend to prevent.

5.1 Integrating Cryptographic Operations

Integrating security into RTS that enables confidentiality, integrity and authentication increases the computational load and may adversely affect the timing constraints of existing time-critical tasks. We now discuss various techniques proposed in literature for incorporating (and optimizing) security services (see Table 2 for a summary).

5.1.1 Static Scheduling Techniques. Xie et al. [18] propose a modified version of EDF (called **EDF_OPT**) that aims to maximize the number of accepted tasks while providing the highest level of security services (such as SSL, authentication) possible to the accepted tasks. They further propose an algorithm (SASES) for finding feasible schedules that maximize confidentiality, integrity, and authentication requirements (as given by the designers) [19]. Lin et al. [20] propose a group-based security model where different security services for authentication/integrity (such as RC4, RC5, DES) are put into different groups. Kang et al. [21] propose a feedback-control scheme that considers system load and security. The above pieces of work consider independent, periodic tasks with same levels of criticality and are designed for single core systems. Similar ideas have been applied to (a) multicore platforms where tasks can be inter-dependent [22]; (b) energy-constrained systems [23–25], and (c) mixed-criticality systems [25, 26].

Table 2. Summary: Integration of Cryptographic Primitives

Reference	Task Model	Scheduling Policy*	Platform	Key Idea	Overhead/Limitation
Xie et al. [18]	Independent, aperiodic†, non-preemptive	EDF	Single core	Find optimal level of cryptography services (e.g., SSL, authentication) without impacting the deadlines	Security models are not well defined, does not provide hard timing guarantees
Xie et al. [19]	Independent, periodic, preemptive	EDF	Single core	Obtain a feasible schedule that maximizes confidentiality, integrity and authentication (CIA) requirements	Hard to quantify different security levels (given by CIA requirements)
Lin et al. [20]	Independent, periodic, preemptive	EDF	Single core	Group multiple security services (e.g., authentication, data integrity) together and find the best combination while retaining schedulability guarantees	Does not distinguish different services (i.e., they use same weights)
Qiu et al. [22]	Dependent, periodic, non-preemptive	N/A	Multicore	Find security services for set of tasks with precedence constraints (i.e., dependency among task executions)	Does not provide hard timing guarantees
Jiang et al. [23, 24]	Independent, periodic, preemptive	EDF, RM	Single core	Minimizing security risk (defined based on cryptography services) under a predefined energy budget	Vague definition of "security risk"
Zhang et al. [26]	Independent, periodic, preemptive	EDF, RM	Single core	Minimize vulnerability (defined as a function of priority and the number of rounds used by a cryptographic scheme) subject to real-time requirements	No systematic way of defining "vulnerability" of an encryption algorithm
Zhang et al. [25]	Dependent, periodic, non-preemptive	Proposed by the authors	Multicore	Assign security-critical tasks to cores such that both energy budget and real-time constraints are satisfied	Vague definition of security levels
Kang et al. [21]	Independent, periodic/aperiodic, preemptive	N/A	Single core	Provide a technique to adaptively control the utilization, execution time and strength of protection (a function of crytographic key length)	Does not provide hard timing guarantees for real-time tasks
Saadatmand et al. [27]	Independent, periodic/aperiodic, preemptive	N/A	Multicore	Dynamically switch between different encryption algorithms at runtime from a precomputed table (where different encryption algorithms are sorted based on their execution times)	Runtime overheads due to extra lookups
Lesi et al. [28, 29]	Independent, periodic, non-preemptive	EDF	Single core	Prevent man-in-the-middle attacks while preserving control performance	May not detect stealthy attacks if an adversary gradually degrades control performance

*EDF: Earliest deadline first; RM: rate monotonic. We refer this column as "N/A" if the authors do not explicitly consider any scheduling algorithm and/or the proposed scheme is independent of a particular scheduling technique.
†Task arrival follows a Poisson distribution.

5.1.2 Adaptive Scheduling. The aforementioned schemes focus on design-time optimizations based on designer-provided parameters and does not change the schedule at runtime. Saadatmand et al. [27] propose a lookup-driven approach where different encryption algorithms and their overheads are listed in a sorted table. At runtime, encryption algorithms are selected (from a higher to lower order, sorted by their execution time) based on available time. They implemented this scheme on the OSE RTOS [49].

Remarks: *We observe that the notion of "security level/service" used in RTS literature is not well-defined and it is hard to quantify how the security posture is improved. The majority of the solutions (eight out of eleven) are designed for single-core platforms. Only a single work [27] is implemented on an actual RTOS while the others are evaluated using simulations.*

Table 3. Summary: Integrating Security Monitoring Techniques

Reference	Task Model	Scheduling Policy*	Platform	Key Idea	Overhead/Limitation
Hamad et al. [30]	Independent, periodic, preemptive	Fixed priority	Single core	Monitor execution time variations of real-time tasks within the scheduler	May not detect attacks that cause minimal timing perturbations; extra tracing overheard; no study on overhead of monitoring
Hao et al. [31]	Independent, periodic, preemptive	Fixed priority	Single core	Insert security checks (e.g., control flow integrity protection) within task code	May result in delayed detection since some jobs skip security checks
Hasan et al. [32]	Independent, periodic, preemptive	RM	Single core	Execute monitoring tasks with a lower priority than real-time tasks	Delayed detection due to more interference (since monitoring tasks run only during idle times)
Hasan et al. [33]	Independent, periodic, preemptive	RM	Single core	Execute monitoring detection tasks with a lowest priority most of the time (i.e., during normal operation); change the mode of operation and execute with a higher priority (for a limited amount of time) if anomalous behavior is suspected	False positive detection may cause unnecessary mode switches
Hasan et al. [34, 35]	Independent, periodic, preemptive	Fixed priority	Multicore	Execute monitoring tasks with a lower priority than real-time tasks and (a) use a fixed core allocation [34]; (b) allow runtime migration to any empty core for faster detection [35]	Delayed detection due to more interference (since monitoring tasks run only during idle times)

*We refer scheduling policy as "RM" if the task priorities follow rate monotonic order (i.e., a task with shorter period is assigned a higher priority).

5.1.3 Control-Aware Solutions. In another direction, Lesi et al. [28, 29] propose techniques to prevent **man-in-the-middle (MITM)** attacks (between sensors and controllers) in real-time control systems. The goal is to find tradeoffs between control performance and security overheads (e.g., overheads for enforcing data integrity technique such as message authentication codes to prevent MITM attacks).

Remarks: *Five out of six papers [31–35] we reviewed in this category do not provide specific security techniques and abstract the underlying checking mechanisms. Only a single paper [30] explicitly designs a checking scheme by observing the timing behavior of the tasks. Majority of the solutions (four out of six) [30–33] are designed for single core platforms.*

5.2 Integrating Security Monitoring Techniques

We now discuss scheduler-level techniques to integrate *security checking* (such as monitoring task execution behavior and protecting control flow integrity). Table 3 summarizes the articles presented in this section.

5.2.1 Time-Based Monitoring. As we mentioned earlier, one of the main characteristics of RTS is that they have strict timing constraints that must be met in order to maintain the correctness of the system. Hamad et al. [30] use the temporal properties (derived from the static timing analysis [50] of the system) and propose an intrusion prediction mechanism. The authors introduce the concept of *red-zone principle* to permit the task to overrun until a predefined limit (called *red-zone*) is reached. Whenever the task exceeds the limit, a recovery process is performed (e.g., terminate the malicious task).

5.2.2 Task-Level Monitoring. In the above work, a single monitoring mechanism executes in the scheduler and checks the timing behavior of the real-time tasks. Hao et al. [31] propose to individually execute security checks (such as checking of control flow) for each task. Different security checks (called *security levels*), however, result in different (worst-case) execution of the tasks and may impact timing requirements. The authors therefore propose to select a subset of jobs to execute the security checks and design a scheduling policy, called security level monotonic (SLM) algorithm, to obtain such set of jobs (that runs security checks).

5.2.3 Securing Legacy Systems. The aforementioned work performs security monitoring at the task-level and increase execution time of the tasks. While prior work is more suitable for newer systems, this is especially challenging for legacy systems where the real-time tasks are already in place and perhaps cannot be modified. Hasan et al. [32–35] propose to execute monitoring mechanisms as independent, periodic tasks (called *security tasks*). The security tasks execute at a lower priority than real-time tasks so that they do not perturb the timing or execution order of the existing real-time tasks (called opportunistic execution) [32].

Hasan et al. [33] further show that when the security tasks always execute with the lowest priority it may result in longer detection times. They then propose a dual-mode model (Contego [33]) that allows the security tasks to execute in two different *modes*: (a) by default security routines execute opportunistically when the system is deemed to be uncompromised; (b) if an anomaly is suspected, the security tasks may switch to higher priority; (c) the system reverts to "normal" mode if: (i) no anomalous activity is found or (ii) the root cause of the problem is detected and malicious entities are removed.

The above work focuses on single core systems. Hasan et al. [34] extend their scheme for multi-core platforms (called *HYDRA*). They further show that, if security tasks can migrate across cores, at runtime, it provides better detection (HYDRA-C) [35]. This scheme provides better monitoring but comes with a cost (in terms of context switch overhead).

We now discuss techniques to mitigate side-channel attacks against RTS (see Table 4 for a summary). We classify these techniques are: (a) prevention of information leakage (Section 5.3) and (b) randomization of task schedule (Section 5.4).

5.3 Side-Channel Defense: Leakage Prevention

We first present techniques that prevent information leakage due to the use of shared resources (Section 5.3.1) and then analyze the effect of power leakage on real-time scheduling (Section 5.3.2) and discuss techniques to defend schedule-based side-channel leakages (Section 5.3.3).

5.3.1 State Cleaning Mechanisms. It is well understood that the use of shared resources (e.g., caches, DRAMs, I/O interconnections) can lead to information leakage between tasks without explicit communication [54, 55]. In particular, every time there is a switch between tasks belonging to different "security levels" (as defined at the design time) there is a possibility of information leakage through shared resources. The information from a task with a higher security level (say τ_H) must not leak to task with a lower security level (τ_L). Mohan et al. [36] propose the idea of mitigating information leakage among tasks with fixed security levels by placing constraints on scheduling algorithms. The main intuition is that every time tasks switch between security levels, the shared resource must be "cleansed" (e.g., cache should be flushed in this case) before a new task is scheduled.

