MDPI

Article

Securing Real-Time Internet-of-Things

Chien-Ying Chen†, Monowar Hasan†* and Sibin Mohan

Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, 61820, USA; {cchen140, mhasan11, sibin}@illinois.edu

* Correspondence: mhasan11@illinois.edu; Tel.: +1-217-819-6783

† These authors contributed equally to this work.

Version June 24, 2019 submitted to Sensors; Typeset by LaTeX using class file mdpi.cls

Abstract: Modern embedded and cyber-physical systems are ubiquitous. A large number of critical cyber-physical systems have real-time requirements (*e.g.,* avionics, automobiles, power grids, manufacturing systems, industrial control systems, *etc.*). Recent developments and new functionality requires real-time embedded devices to be connected to the Internet. This gives rise to the real-time Internet-of-things (RT-IoT) that promises a better user experience through stronger connectivity and efficient use of next-generation embedded devices. However RT-IoT are also increasingly becoming targets for cyber-attacks which is exacerbated by this increased connectivity. This paper gives an introduction to RT-IoT systems, an outlook of current approaches and possible research challenges towards secure RT-IoT frameworks.

Keywords: Security, Real-time systems, Internet-of-things, Survey

1. Introduction

Nowadays smart embedded devices (*e.g.,* surveillance cameras, home automation systems, smart TVs, in-vehicle infotainment systems, *etc.*) are connected to the Internet – this rise in the Internet-of-things (IoT) links together devices/applications that were previously isolated. On the other hand, embedded devices with *real-time properties* (*e.g.,* strict *timing* and *safety* requirements) require interaction between cyber and physical worlds. These devices are used to monitor and control physical systems and processes in many domains, *e.g.,* manned and unmanned vehicles including aircraft, spacecraft, unmanned aerial vehicles (UAVs), self-driving cars; critical infrastructures; process control systems in industrial plants; smart technologies (*e.g.,* electric vehicles, medical devices, *etc.*) to name just a few. Given the drive towards remote monitoring and control, these devices are being increasingly interconnected, often via the Internet, giving rise to the *Real-Time Internet-of-things (RT-IoT)*. Since many of these systems have to meet stringent *safety* and *timing* requirements, any problems that deter from the normal operation of such systems could result in damage to the system, the environment or *pose a threat to human safety*. The drive towards remote monitoring and control facilitated by the growth of the Internet, the rise in the use of commercial-off-the-shelf (COTS) components, standardized communication protocols and the high value of these systems to adversaries are making cyber-security a design priority for such systems. Security breaches are not uncommon in critical IoT applications, especially considering the recent spate of IoT-centric attacks (*e.g.,* the Marai botnet, attacks on the Dyn DNS provider, DoS attacks from IoT devices [1,2]) as well as others centered on safety-critical systems (*e.g.,* Stuxnet [3], BlackEnergy [4], attack demonstrations by researchers on automobiles [5,6] and medical devices [7].) Successful cyber attacks against such systems could lead to problems more serious than just loss of data or availability because of their critical nature [6,8]. Attacks on one or more of these types of systems can have catastrophic results, leading to loss of life or injury to humans, negative impacts on the system and even the environment.

Enabling security in RT-IoT is often more challenging than generic IoT systems due to the additional real-time constraints. The focus of this paper is to introduce the properties/constraints and security threats for RT-IoT (Sections 2-3), summarize security solutions specially designed for such safety-critical domains (Section 4) and highlight the research challenges (Section 5.1). While there exit some surveys [9–13] on security and privacy issues in general-purpose IoT systems, to the best of our knowledge, there is no comprehensive summary in the context of RT-IoT security.

2. Real-Time Internet-of-Things : An Overview

At their core, RT-IoT largely intersect with real-time cyber-physical systems [14]. RT-IoT systems can be considered as a wide inner-connected network, in which nodes can be connected and controlled remotely. Table 1 summarizes some of the common properties/assumptions related to RT-IoT systems. In this section, we intend to outline the elements of RT-IoT as well as the scope of security issues covered in this paper. Figure 1 gives some common scenarios where RT-IoT applications can be implemented.

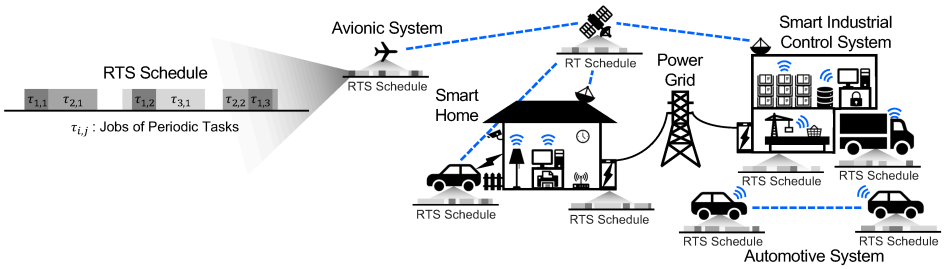


Figure 1. An overview of RT-IoT around everyday living. Dotted lines and radiate symbols indicate the wireless connectivity supported by the devices. Each RT-IoT device executes periodic real-time tasks (say $\tau_{i,j}$ – that denotes the j-th activation of any task τ_i) required for safe operation of the physical system.