Pellizzoni et al. [37] further relax the requirement of total ordering of security levels and propose a more general model. They propose a constraint named `noleak` to capture whether unintended information sharing between any given pair of tasks must be forbidden (e.g., for any two tasks τ_i and τ_j: if $\texttt{noleak}(\tau_i, \tau_j) = True$, then information leakage from τ_i to τ_j must be prevented). The

Table 4. Summary: Side Channel Defense Techniques

Reference	Task Model	Scheduling Policy	Platform	Key Idea	Overhead/Limitation
Mohan et al. [36], Pellizzoni et al. [37]	Independent, periodic, preemptive/ non-preemptive	Fixed priority	Single core	Flush the shared medium (e.g., cache) between the consecutive execution of high-security (i.e., security sensitive) and low-security critical tasks	Depending on platform, flushing can be costly; overhead of flushing reduces schedulability
Jiang et al. [38]	Independent, periodic, preemptive	EDF, RM	Single core	Use statistical analysis and study the difficulty (in time overheads) for the attacker to obtain information about system power usage	No prevention scheme is presented
Bao et al. [39]	Independent, aperiodic, preemptive	Variant of EDF (proposed by the authors)	Single core	Find a task sequence (i.e., schedule) that minimizes thermal side-channel leakage	Does not provide hard timing guarantees
Chen et al. [51]	Periodic, preemptive	Fixed priority	Single core	Find a schedule that prevents an untrusted task to execute certain point in time to minimize the chances of schedule leaks	Designed for single core platforms only, requires additional OS support (e.g., Linux `cgroups`)
Yoon et al. [52, 53]	Hierarchical, periodic, preemptive	Fixed and dynamic priority	Single core (can be adapted to multicore)	Find a schedule that makes the partition execution behavior oblivious to an adversary	Applicability to multi-core is not thoroughly analyzed
Yoon et al. [40], Baek et al. [41], Vreman et al. [42], Krüger et al. [43]	Independent, periodic, preemptive	Fixed priority [40, 41]; schedule independent [42]; slot-based, time-triggered [43]	Single core [40, 42, 43], multi-core [41]	Obfuscate task execution order (i.e., schedule) while retaining schedulability to reduce the predictability	Extra context switch overhead; no clear metric to analyze how randomness can improve security

authors then propose a general flushing mechanism based on the `noleak` relation and compute the effect of the number of flushing invocations on the timing requirements.

Remarks: *We find that researchers study side-channel defensive techniques from two views: (i) leakage prevention by enforcing scheduling constraints (i.e., techniques for preventing cache [36, 37], thermal [38, 39] and schedule [51–53]-based side-channel leakages) and (ii) schedule diversification (preventing system from timing/thermal/cache inference attacks by obfuscating task execution orders) [40–43]. Majority of the papers (9 out of 11) we reviewed [36–43] are evaluated by simulations and do not show how (i) scheduler-level constraints (such as flushing the cache or randomization) can be implemented on practical RTOS/schedulers and (ii) exactly how these techniques can limit the impact of actual side-channel attacks on RTS.*

5.3.2 Power Leakage and Real-Time Scheduling. By analyzing the power traces or by using statistical analysis and error correction techniques, an adversary can gain information about the system [56]. Jiang et al. [38] develop an analytical framework to study the robustness of AES secret keys against differential power analysis attacks [56] for both, RM and EDF real-time scheduling policies. Bao et al. [39] show that different orders of task executions can result in different thermal profiles and thus leak different side-channel information (e.g., processor temperature). They then propose a scheme that minimizes the probability of task execution inference (that an attacker may deduce from the thermal sensor measurements).

5.3.3 Defending Schedule Leakage. SchedGuard [51] protects Linux-based real-time schedulers against side-channel attacks (such as ScheduLeak [44]) by preventing untrusted tasks from executing during specific time segments. The authors integrate SchedGuard into the Linux kernel

using cgroups. Yoon et al. [52, 53] show an algorithmic covert timing-channel between partitions in hierarchical schedulers [57, 58]. In hierarchical scheduling, each partition (i.e., temporal block) exclusively uses CPU for running set of tasks assigned to the corresponding partition. The authors introduce a run-time schedule transformation algorithms (named *Blinder* [52] and *TimeDice* [53]) that make the partitions oblivious to the other partition's varying temporal behavior even if an adversary is able to control the timings of the applications.

5.4 Side-Channel Defense: Schedule Obfuscation

Another way to protect RTS from side-channel attacks is to *randomize the task schedule* to reduce the observability of periodic real-time applications [40–43]. A randomized task schedule results in different scheduling orders (and response times) of the tasks. With randomization, even if an observer is able to capture the exact schedule for a (limited) period of time the rest of the schedule will show different timing behavior for execution of the tasks while retaining real-time guarantees. However, this is not straightforward for RTS since schedule obfuscation leads to priority inversions [59] and may cause missed deadlines.

Instead of selecting the highest priority task at each scheduling point (as is the case for vanilla real-time schedulers), TaskShuffler [40] picks a random task (subject to deadline constraints). Baek et al. [41] further extend TaskShuffler for multicore platforms. Contrary to TaskShuffler, Vreman et al. [42] generates a list of randomized schedules offline, based on a metric—called *upper-approximated entropy*—to quantify the diversity of the execution as well as the probability of learning the schedule (by an adversary). Krüger et al. [43] propose a combined online/offline randomization scheme to reduce determinism for time-triggered systems [60] where tasks are executed based on a pre-computed, offline, slot-based schedule.

6 SYSTEMATIZATION OF SCHEDULER-LEVEL DEFENSES: METRICS & ANALYSES

The various scheduler-level defense techniques discussed thus far (see Section 5) are designed with separate application goals in mind; it is not easy to *characterize* them in a unified way. Hence, we now present a systemization approach to methodologically compare them under a common "metric." The systemization of defense techniques presented here complements our survey and sheds new light on designing a security evaluation metric.

The challenges with analyzing or comparing the scheduler-based security mechanisms are:

(1) the various techniques seem to address different problems;
(2) each subcategory uses a different defensive mechanism, i.e., integration of cryptographic primitives, monitoring and side-channel defenses;
(3) no real metrics exist especially ones that can measure a shared quantity; and
(4) the implementations are not readily available.

Even with such disparate mechanisms, we were able to glean the following insight—all of the techniques are meant to increase the difficulty for adversaries and, in RTS where time and computational resources are at a premium (and are closely tracked), this translates to: *how much time does an attacker have available?* This measure of timing availability could be for an attacker to carry out its objective, the time remaining before it is detected or kicked out, the (limited) time that it has to steal information that will eventually be cleansed or even the window of time (larger, smaller) that the attacker needs to observe for reconstructing useful information. We will enumerate all of these in detail in the remainder of this section. We name this metric, the real-time "**attacker's burden**" (AB). Our metric is inspired by the general concept of *"work factor"* [1] that evaluates the *cost* of circumventing a given security mechanism with the resources available

to a potential attacker. A similar concept is also used by the work in the crypto community where the security provided by a cipher is measured in the number of bits [61, Ch. 3]. In our context, the attacker's burden metric overlays the notion of the *"time"* available to an adversary on top of the computational ''*work factor*", when different scheduler-level defensive mechanisms are enforced.

Note that the computation of this metric is not straightforward due to the differences in the defensive mechanisms, as mentioned earlier. The use of such a metric allows us to compare the various approaches while abstracting away the (i) high-level details such as scheduling policies, defensive mechanisms, system models and assumptions of the adversary as well as (ii) low-level details such as the operating system, execution platforms, attack surfaces, and vulnerabilities.

In the rest of this section, we demonstrate how to measure this "time available to attackers" for the various categories and then formalize it. We first present our evaluation methodology.

Experimental Methodology. From Section 5, we see that the four distinctive scheduler-level defense mechanisms are: (i) integrating cryptographic security services, (ii) periodic monitoring to detect intrusions, (iii) state cleaning mechanisms to prevent storage-based side-channels, and (iv) schedule obfuscation. To measure the attacker's burden, we analyzed each of the proposed mechanisms and extracted the core concept being proposed. This concept was implemented in a simulator [62] and we measured how much time is available to an attacker (the attacker's burden as listed in each of the following sections), once the security mechanism is in place. Depending on the actual technique, we varied relevant parameters and also inputs (a large number of real-time task sets that were generated) to explore the design space. The details of how we generated the input sets and the platform are in the Appendix. Our code and relevant data sets have been open-sourced in a publicly-available repository [62].

6.1 Analysis

We now evaluate all four different categories of schedule-based defenses and demonstrate how to derive our metric for each. We summarize our findings in Section 6.2.

Note: We assume that the reader is familiar with the content presented in Section 5 since our metric is meant to measure (and compare) the techniques presented there.

Study 1: Integrating Cryptographic Services in RTS

The closest analogy for the measurement of an "attacker's burden" is in the integration of cryptographic primitives in RTS [19, 20]. It is well known that the strength of a cryptographic algorithm is estimated by the number of operations (and hence, time) it takes to reconstruct the key [63]. For instance, if the key is l bits long and the adversary can test k keys per time unit, it will take, on the average, $\frac{2^{l-1}}{k}$ time units to find the key. While this method already incorporates a notion of time, albeit in a loose manner, researchers have analyzed additional timing properties, *viz.,*

(1) How long does it to carry out cryptographic operations in RTS [19]?
(2) How many cryptographic operations, M, are required by a real-time job (e.g., if a task τ_i encrypts two individual messages in the same job then $M_i = 2$) [19–21]?
(3) What is the maximum key size, l^{max} that can be tolerated in a system and still meet the deadlines [21]?

Hence, based on this intuition, we define the attacker's burden for real-time cryptographic operations (AB_{crypto}) as a function of the key size (l) the number of services (M) and the maximum key size that is tolerable in the RTS (l^{max}) as follows:

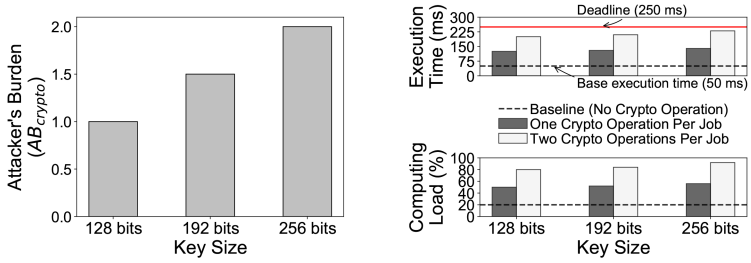
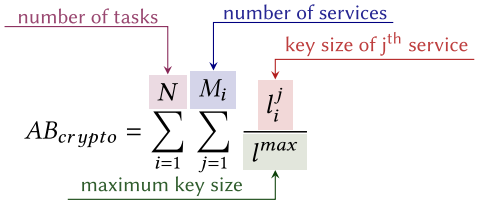


Fig. 3. (a) The attacker's burden (normalized to $[0, 1]$) and (b) computational load for AES encryption with 128, 192, and 256 bits key sizes. The task is sampled at 4 Hz and the deadline is equal to the sampling rate (250 ms). The base load in Figure 3(b) represents computing load without any cryptography operations. Longer key size leaves less time for the attacker due to increased computational difficulty to recover the key.