2.1. Stringent Timing/Safety Requirements and Resources Constraints

Many RT-IoT devices (*e.g.*, sensors, controllers, UAV, autonomous vehicles, *etc.*) will have severely limited resources (*e.g.*, memory, processor, battery, *etc.*) and often require control tasks to complete within a few milliseconds [15]. RT-IoT nodes, apart from a requirement for functional correctness, require that temporal properties be met as well. These temporal properties are often presented in the form of *deadlines*. The usefulness of results produced by the system drops on the passage of a deadline. If the usefulness drops sharply then we refer to the system as a *hard* real-time system (*e.g.*, avionics, nuclear power plants, anti-lock braking systems in automobiles, *etc.*) and if it drops is a more gradual manner then they are referred to as *soft* real-time systems (*e.g.*, multimedia streaming, automated windshield wipers, *etc.*) [16].

Table 1. Properties of Majority RT-IoT Nodes

• Implemented as a system of periodic/sporadic tasks
• Stringent timing requirements
• Worst-case bounds are known for all loops
• No dynamically loaded or self modified codes
• Recursion is either not used or statically bounded
• Memory and processing power is often limited
• Communication flows with mixed timing criticality

2.2. Heterogeneous Communication Traffic

Many conventional RTS typically consist of several independently operating nodes with limited or no communication capabilities. However with the emergence of RT-IoT, cyber-physical nodes not only communicate over closed industrial communication networks but are also often connected via the Internet. Since most real-time applications would need to trigger events based on specific data conditions, a real-time communication channel with guaranteed QoS (*e.g.,* throughput and data processing requirements, *delay guarantees*, *etc.*) would also be necessary to support such applications [17,18].

Another property of RT-IoT is that they often include traffic flows with mixed criticality, *i.e.,* those with varying degrees of timing (and perhaps even bandwidth and availability) requirements: *(a) high priority/criticality traffic* that is essential for the correct and safe operation of the system; examples could include sensors for closed loop control and actual control commands in avionics, automotive or power grid systems; security systems in home automation *(b) medium criticality traffic* that is critical to the correct operation of the system, but with some tolerances in delays, packet drops, *etc.*; for instance, navigation systems in aircraft, system monitoring traffic in power substations, communication messages exchanged between electric vehicles and power grid or home charging station, traffic related to home automation equipment such as water sprinklers, heating, air conditioning, lighting devices, food preparation appliances *etc.*; *(c) low priority traffic* – essentially all other traffic in the system that does not really need guarantees on delays or bandwidth such as engineering traffic in power substations, multimedia flows in aircraft, notification messages from smart home equipment, *etc.* Typically, in many safety-critical RT-IoT, the properties of all high-priority flows are well known, while the number and properties of other flows could be more dynamic (*e.g.,* consider the on-demand video situation where new flows could arise and old ones stop based on the viewing patterns of passengers in a commercial aircraft).

2.3. Real-Time Scheduling Model

Many such systems are implemented using a set of periodic (*e.g.,* fixed temporal septation between consecutive instances) or sporadic (*e.g.,* the tasks that can make an execution request at *any* time, but with a *minimum* inter-invocation interval) tasks [19, Ch. 1][20]. For instance, a sensor management task that monitors the conveyor belt in a manufacturing system needs to be periodic but the tasks that monitor the arrival of automated cars at traffic intersections are sporadic. Another example is an engine control unit (ECU) in a modern vehicle in which the task that controls the valve in the electronic throttle body (ETB) is periodic while the task that handles commands from the in-vehicle computer is sporadic. Application tasks in the RT-IoT nodes are often designed based on the Liu and Layland model [21,22] that contains a set of tasks, Γ where each task $\tau_i \in \Gamma$ has the parameters: (C_i, T_i, D_i), where C_i is the worst-case execution time (WCET), T_i is the period or minimum inter-arrival time, and D_i is the deadline, with $D_i \leq T_i$.

In the multicore context real-time task scheduling can be viewed as solving an allocation problem (*e.g.,* on which processor a task should execute) depending on design criteria [23]– *e.g., (i) No migration:* tasks are allocated to a processor and no migration is permitted; *(b) Task-level migration:* the jobs of a task may execute on different core; however, each job can only execute on a single core. *(c) Job-level migration:* The jobs of a task migrate to and execute on different cores; however, parallel execution of a job is not permitted.

Schedulability tests [23–26] are used to determine if all tasks in the system meet their respective deadlines. If they do, then the task set is deemed to be *'schedulable'* and the system, *safe*.

2.4. CPU Architectures and System Development Model

Despite the fact that most RT-IoT applications are designed using platforms equipped with a single-core CPU, the trend towards multicore systems can be seen as many COTS devices nowadays

are built on top of a multicore environment [23]. For some specific applications (*e.g.,* avionics systems), there exist regulations that restrict the use of additional cores. In such cases, the additional cores that do not execute real-time or safety critical tasks can be utilized to provide layers of security to the system. We have leveraged the use of multicore platforms in the real-time domain and developed security solutions [27–32] as discussed in Section 4.1.

It is also common that multiple vendors are involved in the development of RT-IoT systems. Such a system is said to be developed under the *multi-vendor development model* [33]. In this model, each vendor designs/controls several separate tasks. Figure 2 demonstrates an electronic control unit (ECU) for an avionics system (on an unmanned aerial vehicle) that uses the multi-vendor development model. While this demonstrative example focuses on the avionics domain other RT-IoT systems (*e.g.,* automotive, home automation, *etc.*) could also be created using a similar model (albeit loosely defined).

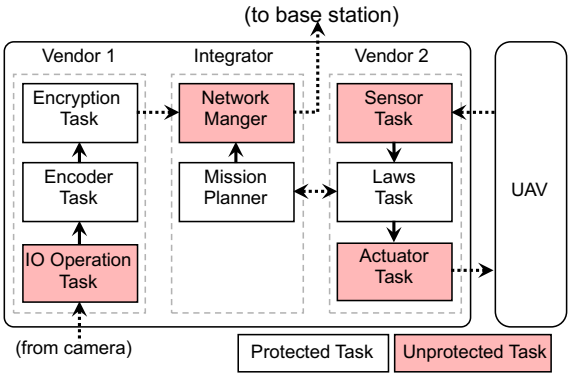


Figure 2. A high-level design of a UAV that exemplifies the multi-vendor development model. In this demonstrative system, three vendors are involved in building the ECU system – *Vendor 1* comprises tasks that process image data from a surveillance camera attached to the ECU; *Vendor 2* is in charge of flight control tasks interacting with the UAV; *Integrator* handles communication between the system and a base station.

3. Security Threats for RT-IoT

RT-IoT systems face threats in various forms depending on the system and the goals of an adversary. In a system developed using vendor-based model, one of the involved vendors can act maliciously. This (potentially unverified/untrusted) vendor could embed malicious functions in its tasks. Bad coding practices could also leave vulnerabilities even if the involved vendors are *not* malicious. leveraging such system vulnerabilities adversaries can execute malicious codes (Section 3.1.1), infer critical system information (Section 3.1.2) and/or perform denial of service attacks (Section 3.1.4). In a system that has network connectivity, the adversary could target the communication interfaces (Section 3.1.3). Due to a lack of authentication in many of these systems, the communication channels could easily be intercepted and forged.

3.1. Attacks on RT-IoT

We classify the attack methodologies on RTS based on the control over computational processes and the functional objective of the attack. One way to acquire control over a target system could be the injection of malicious code (*e.g.,* malware) or by reusing legitimate code for malicious purposes (*e.g., code-injection attacks*). Besides, since RT-IoT nodes can communicate over unreliable mediums such as Internet, the system is also vulnerable to *network-level attacks*. Other than trying to aggressively crash the system (*e.g.,* using *DoS attacks*) the adversary may silently lodge itself in the system and extract sensitive information (*e.g., side-channel attacks*). The side-channel attacks are based on observing

properties (*e.g.*, execution time, memory usage patterns, task schedule, power consumption, *etc.*) of the system. This information may later be used by the attacker to launch further attacks. In the rest of this section, we summarize the common attack surfaces for RT-IoT systems.

3.1.1. Integrity Violation with Malicious Code Injection

An intelligent adversary can get a foothold in the system. For example, an adversary may insert a malicious task that respects the real-time guarantees of the system to avoid immediate detection and/or compromise one or more existing real-time tasks. The attacker may use such a task to manipulate sensor inputs and actuator commands (for instance) and/or modify system behavior in undesirable ways. Integrity violation through code injection attacks conceptually consists of two steps [34]. First, the attacker sends instruction snippets (*e.g.*, a valid machine code program) to the device that is then stored somewhere in memory by the software application receiving it. Such instruction snippets are referred to as *gadgets*. In the second step, the attacker triggers a vulnerability in the application software, *i.e.*, real-time OS (RTOS) or task codes to divert the control flow. Since the instruction snippets represents a valid machine code program, when the program execution jumps to the start address of the data, the malicious code is executed. As we illustrate in Section 4 our recent solutions [27–32,35–37] can be used to detect integrity violations through a combination of hardware/software mechanisms.

3.1.2. Side-Channel Attacks

The adversary may learn important information by side or covert-channel attacks [38] by simply lodging themselves in the system and extracting sensitive information. A side-channel attack manipulates previously unknown channels to acquire useful information from the victim. Memory/cache access time [39], power consumption traces [40], schedule preemptions [41], electromagnetic (EM) emanations [42] and temperature [43] *etc.* are examples of some typical side-channels used by attackers. These attack surfaces are particularly applicable to attacking RT-IoT nodes that execute real-time tasks due to the deterministic behaviors in such systems. A demonstrative cache-timing attack is presented in Section 3.2.2 and Section 4.2.1 illustrates our recent approaches [33,44] to mitigate information leakage that used timing-based attacks on storage-channels.