That is, the amount of time available to an attacker ($\mathcal{T}_{crypto}$) is inversely proportional to the difficulty of obtaining the key, i.e., $\mathcal{T}_{crypto} = \frac{1}{AB_{crypto}}$, where $\sum_{i=1}^{N} \min_{\forall j}\{\frac{l_i^j}{l^{max}}\} M_i \leq AB_{crypto} \leq \sum_{i=1}^{N} M_i$. In other words, the time available to an attacker (between cryptographic operations) is *reduced* and their "burden" *increases* with the key size and number of operations.

Figure 3(a) plots AB_{crypto} for integrating the AES algorithm into RTS—for three key sizes: 128, 192, and 256 bits. As expected, the attacker's burden (y-axis), i.e., the amount of time $\mathcal{T}_{crypto}$ available to break the security mechanism (a cryptographic algorithm in this case), decreases (i.e., AB_{crypto} increases) with an increase in key size due to higher computing demand (x-axis). While this may be obvious in the case of crypto, it is less so for the other security mechanisms as we shall see—in fact, the direct correlation with cryptographic algorithms helps establish a baseline for how the metric can be used.

Figure 3(a) demonstrates that with increasing key sizes (that are still within the max key sizes that are acceptable for a given timing constraint) *an attacker will have to expend increasing amounts of time.* As expected, the computing load of a real-time task also increases with additional cryptographic operations, but just barely, as seen in Figure 3(b) that plots the computing load (y-axis) vs. increasing key sizes (x-axis). The figure shows the base load using the horizontal dotted line (i.e., without any cryptographic operations) and the effect of adding one or two crypto operations in each job cycle (the two bars, dark and light)—while the burden for the attacker is increased due to larger key sizes, the computing load on the system (for normal operations) is only slightly increased. It is important to note that these solutions were designed to finish before the deadlines (the computing loads even with the crypto operations are lower than the deadline) but, until now, no one had captured the additional timing load from the perspective of the attackers.

The time available to an adversary reduces due to increased demands in computational power to recover the key with larger key sizes and/or increased number of cryptographic functions used.

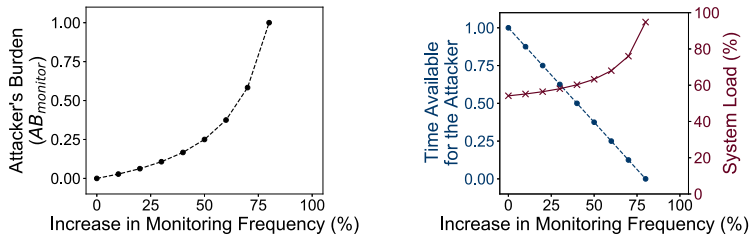


Fig. 4. Tradeoff between monitoring frequency and computing load: more frequent monitoring leaves less time for the attacker (i.e., increases burden), see left plot and blue y-axis of the right plot). However, this also increases computing load (reduced schedulability) as shown in the right plot (red y-axis).

Study 2: Integrating Periodic Security Monitoring Tasks

Security mechanisms based on periodic monitoring operate on the principle that they must execute *at least* as often (if not faster) than a designer specified frequency [32–35, 64] in order to detect intrusions. From the perspective of an attacker, an increased monitoring frequency means a higher chance of detection or, put another way, lesser time between checks to carry out a successful attack. Of course, as before, the frequency of the monitoring tasks is limited by the need to meet the timing requirements (deadlines) of the RTS and do useful work. Hence, the burden on an adversary ($AB_{monitor}$) can be defined as, *how much time is available between successive invocations of a monitoring task*, i.e.,

$$AB_{monitor} = [\ \overbrace{t_{j+1} - t_j}^{\text{time between consecutive invocations}}\ ]^{-1}$$

t_{j+1}: time of $(j+1)^{th}$ invocation; t_j: j^{th} invocation

What $AB_{monitor}$ indicates is that if the jth and $(j+1)$th job of monitoring task, invoked at times t_j and t_{j+1}, respectively (where $t_{j+1} > t_j$), then an attacker will have at most $t_{j+1} - t_j$ units of time (i.e., the monitoring frequency/period of the security task) to launch and complete its attack. Hence, we calculate the time available for an adversary to cause any damage as follows: i.e., $\mathcal{T}_{monitor} = \frac{1}{AB_{monitor}}$. Let U_i is the processor load (i.e., ratio between its execution to period) of the task τ_i and C_M is the execution time of monitor task. We can calculate an lower bound on time available for the attacker as follows: $\mathcal{T}_{monitor} > \frac{C_M}{U_B - \sum_{\forall i \neq M} U_i}$ where and U_B is the maximum available CPU utilization for a given system.[1]

To demonstrate how this metric can be applied, let the designer specified minimum monitoring frequency be designated as a "base frequency".[2] In our simulations, we increase the monitoring frequency and then measure the effects on the attacker burden; this is plotted in Figure 4. The x-axes of Figure 4(a) and Figure 4(b) show the percentage of CPU time spent on monitoring. The y-axis of Figure 4(a) is $AB_{monitor}$. We further plot the time available the attacker and the impact on the CPU load, i.e., the left y-axis is $\mathcal{T}_{monitor}$ while the right y-axis plots the computational load on the system.[3] As shown by the graph, *as we increase the frequency of monitoring the attacker's burden*

[1]For example, rate monotonic (RM) scheduler has utilization bound $U_B = 69.3\%$ [14].
[2]Note that according to the designers of these mechanisms [32], the system will be schedulable at this monitoring frequency, i.e., meet all of its deadlines.
[3]We carried out additional simulations that started with different initial system load conditions and demonstrate the impact of adding increased monitoring—these results are presented in Figure 8 in the Appendix.

increases, i.e., the amount of time left for an attacker is significantly reduced. As expected, the load on the system increases dramatically and, after a certain point, the system becomes unschedulable—i.e., real-time tasks start missing their deadlines. Hence, there is a limit on how much increased monitoring can be tolerated by the system.

Frequent monitoring leaves less time for adversaries to carry out their attacks but this comes with increased costs and potential for missed deadlines—the attacker's burden metric allows designers to map out the costs vs. benefits for such systems.

Study 3: Leakage Prevention by Shared State Cleanup

When considering techniques that use state cleansing to prevent side-channel attacks in RTS [36, 37], the attacker's burden must capture the amount of time left for the adversary to retrieve the information in the shared resource (e.g., caches) before it is "flushed". In most of these techniques (Section 5.3.1), the researchers define a "security relationship" (a lattice [8, 36, 65], a pairwise function [37]) between tasks in the system. When the context switches from a task with a higher security classification to one that is lower, these security algorithms "flush" the shared resource, thus preventing the leakage of information while also incurring the overheads for the cleanup. Hence, we measure the attacker's burden (AB_{flush}) as the *amount of time between consecutive flushing events*,

$$AB_{flush} = \left[\; Q_p \left(t_{j+1} - t_j \right) \middle| \, \forall j \; \right]^{-1}$$

p-percentile time difference of all consecutive context switches

time of (j+1)th context switch

jth context switch

where t_j and t_{j+1} are the jth and $(j+1)$th context switches. Q_p is the pth percentile time difference of all consecutive context switches in the schedule; the attacker will have AB_{flush} unit of time (with probability p) to snoop on the shared medium. We use Q_p since the time intervals between consecutive flushes can vary. So with the pth percentile, we can say that the interval is x with probability p. As in the case of periodic monitoring, we can calculate the time available for the attacker before two consecutive flushing as follows: $\mathcal{T}_{flush} = \frac{1}{AB_{flush}}$.

The x-axis of Figure 5(a) plots the increased frequency (reduced period) for the critical (higher security level) task while the y-axis represents the attacker's burden, AB_{flush}. Assuming that the attacker's frequency/period is fixed and we are able to adjust the frequency of the critical task(s), the figure shows that it becomes increasingly difficult for an attacker to carry out its objectives (i.e., AB_{flush} increases) since there is less time available between the cleaning events. On the other hand, as shown in Figure 5(b) (x-axis frequency of critical tasks; y-axes number of flushes and system load, respectively), the overall load on the system is reduced as the critical task frequency is increased—since there are fewer flushes between instances. Note that the assumption is that the cost of a "cleanse"/flush is assumed to be constant in these articles. Also, the authors have computed an upper bound on the number of flushes required by preemptive and non-preemptive systems as $2n_c + 1$ and $n_c + 1$, respectively where "n_c" is the number of critical jobs in the system. Hence, designers can now compute the costs (state cleanup overheads, increased system utilization) vs benefits (reduced time for adversaries to carry out attacks) by using the attacker's burden value.

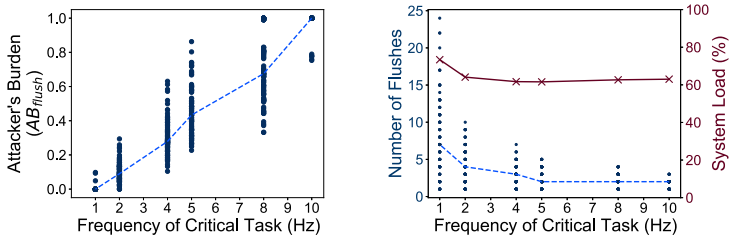


Fig. 5. (a) Attacker burden (as available time between state cleaning) and (b) computing load (as number of flushes) vs period of a critical task. When attacker's task has a fixed period (say, 20 Hz) then a less frequent execution of critical (high security) tasks results in more instances of attacker/low security tasks occurring between two instances of a high security task, thus resulting in more flushing events – this reduces the time, and increases the burden, for attackers.

State cleanup mechanisms reduce the effectiveness of leakage attacks in RTS by decreasing the window of time available to attackers to snoop upon the shared resource and the attacker's burden is able to plot this clearly.