3.1.3. Attacks on Communication Channels

RT-IoT elevates the Internet as the main communication medium between the physical entities. However, Internet, as a insecure communication medium, introduces a variety of vulnerabilities that may put the security and privacy of RT-IoT systems under risk. Threats to communication includes eavesdropping or interception, man-in-the-middle attacks, falsifying, tampering or repudiation of control/information messages [45]. From the perspective of RT-IoT, defending against communication threats is not an easy task. This is because it is challenging to distinguish rogue traffic from the legitimate traffic (especially for the critical/high-priority flows) without degrading the QoS (*e.g.*, bandwidth and end-to-end delay constraints). Threats to communications are usually dealt by integrating cryptographic protection mechanisms [46,47]. However this increases the WCET of the real-time tasks and may require modification of existing schedulers. Many cryptographic operations are also computationally expensive to execute especially on limited resources available in embedded RT-IoT devices. Therefore existing cryptographic approaches may not be a preferable option for many RT-IoT systems. In Section 4.2.3 we illustrate a solution to integrate security mechanisms that can also be used for dealing with communication threats but does not require modification of existing real-time tasks.

3.1.4. Denial-of-Service (DoS) Attacks

Due to resource constraints (*e.g.*, low memory capabilities, limited computation resources, *etc.*) and stringent timing requirements, RT-IoT nodes are vulnerable to DoS attacks. The attacker may take control of the real-time task(s) and perform system-level resource (*e.g.*, CPU, disk, memory, *etc.*) exhaustion. A more severe type of the DoS attack is the distributed denial-of-service (DDoS) attack where a large number of malicious/compromised nodes simultaneously attack the physical plant. In particular, when critical tasks are scheduled to run, an attacker may capture I/O or network ports and perform network-level attacks to tamper with the confidentiality and integrity (*viz.*, safety) of the system. Again the defense mechanisms developed for generic IT or embedded systems do not consider timing, safety and resource constraints of RT-IoT and are not easily adaptable without significant modifications. As described in Section 4.1.4 and 4.2.3, our recent work [32,35–37] may be used to defend against DoS attacks.

But first, in order for those attacks to be successful, *reconnaissance* is one of the early steps that an attacker needs to carry out. We illustrate this in the following (to demonstrate an attack mechanism).

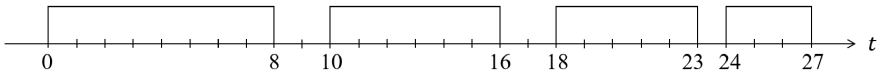
3.2. Reconnaissance: Attack Preparation

Reconnaissance, essentially, is the first step for launching other successful attacks and, at the very least, the attacker gains important information about the system's internals.

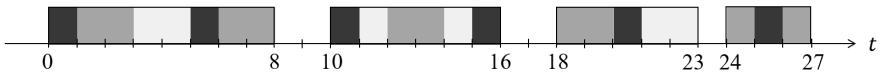
3.2.1. ScheduLeak

In initial work [48], we developed an algorithm, "*ScheduLeak*", to show the feasibility of a *schedule-based side-channel attack targeting real-time embedded systems* with a multi-vendor development model introduced in Section 2.4. The adversary could be one of the vendors or an attacker who compromises a vendor. The *ScheduLeak* algorithm utilizes an *observer task* that has the lowest priority in the victim system to observe *busy intervals*. A "busy interval" is a block of time when one or more tasks are executing – an adversary *cannot* determine *what* tasks are running *when* by just measuring or observing the busy intervals as they are.

The *ScheduLeak* algorithm can be represented as a function $R(\Gamma, W) = J$, where W is a set of observed busy intervals and J is the inferred schedule information that can be used to pinpoint the possible start time of any particular victim task. Such a function is illustrated by Fig. 3. By using the *ScheduLeak* algorithm, an attacker can deconstruct the observed busy periods (with up to 99% success rate if tasks have fixed execution times) into their constituent jobs and precisely pinpoint the instant when a task is scheduled.



(a) Busy intervals observed by attacker's observer task.



(b) Schedules reconstructed by the *ScheduLeak* algorithm.

Figure 3. An example of the schedules produced from a task set of three tasks [48]. The *ScheduLeak* algorithm can recover the precise schedules from the observed busy intervals.

3.2.2. Targeted Attacks

It's worth mentioning that the effectiveness of side-channel attacks is enhanced when combined with the reconnaissance step we just introduced. For example, in the demonstrative ECU system introduced in Section 2.4, let us assume code inserted into *Vendor 2* would like to identify whether the surveillance camera controlled by the I/O Operation Task is enabled. The attacker can launch a *ScheduLeak* algorithm to infer exact start times of the IO Operation Task and carry out a cache-timing attack to gauge cache usage when an I/O Operation Task is scheduled. Figure 4 shows the result of such a cache-timing attack. By launching a *ScheduLeak* attack and knowing when the I/O Operation Task is scheduled to execute, the attacker probes the cache usage only when the task is active. The result indicates that the attacker is able to identify the instant when the camera is on (*i.e.,* when a large amount of data is processed by I/O Operation Task).

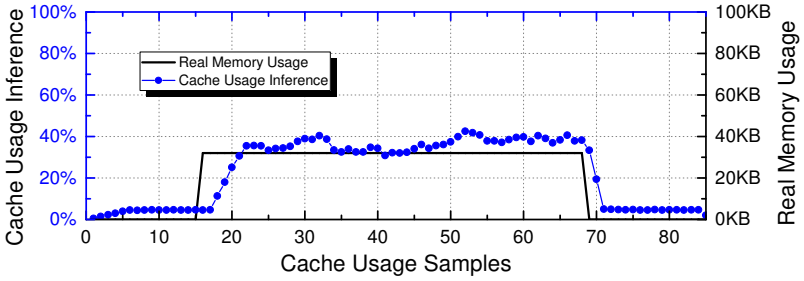


Figure 4. A demonstration of a cache-timing attack [48]. The X-axis is sample points and Y-axis shows both cache usage inference (round dots) and real memory usage amount (the solid line). It shows that a successful cache-timing attack can precisely infer the memory usage of the victim task.

4. Securing RT-IoT: Host-based Approaches

In what follows we summarize our initial attempts to provide security in RT-IoT nodes. We refer to these approaches as *host-based* solutions since they primarily focus on securing an individual RT-IoT node. These approaches can be classified into two major classes: *(i)* solutions that require custom hardware support to provide security and *(ii)* the solutions at the scheduler/software level that do not require any architectural modifications. Table 2 summarizes these security mechanisms for RT-IoT systems.

4.1. Security with Hardware Support

The key idea of providing security *without* compromising the *safety* of the physical system is built on the *Simplex* framework [51]. Simplex is a well-known real-time architecture that utilizes a minimal, verified controller (*e.g., safety controller*) as backup when the complex, high-performance controller (*e.g., complex controller*) is not available or malfunctioning. The goal of the Simplex method is to guarantee that even though a safety-critical system is controlled by a complex controller, the physical system would remain *safe*. We have used the idea of Simplex in the context of RT-IoT security [27–30,32]. The key concept of using Simplex-based architecture for security is to use a minimal simple subsystem (say a trusted core) to *monitor* the properties (*i.e.,* timing behavior [27,28], memory access [29], system call trace [30], behavioral anomalies [32], *etc.*) of an untrusted entity (*e.g.,* monitored core) that is designed for more complex tasks and/or exposed to less secure mediums (*e.g.,* network, Internet, I/O channels, *etc.*).

4.1.1. Secure System Simplex Architecture (S3A)

As mentioned in Section 2, the worst-case, best-case and average-case behaviors for most RT-IoT nodes are calculated ahead of time to ensure that all resource and schedulability requirements will be

Table 2. Summary of Security Solutions for RT-IoT

Reference	Approach	Attack Surface	Overhead/Costs
Simplex-based security [27–31]	Use verified/secure hardware module to monitor system behavior (*e.g.*, timing [28] and execution pattern [27], memory access [29], system call usage [30], control flow [31])	Code injection attacks	Require custom hardware or monitoring unit
Security by platform-level reset [32,49]	Periodically and/or asynchronously (*e.g.*, upon detection of a malicious activity) restart the platform and load an uncompromised OS image	Code injection, side channel and DoS attacks	Extra hardware to ensure safety during periodic/asynchronous restart events
Cache flushing [33,44]	Flush the shared medium (*e.g.*, cache) between the consecutive execution of high-priority (security sensitive) and low-priority (potentially vulnerable) tasks	Side-channel (cache) attacks	Overhead of cache flushing reduces task-set schedulability
Schedule randomization [50]	Randomize the task execution order (*i.e.*, schedule) to reduce the predictability	Side-channel attacks	Extra context switch
Security task integration for legacy RT-IoT [35,37]	Execute monitoring/intrusion detection tasks with a priority lower than real-time task to preserve the real-time task parameters (*e.g.*, period, WCET and execution order)	Code injection, side-channel, DoS and/or communication attacks depending on the what monitoring tasks are used	Running security task with lower priority may cause longer detection time due to high interference (*e.g.*, preemption) from real-time tasks
Adaptive security task integration [36]	Execute monitoring/intrusion detection tasks with a lowest priority most of the time (*e.g.*, during normal system operation) – however change the mode of operation execute with a higher priority (for a limited amount of time) if any anomalous behavior is suspected	Code injection, side-channel, DoS and/or communication attacks depending on the what monitoring tasks are used	False positive detection may cause unnecessary mode switches

met during system operation. *S3A* [27] utilizes this knowledge of deterministic execution profile of the system and use to detect the violation of predicted (*e.g.*, uncompromised) system behavior. *S3A* is one of our earliest efforts to use another (FPGA-based, in this case) trusted hardware component that monitors the behavior (*e.g.*, *execution time* and the *period*) of a real-time control application running on a untrustworthy main system. The goal of this Simplex-based architecture is to detect an infection as quickly as possible and then ensure that the physical system components always remain safe. Using an FPGA-based implementation and considering inverted pendulum (IP) as the physical plant we demonstrated that *S3A* can detect intrusions in less than 6 μs without violating safety requirements of the actual plant.

4.1.2. SecureCore Framework

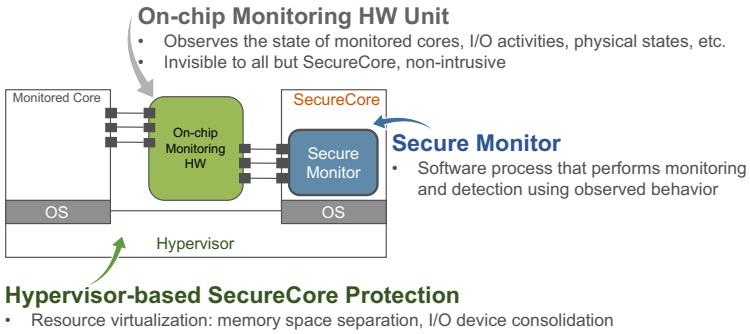


Figure 5. An illustration of *SecureCore* framework. The trusted core is used to monitor the behavior of the complex (and potentially vulnerable) core used for executing application/control tasks.