Study 4: Schedule Obfuscation

As mentioned earlier, schedule randomization [40–43] has been suggested as a mechanism to prevent side-channel attacks in RTS. From this work, we note that there is a direct correlation between the ability of an adversary to recreate a task schedule (or timing behavior) to: (i) the tine window for carrying out observations and (ii) the predictable, repeating execution patterns of the real-time tasks. Hence, the main objective of these obfuscation mechanisms is to reduce both of the above factors. By elongating the window of time when a task can appear, the authors reduce the predictability of the behavior that can be observed by an attacker. Since at any point only one outstanding job of a task can exist in a window (the window is usually the period of the task),[4] *an attacker has to observe the system for a larger window of time*, thus leaving it with very little time to reconstruct the scheduling behavior before the next instance of the job arrives.

Hence, to measure the burden on an attacker, we calculate the "spread" of a task τ_i, i.e., *what is the largest window of time when a task can appear*. For a given schedule, we compute the attacker's burden, AB_{rand}, as:

$$AB_{rand} = \frac{R_i}{T_i}$$

(R_i: response time; T_i: period)

This formula computes the spread of the response times, R_i (time between arrival and completion of a task), across its period, T_i. For a given schedule, a higher value of AB_{rand} indicates that the task can appear in a larger window of time, once released. We further calculate the time available for the attacker when the randomization is active as follows: $\mathcal{T}_{rand} = T_i - \frac{1}{T_i} AB_{rand}$. The metric

[4]One job per window is a common assumption in RTS [40, 44, 66].

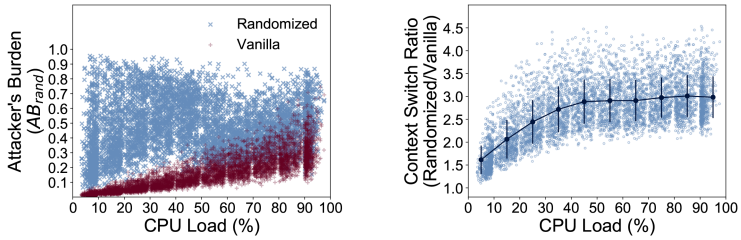


Fig. 6. (a) Attacker's burden (observability of a task from the schedule, defined as a ratio of its response time and period) and (b) context switch ratio for both vanilla execution and obfuscated schedule. Randomizing task schedule reduces determinism (i.e., more burden on the attacker to infer a task) as shown by the higher ratio (left figure) with a increase in number of context switches (right figure).

$\mathcal{T}_{rand}$ tells us how much "slack time" available for an untrusted task to carry out any malicious action (say observing the behavior of an victim task) before its next periodic invocation.

In Figure 6(a) we plot this attacker burden (y-axis) against the utilization (x-axis) of both randomized as well as vanilla systems, for a **rate monotonic (RM)** scheduler. As we see from the figure, the purple dots (vanilla, without schedule obfuscation) show a much narrower range of times when tasks execute—thus making it easier for adversaries to observe and recreate their behavior. When the schedule obfuscation/randomization schemes are applied, the tasks appear more spread out and it becomes harder for an adversary to predict their schedules and/or timing behavior.

The problem with introducing randomization techniques is that they increase context switch overheads as shown in Figure 6(b)—the y-axis plots the ratio of context switches in randomized schedules to vanilla schedules while the x-axis plots the CPU utilization. As we see from the plot, the randomized schedules have significantly higher context switch overheads but the attacker's burden provides information on the security gains obtained from such mechanisms to designers—hence, they can decide whether the security vs. overheads tradeoff is worth investing in.

Schedule obfuscation mechanisms increase the burden on attackers since they have less effective time to recreate the behavior of a task as they must expend more time on observing the schedules.

Study 5: Comparison across All Four Schemes

We finally compare all four schemes in Figure 7 with a common metric (CPU load, x-axis in the figure) and plot the attacker's burden (y-axis, normalized into the interval $[0, 1]$). We note that while we plot the observations from different schemes in the same figure for illustration purposes, they are four different (and often complementary) techniques. For a pair of schemes (i, j), a higher value in y-axis (say for a given CPU load) for technique i does not imply it adds a larger burden on the attacker than j. Instead, the goal of our evaluation is to present the "trends" (i.e., whether the burden on the attacker increases/reduces with varying CPU load) in the attacker's burden metric for these four distinct approaches. As we see in Figure 7 (and also from our previous study), the time available for an adversary is reduced (i.e., more burden) for the first three schemes (see Sections 5.1, 5.2, and 5.3.1) with increased load. As we discussed in Section 6.1 (and also illustrated in Figure 7), for randomization, the time between arrival and completion of a task is higher in the low-to-medium CPU loads. That is, a system that with less than 60% load has more burden on the attacker (to infer the task execution pattern from the schedule) than the highly utilized systems. The effect of randomization reduces (less burden) in higher loads to ensure the timing constraints for all tasks.

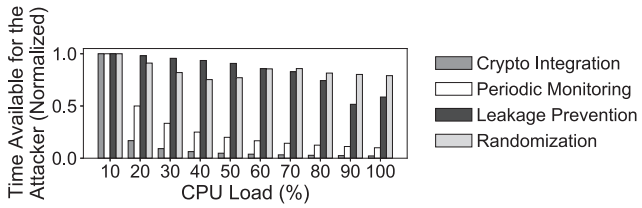


Fig. 7. Comparison of different schemes under a common metric (CPU load). For the first three schemes (i.e., except randomization), the available time to disrupt the system is reduced (i.e., more burden on the attacker) with increased load. For randomization, the burden on the attacker (i.e., inferability of a task from the schedule) is most in low-to-medium loads.

Table 5. Summary of Our Observations on the Attacker's Burden

Approach	Metric: Attacker's Burden	Expression	Observation
Integration of cryptographic services (Section 5.1)	Time to recover the key – inversely proportional to the key size and number of services	$AB_{crypto} = \frac{1}{\sum_{i=1}^{N}\sum_{j=1}^{M_i}\frac{l_i^j}{l^{max}}}$	The time available for an adversary to recover the key decreases for larger key sizes due to increased computing load
Periodic monitoring (Section 5.2)	Time between invocations of monitor task	$AB_{monitor} = t_{j+1} - t_j$	Attacker's burden metric captures the tradeoff between monitoring frequency and computing load: unfettered, frequent security monitoring can violate real-time constraints
Leakage prevention by flushing (Section 5.3.1)	Time between flushing shared medium	$AB_{flush} = Q_p(t_{j+1} - t_j \mid \forall j)$	Our metric finds the effectiveness of information leakage—increasing frequency of critical task (for a given observation frequency of an adversary) can minimize the chances of cache information leakage
Schedule randomization (Section 5.4)	Observability of task response times (difference between arrival and completion) to its period	$AB_{rand} = \frac{R_i}{T_i}$	Attacker's burden metric captures the difficulty of observing task execution pattern from a (randomized) schedule by calculating how sparsely a task appears in the schedule

Our attacker's burden metric is able to capture differences and commonalities across the four distinct scheduler-level security classes. For the first three approaches (integration of cryptographic primitives, periodic monitoring, state cleaning techniques), the time available for an adversary reduces, hence, more burden with increased load, while for randomization, the burden is more (i.e., task spread is high and hard to infer) in systems with low-to-medium loads.

6.2 Summary

In this section, we introduce the notion of the "attacker's burden" (summarized in Table 5) and methods to compute it for various real-time scheduler-based defense mechanisms. For each class of defensive techniques introduced in Section 5, we are able to *compute the reduced time available to an adversary* —these are directly comparable, even across defensive classes! By seeing how much time is available to an adversary, along with the overheads that are imposed, we can design systems that are tolerant to both—attacks as well as overheads.

7 OTHER RESEARCH

So far, we discussed scheduler-level (software) solutions. There also exist techniques that use hardware/software-based architectural frameworks (Section 7.1) and research on secure RTOS design (Section 7.2) as we present in this section.

Table 6. Summary of Hardware/Software-Based Solutions

Technique	Key Idea	Overhead/Limitation
Memory isolation and access control [68–70]	Runtime memory access control through on-chip memory protection unit [68, 69] or software modules [70].	Platform dependent (e.g., requires an on-chip memory protection unit) [68, 69] and limited portability (e.g., only supports bare-metal or FreeRTOS applications) [69, 70]
Monitoring by separate computing unit [67, 71–75]	Use verified/secure hardware module to monitor system behavior (e.g., timing [71, 72], execution patterns [73], memory access [74], system call usage [75], control flow [67])	Limited compatibility with COTS systems since they require custom hardware/monitoring unit
Proactive defence by platform reboot [76–78]	Periodic and/or event-driven (say when an abnormal activity is detected) reboot and reload an OS/applications from a read-only media	Requires extra hardware for triggering (periodic/asynchronous) restart events
Trusted execution environment (TEE)-based security [79–81]	Leverage TEEs (i.e., TrustZone [79, 80] and SGX [81]) to execute whole RTOS [79] or task segments [80, 81] within a secure enclave	Context switch overheads; no isolation/protection among tasks running inside secure enclaves

7.1 Hardware/Software-Based Mechanisms

We now present techniques that either require architectural support and/or custom hardware (see Table 6 for a summary).

7.1.1 Control Flow Integrity. Abdi et al. [67] propose a hardware-based mechanism where an on-chip control flow monitoring module with a dedicated memory unit directly hooks into the processor and tracks the control flow of the tasks. The monitoring module monitors the control flow at runtime and compares it to a stored control flow graph (obtained offline).

Many modern micro-controllers (e.g., ARM Cortex-M and Cortex-R) provide hardware-enforced memory isolation. MINION [68] leverages those COTS **memory protection units (MPUs)**. All software modules are executed in an unprivileged mode and only a lightweight software module is allowed to run in the privileged mode—this is to reduce the attack surface and minimize privilege mode switching overheads. A similar line of work is the RECFISH framework [69] that provides a binary instrumentation method for ARM platforms that protects both, bare-metal applications and those that run on a RTOS (FreeRTOS) by memory isolation. RECFISH uses hardware privilege-levels and context switching to isolate shadow stacks from untrusted code. Du et al. [70] introduce a software-based approach, Kage, that stores all control data in separate memory regions from untrusted data. The authors implemented Kage as an extension to FreeRTOS. In Kage, (a) a Kage-compliant compiler transforms code to protect critical memory regions and add control-flow integrity checks and (b) a set of secure APIs allow safe updates to the protected data.