As illustrated in Fig. 5 the idea of *SecureCore* architecture is to utilize the redundancy in multicore chips to create a trusted entity (*e.g.*, a 'secure' core) that can continuously monitor the system behavior (*e.g.*, code execution pattern [28], memory usage [29], system call trace [30]) of a real-time application on an untrustworthy entity (*e.g.*, monitored core). The SecureCore is protected by hypervisor-based approaches (*e.g.*, by isolating memory regions and I/O device consolidation). The secure monitor (a software process) in the SecureCore uses the on-chip hardware monitoring unit to observe the states (*e.g.*, I/O activities, memory usages, *etc.*) of monitored cores and checks the system behavior at runtime.

The initial *SecureCore* architecture [28] uses a statistical learning-based mechanism for profiling the correct execution behavior of the target system and uses these profiles to detect malicious code execution. Given the probability distribution $P(e)$ of a legitimate execution instance, the secure monitor compares $P(e)$ with a predefined minimum required probability θ — if $P(e)$ is below the threshold probability (*e.g.*, $P(e) < \theta$) the execution instance to is considered as malicious. The *SecureCore* framework is also extended [29] to profile memory behavior (referred to as memory heat map (MHM)) and then detect deviations from the normal memory behavior patterns. MHM represents how many times a particular memory region was accessed during a time interval. We proposed machine learning algorithms to characterize the information contained in the MHMs and then detect deviations from the normal memory behavior patterns. We have also extended *SecureCore* architecture to detect anomalous executions using a distribution of system call frequencies. Specifically we have proposed [30] to use clustering algorithms (*e.g.*, global k-means clustering with the Mahalanobis distance) to learn the legitimate execution contexts (by means of distribution of system call frequencies) of real-time applications and then monitor them at run-time to detect intrusions.

4.1.3. Control Flow Monitoring

We then proposed hardware-based approach for checking the integrity of code flow of real-time tasks [31]. In particular, we add an on-chip control flow monitoring module (OCFMM) with a dedicated memory unit that directly hooks into the processor and tracks the control flow of the tasks. The control flow graph (CFG) of tasks is produced from the program binary and loaded into the OCFMM memory in advance (*e.g.*, during system boot). The detection module inside OCFMM compares the control flow of the running program with the stored one (*e.g.*, CFG profiles that are loaded into the dedicated memory at boot time) during program execution. At run-time (*e.g.*, during execution of a given block) CFG profiles for the next-possible blocks are pre-fetched. The decision module continuously scans the current block and validates the execution flow by comparing the current address of the program counter (PC) against the possible, previously fetched destination

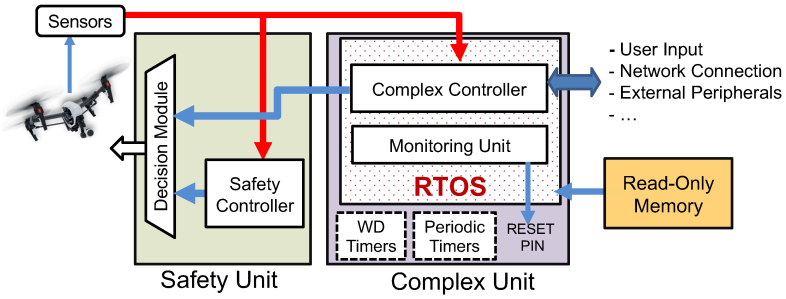


Figure 6. The *ReSecure* framework [32]: safety unit is the bare-metal verified component, complex unit is not verified. The decision module switches between the controllers to provide overall system safety.

addresses. If any mismatch occurs, the detection module raises a *detection flag* that indicates a possible breach.

4.1.4. Security via Platform-level Reset

In traditional computing systems (*e.g.*, servers, smart phones, *etc.*), software problems are often resolved by restarting either the application process or the platform [52,53]. However, unlike those conventional computing systems restart-based recovery mechanisms are not straightforward in RT-IoT due to the real-time constraints as well as interactions of the control system with the physical world (for example, a UAV can quickly be destabilized if its controller restarts). In initial work [32] we proposed a restart-based concept to improve security guarantees for RT-IoT. This Simplex-based framework, that we refer to as *ReSecure*, is specifically designed to improve security of safety-critical RT-IoT systems. In particular, we propose to *restart* the platform periodically/asynchronously and load a fresh image of the applications and OS from a *read-only media* after each reboot with the objective of *wiping out* the intruder or malicious entity. The ReSecure architecture (see Fig. 6) produces a verified system (by using a safety unit) despite the use of an unverified complex controller (*e.g.*, complex unit). OS/Firmwire in complex unit is exposed to external (possible attack) surfaces and can fail. Decision module predicts if the future states are safe. Watchdog (WD) and periodic timers restart the complex unit (and reload OS image from read-only memory) upon fail-stop.

Our primary focus here is to ensure the safety of the system despite the presence of malicious entity. The main idea is that, if we restart the system *frequently enough*, it is less likely that the attacker will have time to re-enter the system and cause meaningful damage (such as data breaches and jeopardizing safety) to the system. After every restart, there will be a predictable down time (during the system reboot), some operational time (before system is compromised again) and some compromised time (until the compromise is detected or the periodic timer expires). The length of each one of the above intervals depends on the type and configuration of the platform, adversary models, complexity of the exploits, *etc.* As a general rule, the effectiveness of the restarting mechanism increases: *(i)* as the time to re-launch the attacks increases, or *(ii)* the time to detect attacks and trigger a restart decreases. We also evaluate the expected lack of availability due to restarts and the expected damage from the attacks/exploits given a certain restart configuration.

In later work [49], we introduced the secure execution interval (SEI) – a period of time after each restart and before the untrusted applications begin to execute, when the execution environment is not yet contaminated and hence security is guaranteed. During SEI, the system executes trusted code to determine the next restart time based on the current discrete state of the physical system. When necessary, a safety controller can override the control of the system (during SEI) to guide the system back to a safe state. In addition, we introduced a root of trust (RoT) – an isolated hardware timer responsible for enforcing the restart process by issuing the restart signal at designated times (computed by the trusted code in SEI). RoT is designed to be programmable only once in each

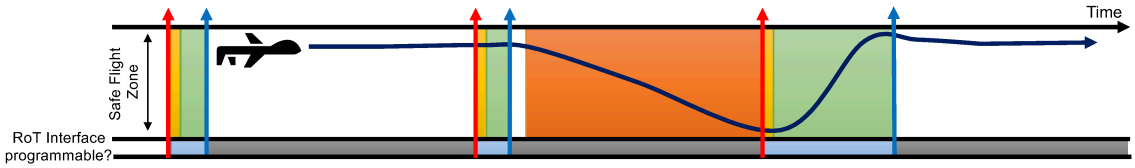


Figure 7. An example of a UAV system operating under the *ReSecure* framework [49]. The black line coming out from the UAV indicates the distance before it gets out of the safe flight zone. The red arrows annotate the triggering of the restarting points by the RoT. The blue arrows annotate the exit of the SEI (and that the next restart time is scheduled in RoT). We use different colors to illustrate the different phases of the system operation – *(a) white*: the main flight controller is in charge and system is not compromised; *(b) yellow*: the system is restarting; *(c) green*: SEI is active, the safety controller is running and the next restart time is being calculated; *(d) orange*: the system is compromised and the adversary is in charge; *(e) blue/gray*: the time spans when the (RoT) interface is available and unavailable, respectively.

execution cycle and only during SEI. Since it is inaccessible outside of SEI and works independently, the triggering of the restart process is not affected even when the system is compromised. An example our framework operating in a UAV system is illustrated in Fig. 7. The UAV operates normally within its safe flight zone and the safety controller does not need to be activated during SEI. Once the attacker compromises the system after the second restart (the orange area), the UAV flying towards its unsafe zone. Before the UAV reaches the unsafe zone, the hardware timer is up in RoT and triggers a restart. The safety controller (in SEI) takes over the control and brings the UAV back to the safe zone. Once the UAV returns to a predefined safe zone threshold, SEI ends and hands the control backs to the applications.

4.2. Security without Architectural Modifications

Despite the fact that architectural modification can improve the security posture of RT-IoT nodes, those approaches require an overall redesign and may not be suitable for systems developed using COTS components. We now review the some of the approaches that we recently proposed to enhance security in RT-IoT without custom hardware support.

4.2.1. Dealing with Side-Channel Attacks

As introduced in Section 3.2.2, we demonstrated that an attacker can carry out a cache-timing attack to indirectly estimate memory usage behavior. It is due to the lack of isolation for shared resources across different tasks in most COTS-based RT-IoT systems. The overlap between tasks happens when the system transitions from one task to another. Therefore, capturing security constraints between tasks becomes essential for preventing side-channel attacks.

In previous work [44], we proposed to integrate security in RT-IoT by introducing techniques to add constraints to tasks scheduled with fixed-priority real-time schedulers. Based on user-defined security levels for each task, the scheduler *flushes shared cache* when the system is transitioning from a high security task (*i.e.*, a task demanding higher confidentiality) to a low security task (*i.e.*, an insecure task potentially compromised). Let us consider the set of security levels for tasks, S, that forms a *total order*. Hence, any two tasks (τ_i, τ_j) may have one of the following two relationships when considering their security levels, $s_i, s_j \in S$: *(i)* $s_i \prec s_j$, meaning that τ_i has higher security level than τ_j or *(ii)* $s_j \prec s_i$.

We proposed the idea of mitigating information leakage among tasks of varying security levels, by *transforming security requirements into constraints on scheduling algorithms*. The approach of modifying or constraining scheduling algorithms is appealing because, *(a)* it is a software based approach and hence easier to deploy compared to hardware based approaches and *(b)* it allows for reconciling the security requirements with real-time or schedulability requirements. Consider

a simple case with two periodic tasks, a high priority task H and a low priority task L scheduled by a fixed-priority scheduling policy. Assume that $s_H \prec s_L$; hence, information from H must not leak to L. These tasks must be scheduled on a single processor, P, so that both deadlines (D_H, D_L) are satisfied. If L (or any part thereof) executes immediately after (any part) or all of H, then L could "leak" data from resources recently used by H. The main intuition is that a penalty must be paid for each shared resource in the system every time tasks switch between security levels. In this case, *the cache must be flushed before a new task is scheduled*. Hence, we proposed the use of an independent task, called the *Flush Task* for this purpose.