7.1.2 Slack-Aware Monitoring. As we mentioned earlier, the worst-case execution time estimation for real-time tasks provides the designer a safe upper bound while determining the schedulability of the system at the design time. However, the tasks often run faster than conservatively estimated timing bound, leaving behind dynamic slack (i.e., the time instance when no other task is executing). Lo et al. [72] leverage this slack time and propose a hardware architecture for run-time monitoring (i.e., monitoring is only performed when enough slack exists).

7.1.3 Simplex-Based Security. Simplex [16] is a well-known real-time architecture for improved fault-tolerance that utilizes a minimal, verified controller as backup when the main, high-performance controller is not available or malfunctioning. While traditionally Simplex has been used for fault-tolerance [82, 83], recently researchers proposed to use Simplex-based architectures for securing RTS [71, 73–75, 77]. The key concept of using the Simplex for security is to use a minimal, simple, subsystem (say a trusted core) to monitor the properties (i.e., timing [71, 73], memory access [74], system call usage [75], behavioral anomalies [77]) of an untrusted entity that is designed for more complex tasks and/or exposed to less secure mediums (e.g., network, I/O ports).

Table 7. Research on Secure RTOS Design

Reference	Approach	Application Domain
seL4 [89]	Formally verified microkernel-based hypervisor; provides isolation between security-critical applications	Domain independent
Composite [90]	Hardware isolated configurable components for better reliability/security	Domain independent
ERTOS [91]	Resource isolation/protection for different subsystems	Domain independent
AOS [92]	Modular isolation and access control	Avionics
TrackOS [93]	RTOS-level control flow integrity protection	Domain independent
GenRTOS [94]	Generic OS abstractions with minimal (timing-related) services	Domain independent

S3A and SecureCore Framework. In S3A [71], a trusted hardware component monitors the execution behavior of a real-time control application running on a untrustworthy main system. The S3A framework utilizes the knowledge of deterministic execution profile and timing of the system (obtained at the design time) and use it to detect the violations (in execution time and activation period of control tasks) of expected system behavior.

The SecureCore framework [73–75] utilizes a trusted entity that can continuously monitor the behavior of a real-time application on an untrustworthy entity. The initial SecureCore architecture [73] uses a statistical learning-based mechanism for profiling the correct execution behavior. The SecureCore framework is also extended (a) to profile memory behavior [74] and (b) detect anomalous executions using a distribution of system call frequencies [75].

Restart-Based Recovery. Both S3A and SecureCore are *reactive* security mechanisms in a sense that they do not prevent the intrusions and only focus on the aftermath of the infection of application codes. Abdi et al. [76–78] propose a *proactive* mechanism (by restarting the platform). Unlike conventional computing systems (e.g., servers, smartphones) restart-based mechanisms are much harder to implement in RTS due to the temporal constraints and interactions with the physical entities (for example, a UAV can quickly be destabilized if its controller restarts at the wrong time!). They develop frameworks to frequently reboot the system and load a fresh image of the tasks and OS from read-only media, so that attackers can have less time to either destabilize or even re-enter the system and cause meaningful damage [76–78].

7.1.4 Trusted Execution Environments (TEEs) for RTS. While TEEs (such as ARM TrustZone [17] and Intel SGX [84]) are supported by hardware, they still create significant overheads especially in the context of RTS. In Super-TEE [80], multiple real-time code/application sections can fuse together to reduce TEE (ARM TrustZone) execution overheads. FreeTEE [79], a virtualization-based solution, allows Linux and FreeRTOS to execute simultaneously while maintaining real-time performance. Although directly not in the context of security, hypervisor-based solutions that utilize TrustZone for real-time use-cases have also been proposed in literature [85–88]. AegisDNN [81] uses SGX enclaves to protect the critical part of real-time inference tasks. While researchers are exploring techniques to incorporate TrustZone and SGX-based TEEs for real-time applications, research in this segment requires further investigation.

7.2 Designing Secure Real-Time Operating Systems (RTOS)

One way to address threats at the RTOS level is to extend partitioning and provide isolation. Commercial RTOS such as VxWorks [95] and QNX [96] have built-in security extensions. VxWorks provides secure boot, a secure run-time loader (to prevent authorized access), network security through SSL and encrypted containers [97]. QNX relies on secure programming standards (e.g., POSIX PSE52) and enables security by resource partitioning [98].

There also exists academic research to design secure RTOS (see Table 7). seL4 [89], a formally verified, open-source, secure microkernel, supports real-time applications with different criticality using temporal isolation. Composite [90] is an open-source minimal kernel with configurable

components. ERTOS [91] is another component-based RTOS that provides spatial and temporal isolation for different subsystems.

AOS [92] is a modular kernel for avionics applications that provides fundamental services such as secure initialization, resource, process and time management/synchronization, exception handling and I/O support. TrackOS [93] provides built in support for **control-flow integrity (CFI)** checks for real-time tasks. GenRTOS [94] is a generic low-level RTOS model that provide minimal OS services (e.g., scheduling, time management, inter-process communication and device drivers) for time-critical tasks.

8 DISCUSSION AND OPEN RESEARCH ISSUES

Research in RTS security domain is still relatively new. We now discuss the hurdles faced by the researchers and potential future research directions.

8.1 Vulnerability and Damage Analysis

A better understanding of the vulnerabilities in RTS will enable designers to develop systems with increased security (and hence safety) guarantees. From our study, we find that there exist very few studies on attack mechanisms for RTS. For instance, the majority of the attacks were demonstrated on single core platforms and a further study is required to understand scheduler-level side-channels [44, 45] and effects of Butterfly attacks [46] for multicore RTS. We believe that an important direction for future research is *(i) identifying the risks/vulnerabilities* (for both, single and multicore RTS) and *(ii) studying possible consequences* of successful attacks, i.e., how much damage an adversary can inflict.

8.2 Response and Recovery Mechanisms

A key reason for detecting attacks early is to provide enough information to system operators so that they can respond to and recover from attacks. Research on human-computer interaction can improve the awareness and responses of operators. Systems with real-time requirements often use autonomous, decision-making algorithms for controlling elements in the physical world. In addition to recovery with a human in the loop, there is also a need for automatic recovery (on the detection of an attack). We find that while there exist some research in detection mechanisms for RTS [30–32, 34, 35, 67, 72–75], they do not consider the after-effects of an intrusion. We need further studies to design *autonomous attack detection, isolation and response algorithms* for safety-critical RTS.

8.3 Certification and Regulatory Issues

A distinction of RTS, when compared to conventional IT security, is that software patching and frequent updates are not well suited for critical systems. The addition of new security mechanisms may pose safety concerns (e.g., a power plant was shut down because a computer rebooted after the installation of a patch [99]). Upgrading a safety-critical RTS requires months of advance planning and many layers of certifications [100–102]. We find that the solutions proposed in the literature do not explicitly consider certification/regulatory requirements in their design. *Developing, analyzing and testing of security solutions that comply with certification requirements* is one of the key areas of real-time security research.

8.4 Security for Legacy Systems

Large industrial control systems also have a significant amount of legacy components. Software updates and patching might violate existing safety certifications. For properly securing such legacy-

critical RTS, the underlying security mechanisms must satisfy some minimum performance requirements and the implementation should be well tested and vetted by certification agencies. Our study finds that the majority of the security solutions proposed in the literature are not directly adaptable for legacy RTS (i.e., they are suitable for newer/customized systems). We believe that *(i) understanding of the security requirements (and vulnerabilities)* of legacy systems and *(ii) design security techniques with little or no modification* on existing hardware/software/components are vital areas of research to secure billions of deployed safety-critical systems.

8.5 Security for AI-enabled Next-Generation Systems

While traditionally, application tasks in RTS carry out more straightforward functionalities such as computations related to control loop updates, the advent of modern IoT-specific applications (such as autonomous driving) and the emergence of edge computing bring the use of artificial intelligence (AI) to real-time devices that require the end nodes to process large-scale data. Modern real-time applications often require machine learning (ML)-based inferences for achieving intelligent features such as object recognition, image and video processing, and natural language processing. Any manipulation of ML parameters may lead to misclassification. There is a need to prevent the leakage of critical parameters, including data structures and location in memory, while retaining real-time requirements. Even in the presence of malicious actions, the degree of misclassification should be contained within a permissible and predictable range and the task must not miss its deadlines. There is a lack of *predictable, secure and resilient ML models* that work for resource-constraint real-time devices. The *development of real-time aware, secure and predictable ML/AI-driven system* is an open area that needs concerted research efforts from academia, industry researchers and standardization bodies.

8.6 Availability of Evaluation Platforms

The lack of real-time benchmark programs and evaluation platforms is one of the major challenges in real-time cyber-physical security research especially to perform a sound evaluation. This is partly because of the diversity of real-time applications and software as well as the hardware-dependent nature of cyber-physical platforms. In addition, the majority of safety-critical applications are proprietary in nature and are rarely open-sourced. As a result, existing academic real-time security research is mainly carried out using simulations and/or limited case studies (see Section 4-Section 7) instead of a large-scale experimental evaluation. A better *coordination between industry and academic researchers* can open up opportunities for more open evaluation platforms that can help the designers identify potential vulnerabilities as we discuss next.

8.7 Privacy and Deterrence

In addition to security and safety-related problems, RTS can also have profound privacy implications. RTS end devices can collect private data related to diverse human activities (e.g., location information, driving habits, electricity consumption, biosensor data) at different levels of granularity. Due to the passive manner of collection, end users are largely unaware of the process (e.g., automobile manufacturers are often remotely collecting a variety of driving history data from cars in an effort to increase the reliability of their products) [103]. If the data collected by corporations is exposed to other malicious actors (through a variety of legal or illegal means) it can be detrimental to user privacy. *Deterrence* usually depends on successful legislation, law enforcement and international collaboration for tracking crimes committed across geographical borders [104]. We believe that the *identification of new deterrence mechanisms for the security and privacy of RTS* is a promising area of research.

Table 8. Summary of Related Surveys and This Research

Reference	Focus	Article Type	Remarks
Chai et al. [105]	RTS security	Survey	Lacks qualitative comparison across different schemes
Chen et al. [106]	RTS security	Survey	Limited scope; summary of author's own research (14 papers)
Ravi et al. [107]	Embedded systems security	Survey	Survey of general embedded system security; timing/safety constraints are not considered
Param et al. [108]	Embedded systems security	Survey	Survey of general embedded system security; timing/safety constraints are not considered
Ding et al. [109]	Industrial CPS security	Survey	Survey of industrial CPS security, timing constraints are not considered
Humayed et al. [110]	CPS security	Survey	Survey of CPS security, timing constraints are not considered
Giraldo et al. [111]	CPS security	Survey of surveys	Survey of CPS security surveys, timing constraints are not considered
This work	*RTS security*	*Survey, systematization*	*Through study of RTS security field with a* ***review of 54 papers published in last 27 years****, in-depth analysis of scheduler-level defenses*

9 RELATED SURVEYS

There exist two prior surveys on RTS security. One of the earliest research by Chai et al. [105] presents a short review of RTS security techniques. This is a relatively old work and newer papers (2018 and beyond) are not covered. The survey also lacks qualitative comparisons across different techniques. Our prior survey [106] is limited in scope since it includes our prior work only. In contrast, in this article, we (i) provide an in-depth review and taxonomy of RTS security solutions and (ii) analyze various scheduler-level techniques with a newly introduced *metric* (i.e., attacker's burden).

There also exist prior surveys on security techniques for general embedded systems [107, 108] and broader cyber-physical system (CPS) domains [109–111]. The intrinsic time and safety constraints of RTS distinguish the security requirements/solutions those are proposed for general embedded systems and/or CPS. Our survey complements prior work and provides a holistic overview of the field (see Table 8 for a relative comparison). To the best of our knowledge, this is the first comprehensive effort on systematizing real-time security research.

10 CONCLUSION

Modern real-time embedded systems have evolved in a complex manner due to autonomous systems and cloud-like transparent infrastructure. They are also increasingly facing serious security problems. There is a need for a multi-layered, systematic, engineering approach to secure such critical systems. In this SoK we present a comprehensive review of RTS security issues and analyze various scheduler-level techniques. It is our intent that this systematization will guide future research efforts and ultimately improve the security of this field. We believe that our metric will help the designers to characterize security of systems—both, from the attacks and overhead perspectives.

A APPENDIX

A.1 List of Included Papers

Table 9 lists the papers (and their publication venues) included in this study.

Table 9. Summary of the Included Papers

Publication Venue*	Count	Reference
Major systems/security conferences	6	Bernstein et al. [92], Du et al. [70], Kim et al. [68], Klein et al. [89], Yoon et al. [52, 53]
Major real-time system journals and conferences	17	Bechtel et al. [48], Chen et al. [44], Chen et al. [51], Hasan et al. [32, 33], Jiang et al. [23], Kang et al. [21], Krüger et al. [43], Lesi et al. [29], Lo et al. [72], Mahfouzi et al. [46], Mohan et al. [36], Mukherjee et al. [80], Pellizzoni et al. [37], Walls et al. [69], Xian et al. [81], Yoon et al. [40], Yoon et al. [73]
Major embedded system journals and conferences	5	Abdi et al. [77, 78], Lesi et al. [28], Parmer et al. [90], Xie et al. [19]
Major design automaton conferences	3	Hasan et al. [34, 35], Yoon et al. [74]
Miscellaneous (journals, conferences, book chapters)	22	Abad et al. [67], Abdi et al. [76], Baek et al. [41], Bao et al. [39], Chen et al. [91], Hamad et al. [30], Hao et al. [31], Jiang et al. [38] Jiang et al. [24], Kiszka et al. [94], Lin et al. [20], Liu et al. [45], Mohan et al. [71], Pike et al. [93], Pinto et al. [79], Saadatmand et al. [27], Qiu et al. [22], Vreman et al. [42], Xie et al. [18], Yoon et al. [75], Zhang et al. [25, 26]
Total papers reviewed:	**268**	
Total papers included:	**54**	

*In our study, (i) systems/security conferences include: ACSAC, Asia CCS, CCS, DSN, Euro S&P, HOST, NDSS, OSDI, S&P, Security, SOSP; (ii) real-time venues include: ECRTS, RTAS, RTCSA, RTNS, RTSS, RTS; (iii) embedded system venues include: EMSOFT, ICCPS, TECS, TC, IoT; (iv) design automation conferences include: DAC, DATE. We mark the publication venue as miscellaneous if it does not belong to the above list.

A.2 Experiment Setup

We developed an in-house simulator [62] for our analyses of the attacker's burden introduced in Section 6. Our simulator is platform-independent and written in Python 3.5. We evaluated all four scheduler-level techniques using simulated workloads. The parameters (scheduling policy, priority assignments, schedule duration, number of tasks, periods, execution times) selected in our experiments are identical to those used by the real-time community [32–37, 40, 44]. Due to the different semantics of the techniques, we customized the experiment setup and selected parameters that are meaningful (and realistic) for different approaches, as presented below.

Integrating Cryptography Services (Figure 3). We considered a periodic task with period 250 ms (i.e., sampled at 4 Hz) and requires $m = \{0, 1, 2\}$ cryptographic operations per job (where m is varied as experimental parameter, see Figure 3(b)). We used the values from an earlier work [112] (that measures the execution time of AES encryption for 1-MB messages with different key sizes running on a quad core 1.2-GHz ARM Cortex-A7 platform) and calculated the computing load.

Periodic Monitoring (Figure 4). We considered $[3, 15]$ real-time tasks (with periods $[10, 1000]$ ms) and a single security checking task (with varying periods as an experimental parameter). We assigned rate monotonic priority order (i.e., tasks with shorter periods were assigned higher priorities) [14]. We vary the system load from 2.5% to 97.5% with a step size of 2.5. For a given system load, the individual task load was calculated by using the UUnifast algorithm [113]—since this is a standard technique used by the real-time community. For each load condition, we generated 250 different tasksets. We considered the base period of the security task was 5, 000 ms and decreased the periods (i.e., increased frequency of monitoring) from this base value. We found this value by trial-and-error to ensure that all the generated tasksets were schedulable for demonstration purposes.

State Cleansing Mechanisms (Figure 5). In this setup, we considered 5 real-time tasks and the system load was no more than 50%. The periods of the tasks were selected from $\{25, 40, 50, 100, 125, 200, 250, 500, 1000\}$ ms—this was to ensure that each taskset has a common "hyperperiod"[5] in our experiments. For a given load U_i and period T_i for task τ_i, its (worst-case) execution time was calculated by $\lceil T_i \times U_i \rceil$. The task priorities follow rate monotonic order. The

[5]Hyperperiod is the smallest interval of time (typically defined as the least common multiple of the periods of the tasks) after which the periodic patterns of all the tasks repeat.

attacker's task (lowest priority) was sampled at 20 Hz (50 ms) and we selected the critical (i.e., victim) task period from {100, 125, 200, 250, 500, 1000} ms. For each period value, we generated 100 different taskset configurations. For each configuration, we simulated the schedule for one hyperperiod (since the subsequent schedules will exhibit the same behavior due to the deterministic execution pattern of the system). The flushing overhead was related to the task execution time, i.e., $\lceil \frac{C_c}{3} \rceil$ where C_c is the execution time of the critical task.

Schedule Randomization (Figure 6). In this experiment, we grouped the tasksets by computational loads (i.e., $\{[0.001+0.1\cdot x, 0.1+0.1\cdot x)\times 100\% \mid 0 \leq x \leq 9 \wedge x \in \mathbb{Z}\}$). Each group had 6 subgroups with $n = \{5, 7, 9, 11, 13, 15\}$ tasks and 100 task sets were generated for each subgroup. The generated task sets were tested to be schedulable based on the rate monotonic priority assignment algorithm [14]. Researchers show that an attacker can learn critical information from a schedule [44] within a duration of $10 \cdot T$ where T is the least common multiple of the arrival rates — i.e., periods of the observer task (the attacker's task) and the victim task (the task under attack). We, therefore, set this value as simulation duration in our experiments. We selected observer task and the victim task with index $(\lfloor \frac{n}{3} \rfloor + 1)$ and $(n - \lfloor \frac{n}{3} \rfloor)$, respectively, in a taskset of n tasks (indexed from 1 to n where a larger index implies higher priority).

A.3 Impact of Periodic Monitoring with Different System Loads

We also measure total computing load (y-axis in Figure 8) while varying frequency (x-axis) for three scenarios: (i) low (total load less than 30%), (ii) medium (30%–50%), and (iii) high (more than 70%). The red dotted line shows maximum (feasible) load (i.e., 100% processor utilization) and shaded regions indicate overloaded system. As we see, allowing unfettered execution for the security tasks (especially for medium-to-high utilization scenarios) can add a significant load to the system and will break real-time requirements.

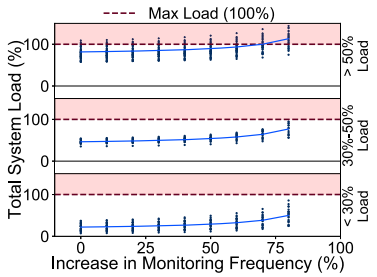


Fig. 8. Tradeoff between monitoring frequency and computing load: unfettered, frequent execution for the monitoring task increases computing load significantly and can break real-time requirements.

REFERENCES

[1] Jerome H Saltzer and Michael D Schroeder. 1975. The protection of information in computer systems. *Proc. of the IEEE* 63, 9 (1975), 1278–1308.

[2] Aini Hussain, M. A, Hannan, Azah Mohamed, Hilmi Sanusi, and A. K. Ariffin. 2006. Vehicle crash analysis for airbag deployment decision. *Int. J. of Auto. Tech.* 7, 2 (2006), 179–185.

[3] Nicolas Falliere, Liam O. Murchu, and Eric Chien. 2011. W32. stuxnet dossier. *White Paper, Symantec Corp., Security Response* 5 (2011), 6.

[4] Robert M. Lee, Michael J. Assante, and Tim Conway. 2016. Analysis of the cyber attack on the Ukrainian power grid. *SANS Industrial Control Systems* (2016).

[5] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. 2010. Experimental security analysis of a modern automobile. In *IEEE S&P*. 447–462.

[6] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. 2011. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Sec. Symp.*
[7] Shane S Clark and Kevin Fu. 2011. Recent results in computer security for medical devices. In *MobiHealth*. 111–118.
[8] Joon Son and Alves-Foss. 2006. Covert timing channel analysis of rate monotonic real-time scheduling algorithm in MLS systems. In *IEEE Inf. Ass. Wor.* 361–368.
[9] Hugo Teso. 2013. Aircraft hacking: Practical aero series. In *HITB Sec. Conf.*
[10] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J. Wellings. 1993. Applying new scheduling theory to static priority pre-emptive scheduling. *SE Journal* 8, 5 (1993), 284–292.
[11] Enrico Bini and Giorgio C. Buttazzo. 2004. Schedulability analysis of periodic fixed priority systems. *IEEE TC* 53, 11 (2004), 1462–1473.
[12] Robert I. Davis and Alan Burns. 2011. A survey of hard real-time scheduling for multiprocessor systems. *ACM CSUR* 43, 4, Article 35 (2011), 35:1–35:44 pages.
[13] Luotao Fu and Robert Schwebel. Real-time Linux wiki. https://rt.wiki.kernel.org/index.php/rt_preempt_howto. ([n. d.]). [Online].
[14] Chung Laung Liu and James W. Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *JACM* 20, 1 (1973), 46–61.
[15] Alan Burns and Robert I. Davis. 2018. A survey of research into mixed criticality systems. *ACM CSUR* 50, 6 (2018), 82.
[16] Lui Sha. 2001. Using simplicity to control complexity. *IEEE Software* 18, 4 (2001), 20–28.
[17] Sandro Pinto and Nuno Santos. 2019. Demystifying ARM TrustZone: A comprehensive survey. *ACM CSUR* 51, 6 (2019), 130.
[18] Tao Xie, Andrew Sung, and Xiao Qin. 2005. Dynamic task scheduling with security awareness in real-time systems. In *IEEE IPDPS*. IEEE, 1–8.
[19] Tao Xie and Xiao Qin. 2007. Improving security for periodic tasks in embedded systems through scheduling. *ACM TECS* 6, 3 (2007), 20.
[20] Man Lin, Li Xu, Laurence T. Yang, Xiao Qin, Nenggan Zheng, Zhaohui Wu, and Meikang Qiu. 2009. Static security optimization for real-time systems. *IEEE Trans. on Indust. Info.* 5, 1 (2009), 22–37.
[21] Kyoung-Don Kang and Sang H. Son. 2006. Systematic security and timeliness tradeoffs in real-time embedded systems. In *IEEE RTCSA*. 183–189.
[22] Meikang Qiu, Lei Zhang, Zhong Ming, Zhi Chen, Xiao Qin, and Laurence T. Yang. 2013. Security-aware optimization for ubiquitous computing systems with SEAT graph approach. *J. of Comp. and Sys. Sci.* 79, 5 (2013), 518–529.
[23] Wei Jiang, Ke Jiang, and Yue Ma. 2012. Resource allocation of security-critical tasks with statistically guaranteed energy constraint. In *IEEE RTCSA*. 330–339.
[24] Wei Jiang, Ke Jiang, Xia Zhang, and Yue Ma. 2015. Energy optimization of security-critical real-time applications with guaranteed security protection. *J. of Sys. Arch.* 61, 7 (2015), 282–292.
[25] Xia Zhang, Jinyu Zhan, Wei Jiang, Yue Ma, and Ke Jiang. 2013. Design optimization of security-sensitive mixed-criticality real-time embedded systems. In *IEEE ReTiMiCS*.
[26] Xia Zhang, Jinyu Zhan, Wei Jiang, and Yue Ma. 2013. A vulnerability optimization method for security-critical real-time systems. In *IEEE NAS*. 215–221.
[27] Mehrdad Saadatmand, Antonio Cicchetti, and Mikael Sjödin. 2012. Design of adaptive security mechanisms for real-time embedded systems. In *USENIX ESSoS*. 121–134.
[28] Vuk Lesi, Ilija Jovanov, and Miroslav Pajic. 2017. Security-aware scheduling of embedded control tasks. *ACM TECS* 16 (2017), 188:1–188:21.
[29] Vuk Lesi, Ilija Jovanov, and Miroslav Pajic. 2017. Network scheduling for secure cyber-physical systems. In *IEEE RTSS*. 45–55.
[30] Mohammad Hamad, Zain AH Hammadeh, Selma Saidi, Vassilis Prevelakis, and Rolf Ernst. 2018. Prediction of abnormal temporal behavior in real-time systems. In *ACM SAC*. 359–367.
[31] Xiaochen Hao, Mingsong Lv, Jiesheng Zheng, Zhengkui Zhang, and Wang Yi. 2019. Integrating cyber-attack defense techniques into real-time cyber-physical systems. In *IEEE ICCD*. 237–245.
[32] Monowar Hasan, Sibin Mohan, Rakesh B. Bobba, and Rodolfo Pellizzoni. 2016. Exploring opportunistic execution for integrating security into legacy hard real-time systems. In *IEEE RTSS*. 123–134.
[33] Monowar Hasan, Sibin Mohan, Rodolfo Pellizzoni, and Rakesh B. Bobba. 2017. Contego: An adaptive framework for integrating security tasks in real-time systems. In *Euromicro ECRTS*. 23:1–23:22.
[34] Monowar Hasan, Sibin Mohan, Rodolfo Pellizzoni, and Rakesh B. Bobba. 2018. A design-space exploration for allocating security tasks in multicore real-time systems. In *DATE*. 225–230.
[35] Monowar Hasan, Sibin Mohan, Rodolfo Pellizzoni, and Rakesh B. Bobba. 2020. Period adaptation for continuous security monitoring in multicore systems. In *DATE*.

[36] Sibin Mohan, Man-Ki Yoon, Rodolfo Pellizzoni, and Rakesh B. Bobba. 2014. Real-time systems security through scheduler constraints. In *Euromicro ECRTS*. 129–140.
[37] Rodolfo Pellizzoni, Neda Paryab, Man-Ki Yoon, Stanley Bak, Sibin Mohan, and Rakesh B. Bobba. 2015. A generalized model for preventing information leakage in hard real-time systems. In *IEEE RTAS*. 271–282.
[38] Ke Jiang, Lejla Batina, Petru Eles, and Zebo Peng. 2014. Robustness analysis of real-time scheduling against differential power analysis attacks. In *IEEE ISVLSI*. 450–455.
[39] Chongxi Bao and Ankur Srivastava. 2014. A secure algorithm for task scheduling against side-channel attacks. In *ACM TrustED*. ACM, 3–12.
[40] Man-Ki Yoon, Sibin Mohan, Chien-Ying Chen, and Lui Sha. 2016. TaskShuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems. In *IEEE RTAS*. 1–12.
[41] Hyeongboo Baek and Chang Mook Kang. 2020. Scheduling randomization protocol to improve schedule entropy for multiprocessor real-time systems. *MDPI Symmetry* 12, 5 (2020), 753.
[42] Nils Vreman, Richard Pates, Kristin Krüger, Gerhard Fohler, and Martina Maggio. 2019. Minimizing side-channel attack vulnerability via schedule randomization. In *IEEE CDC*. IEEE, 2928–2933.
[43] Kristin Krüger, Marcus Völp, and Gerhard Fohler. 2018. Vulnerability analysis and mitigation of directed timing inference based attacks on time-triggered systems. In *EUROMICRO ECRTS*, Vol. 106. 22:1–22:17.
[44] Chien-Ying Chen, Sibin Mohan, Rodolfo Pellizzoni, Rakesh B. Bobba, and Negar Kiyavash. 2019. A novel side-channel in real-time schedulers. In *IEEE RTAS*. 90–102.
[45] Songran Liu, Nan Guan, Dong Ji, Weichen Liu, Xue Liu, and Wang Yi. 2019. Leaking your engine speed by spectrum analysis of real-time scheduling sequences. *J. of Sys. Arch.* 97 (2019), 455–466.
[46] Rouhollah Mahfouzi, Amir Aminifar, Soheil Samii, Mathias Payer, Petru Eles, and Zebo Peng. 2019. Butterfly attack: Adversarial manipulation of temporal properties of cyber-physical systems. In *IEEE RTSS*. 93–106.
[47] David Kroft. 1981. Lockup-free instruction fetch/prefetch cache organization. In *IEEE ISCA*. 81–87.
[48] Michael Bechtel and Heechul Yun. 2019. Denial-of-service attacks on shared cache in multicore: Analysis and prevention. In *IEEE RTAS*. 357–367.
[49] Enea OSE: High-performance, POSIX compatible, multicore real-time operating system. https://www.enea.com/globalassets/downloads/operating-systems/enea-ose/datasheet-enea-ose.pdf. ([n. d.]).
[50] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenstrom. 2008. The worst-case execution-time problem-overview of methods and survey of tools. *ACM TECS* 7, 3 (2008), 36.
[51] Jiyang Chen, Tomasz Kloda, Ayoosh Bansal, Rohan Tabish, Chien-Ying Chen, Bo Liu, Sibin Mohan, Marco Caccamo, and Lui Sha. 2021. SchedGuard: Protecting against schedule leaks using Linux containers. In *IEEE RTAS*. 14–26.
[52] Man-Ki Yoon, Mengqi Liu, Hao Chen, Jung-Eun Kim, and Zhong Shao. 2021. Blinder: Partition-oblivious hierarchical scheduling. In *USENIX Securit*.
[53] Man-Ki Yoon, Jung-Eun Kim, Richard Bradford, and Zhong Shao. 2022. TimeDice: Schedulability-preserving priority inversion for mitigating covert timing channels between real-time partitions. In *IEEE/IFIP DSN*. 453–465. DOI: http://dx.doi.org/10.1109/DSN53405.2022.00052
[54] Paul C, Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *IACR CRYPTO*. 104–113.
[55] John Demme, Robert Martin, Adam Waksman, and Simha Sethumadhavan. 2012. Side-channel vulnerability factor: A metric for measuring information leakage. In *IEEE ISCA*. 106–117.
[56] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. 2011. Introduction to differential power analysis. *J. of Cryp. Eng.* 1, 1 (2011), 5–27.
[57] Saowanee Saewong, Ragunathan (Raj) Rajkumar, John P. Lehoczky, and Mark H. Klein. 2002. Analysis of hierarchical fixed-priority scheduling. In *Euromicro ECRTS*. 173–181.
[58] Rob Davis and Alan Burns. 2008. An investigation into server parameter selection for hierarchical fixed priority pre-emptive systems. In *IEEE RTNS*.
[59] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. 1990. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. on Comp.* 39, 9 (1990), 1175–1185.
[60] Hermann Kopetz. 1991. Event-triggered versus time-triggered real-time systems. In *Op. Sys. of the 90s and Bey.* Springer, 86–101.
[61] Jean-Philippe Aumasson. 2017. *Serious Cryptography: A Practical Introduction to Modern Encryption.* No Starch Press.
[62] Link removed due to anonymity requirements. ([n. d.]).
[63] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. 2009. Lest we remember: Cold-boot attacks on encryption keys. *Comm. of the ACM* 52, 5 (2009), 91–98.

[64] Sibin Mohan. 2008. Worst-case execution time analysis of security policies for deeply embedded real-time systems. *ACM SIGBED Review* 5, 1 (2008), 8.
[65] W-M Hu. 1992. Lattice scheduling and covert channels. In *IEEE S&P*. 52–61.
[66] Man-Ki Yoon, Jung-Eun Kim, Richard Bradford, and Lui Sha. 2013. Holistic design parameter optimization of multiple periodic resources in hierarchical scheduling. In *DATE*. 1313–1318.
[67] Fardin Abdi, Joel Woude, Yi Lu, Stanley Bak, Marco Caccamo, Lui Sha, Renato Mancuso, and Sibin Mohan. 2013. On-chip control flow integrity check for real time embedded systems. In *IEEE CPSNA*. 26–31.
[68] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. 2018. Securing real-time microcontroller systems through customized memory view switching.. In *NDSS*.
[69] Robert J. Walls, Nicholas F. Brown, Thomas Le Baron, Craig A. Shue, Hamed Okhravi, and Bryan C. Ward. 2019. Control-flow integrity for real-time embedded systems. In *Euromicro ECRTS*. 2:1–2:24.
[70] Yufei Du, Zhuojia Shen, Komail Dharsee, Jie Zhou, Robert J. Walls, and John Criswell. 2022. Holistic control-flow protection on real-time embedded systems with kage. In *USENIX Security*.
[71] Sibin Mohan, Stanley Bak, Emiliano Betti, Heechul Yun, Lui Sha, and Marco Caccamo. 2013. S3A: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems. In *ACM International Conference on High Confidence Networked Systems*. ACM, 65–74.
[72] Daniel Lo, Mohamed Ismail, Tao Chen, and G. Edward Suh. 2014. Slack-aware opportunistic monitoring for real-time systems. In *IEEE RTAS*. 203–214.
[73] 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 *IEEE RTAS*. 21–32.
[74] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, and Lui Sha. 2015. Memory heat map: Anomaly detection in real-time embedded systems using memory behavior. In *ACM/EDAC/IEEE DAC*. 1–6.
[75] 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 *ACM/IEEE IoTDI*. 191–196.
[76] Fardin Abdi, Monowar Hasan, Sibin Mohan, Disha Agarwal, and Marco Caccamo. 2016. ReSecure: A restart-based security protocol for tightly actuated hard real-time systems. In *IEEE CERTS*. 47–54.
[77] Fardin Abdi, Chien-Ying Chen, Monowar Hasan, Songran Liu, Sibin Mohan, and Marco Caccamo. 2018. Guaranteed physical security with restart-based design for cyber-physical systems. In *ACM/IEEE ICCPS*. 10–21.
[78] Fardin Abdi, Chien-Ying Chen, Monowar Hasan, Songran Liu, Sibin Mohan, and Marco Caccamo. 2018. Preserving physical safety under cyber attacks. *IEEE IoT J.* 6, 4 (2018), 6285–6300.
[79] Sandro Pinto, Daniel Oliveira, Jorge Pereira, Jorge Cabral, and Adriano Tavares. 2015. FreeTEE: When real-time and security meet. In *IEEE ETFA*. 1–4.
[80] Anway Mukherjee, Tanmaya Mishra, Thidapat Chantem, Nathan Fisher, and Ryan Gerdes. 2019. Optimized trusted execution for hard real-time applications on COTS processors. In *ACM RTNS*. 50–60.
[81] Yecheng Xiang, Yidi Wang, Hyunjong Choi, Mohsen Karimi, and Hyoseung Kim. 2021. AegisDNN: Dependable and timely execution of DNN tasks with SGX. In *IEEE (RTSS*. 68–81. DOI: http://dx.doi.org/10.1109/RTSS52674.2021.00018
[82] Xue Liu, Qixin Wang, Sathish Gopalakrishnan, Wenbo He, Lui Sha, Hui Ding, and Kihwal Lee. 2008. ORTEGA: An efficient and flexible online fault tolerance architecture for real-time control systems. *IEEE Trans. Ind. Inf.* 4, 4 (2008), 213–224.
[83] X. Wang, N. Hovakimyan, and L. Sha. 2013. L1Simplex: Fault-tolerant control of cyber-physical systems. In *ACM/IEEE ICCPS*. 41–50.
[84] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *IACR Crypt. ePrint Arch.* 086 (2016), 1–118.
[85] Se Won Kim, Chiyoung Lee, MooWoong Jeon, Hae Young Kwon, Hyun Woo Lee, and Chuck Yoo. 2013. Secure device access for automotive software. In *IEEE ICCVE*. 177–181.
[86] Sandro Pinto, Jorge Pereira, Tiago Gomes, Adriano Tavares, and Jorge Cabral. 2017. LTZVisor: TrustZone is the key. In *Euromicro ECRTS 2017*. 4:1–4:22.
[87] José Martins, João Alves, Jorge Cabral, Adriano Tavares, and Sandro Pinto. 2017. μRTZVisor: A secure and safe real-time hypervisor. *MDPI Electronics* 6, 4 (2017), 93.
[88] S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares. 2019. Virtualization on TrustZone-enabled microcontrollers? Voilà!. In *IEEE RTAS*. 293–304.
[89] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. SeL4: Formal verification of an OS kernel. In *ACM SOSP*. 207–220.
[90] Gabriel Parmer and Richard West. 2013. Predictable and configurable component-based scheduling in the composite OS. *ACM TECS* 13, 1s (2013), 1–26.
[91] Hui Chen and ShiPing Yang. 2011. Research on ultra-dependable embedded real time operating system. In *2011 IEEE/ACM GreenCom*. 144–151.

[92] Mary M. Bernstein and Chulsoo Kim. 1994. AOS: An avionics operating system for multi-level secure real-time environments. In *ACSA ACSAC*. 236–245.
[93] Lee Pike, Pat Hickey, Trevor Elliott, Eric Mertens, and Aaron Tomb. 2016. TrackOS: A security-aware real-time operating system. In *LNCS RV*. 302–317.
[94] Jan Kiszka and Bernardo Wagner. 2007. Modelling security risks in real-time operating systems. In *IEEE INDIN*, Vol. 1. IEEE, 125–130.
[95] H. Neugass, G. Espin, H. Nunoe, R. Thomas, and D. Wilner. 1991. VxWorks: An interactive development environment and real-time kernel for gmicro. In *IEEE TRONSHOW*. 196–207.
[96] Frank Kolnick. 1998. The QNX 4 real-time operating system. *Basis Comp. Sys. Inc.* (1998).
[97] 2014. *Security Profile For VxWorks*. Technical Report. Wind River. [Online]. Available: https://tinyurl.com/vxworkssec.
[98] 2015. *QNX OS for Security*. Technical Report. QNX Software Systems. [Online]. Available: https://tinyurl.com/qnxsecurity.
[99] Brian Krebs. 2008. Cyber incident blamed for nuclear power plant shutdown. *Washington Post* 5 (2008).
[100] Andrew J. Kornecki and Janusz Zalewski. 2010. Hardware certification for real-time safety-critical systems: State of the art. *Elsevier Ann. Rev. in Control* 34, 1 (2010), 163–174.
[101] Andrew Kornecki and Janusz Zalewski. 2009. Certification of software for real-time safety-critical systems: State of the art. *Springer Inn. in Sys. & Soft. Eng.* 5, 2 (2009), 149–161.
[102] Andrew J. Kornecki. 2007. *Software Development Tools for Safety-Critical, Real-Time Systems Handbook*. Aviation R&D, FAA.
[103] Alvaro Cardenas. 2019. Cyber-physical systems security knowledge area – Issue 1.0. *CYBOK* (2019).
[104] Alvaro Cardenas, Saurabh Amin, Bruno Sinopoli, Annarita Giani, Adrian Perrig, and Shankar Sastry. 2009. Challenges for securing cyber physical systems. In *Wksh. on Fut. Dir. in Cyber-Phy. Sys. Sec.*, Vol. 5.
[105] Hongxia Chai, Gongxuan Zhang, Junlong Zhou, Jin Sun, Longxia Huang, and Tian Wang. 2019. A short review of security-aware techniques in real-time embedded systems. *J. of Cir., Sys. and Comp.* 28, 02 (2019).
[106] Chien-Ying Chen, Monowar Hasan, and Sibin Mohan. 2018. Securing real-time Internet-of-Things. *Sensors* 18, 12 (2018).
[107] Srivaths Ravi, Anand Raghunathan, Paul Kocher, and Sunil Hattangady. 2004. Security in embedded systems: Design challenges. *ACM TECS* 3, 3 (Aug. 2004), 461–491. http://dx.doi.org/10.1145/1015047.1015049
[108] Sri Parameswaran and Tilman Wolf. 2008. Embedded systems security-an overview. *Des. Aut. for Emb. Sys.* 12, 3 (2008), 173–183.
[109] Derui Ding, Qing-Long Han, Yang Xiang, Xiaohua Ge, and Xian-Ming Zhang. 2018. A survey on security control and attack detection for industrial cyber-physical systems. *Neurocomputing* 275 (2018), 1674–1683.
[110] A. Humayed, J. Lin, F. Li, and B. Luo. 2017. Cyber-physical systems security – a survey. *IEEE IoT J.* 4, 6 (2017), 1802–1831.
[111] Jairo Giraldo, Esha Sarkar, Alvaro A. Cardenas, Michail Maniatakos, and Murat Kantarcioglu. 2017. Security and privacy in cyber-physical systems: A survey of surveys. *IEEE Des. & Test* 34, 4 (2017), 7–17.
[112] Daniel A. F. Saraiva, Valderi Reis Quietinho Leithardt, Diandre de Paula, André Sales Mendes, Gabriel Villarrubia González, and Paul Crocker. 2019. PRISEC: Comparison of symmetric key algorithms for IoT devices. *MDPI Sensors* 19, 19 (2019), 4312.
[113] Enrico Bini and Giorgio C. Buttazzo. 2005. Measuring the performance of schedulability tests. *RTS Journal* 30, 1-2 (2005), 129–154.

Received 13 February 2024; accepted 15 February 2024