In subsequent work [33], we relaxed many of the restrictions (*e.g.*, the requirement of total ordering of security levels) and proposed a *new, more general model* to capture security constraints between tasks in a real-time system. This includes the analysis for the schedulability conditions with both preemptive and non-preemptive tasks. We proposed a constraint named *noleak* to capture whether *unintended information sharing between a pair of tasks must be forbidden*. Using this constraint we can prevent the information leakage via implicitly shared resources. For any two tasks τ_i and τ_j: if $noleak(\tau_i, \tau_j) =$ `True`, then information leakage from τ_i to τ_j must be prevented; if $noleak(\tau_i, \tau_j) =$ `False`, no such constraints need to be enforced. We showed that the system remains schedulable (*e.g.*, all the tasks can meet their deadline) under the proposed constraints without significant performance impact.

4.2.2. Schedule Randomization

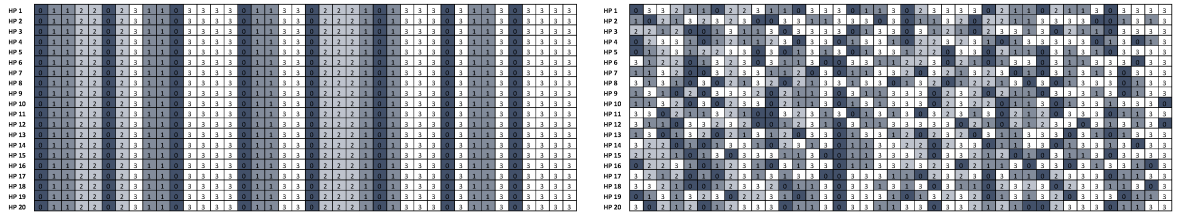
One way to protect a system from certain attacks (*e.g.*, the schedule-based side-channel attack mentioned in Section 3.2.1), is to *randomize the task schedule* to reduce the deterministic observability of periodic RT-IoT applications. By randomizing the task schedules we can enforce non-determinism since every hyper-period[1] will show different order (and timing) of execution for the tasks. Unlike traditional systems, randomizing task schedules in RT-IoT is not straightforward since it leads to priority inversions [54] that, in turn, may cause missed deadlines – hence, putting the safety of the system at risk.

Hence, we proposed *TaskShuffler* [50], a randomization protocol for fixed-priority scheduling algorithm, to achieve such randomness in task schedule. For instance, by picking a random task instead of the one with the highest-priority at each scheduling point, subject to the deadline constraints. The degree of randomness is flexible in *TaskShuffler*. Based on the system's needs, *TaskShuffler* implements the following randomization schemes:

- *Randomization (Task Only)*: This is the most basic form of randomization in contrast to other schemes introduced below. We randomly pick a task to execute whenever a task arrives or finishes its job, *i.e.*, at the scheduling points. The effectiveness against the schedule-based side-channel attack is limited since the busy intervals in this scheme remains the same.
- *Randomization with Idle Time Scheduling*: In addition to the randomness provided in the basic scheme, we include the *idle task* (*e.g.*, the dummy task executed by an RTOS when other real-time tasks are not running) at each scheduling point. It eliminates the periodicity of busy intervals (from hyper-period's point of view). This scheme makes it harder to produce effective results from the schedule-based side-channel attack.
- *Randomization with Idle Time Scheduling and Fine-grained Switching*: To push the randomization to an extreme, one could choose to randomize the schedule every tick. That is, the scheduler will randomly pick a task to execute, subject to the deadline constraints, in every tick interrupt. This way, we gain the most randomness for the schedule. Figure 8 illustrates an instance of the

[1] Hyper-period is the smallest interval of time after which the periodic patterns of all the tasks repeats itself – typically defined as the least common multiple of the periods of the tasks.

randomized schedule for an simple taskset with three tasks. However, it greatly increases the overheads and thus may not be applicable for all use cases.



(a) **(b)**

Figure 8. Examples of the schedule randomization protocol with three tasks: *(a)* vanilla fixed priority scheduling (*e.g.,* schedules without randomization); *(b) TaskShuffler* (fine-grained scheduling with randomizing idle times). Blocks numbered 0 to 2 are the execution of periodic tasks while blocks numbered 3 indicate idle time (*i.e.,* the idle task). The following taskset parameters are used in the illustration: $\tau_0(5,1,5)$, $\tau_1(8,2,8)$, $\tau_2(20,3,20)$ where each task $\tau_i(C_i,T_i,D_i)$, $0 \leq i \leq 2$ is characterized by WCET (C_i), period (T_i) and deadline (D_i). Each row represents a hyperperiod and the figure shows the schedule of 20 hyperperiod. For vanilla scheduling, task schedules are repeating each hyperperiod. In contrast, *TaskShuffler* scrambles the schedule across hyperperiod and thus make it harder to predict a particular task execution instance.

IoT systems with real-time properties are predictable by design. This very determinism can become a vulnerability in the hands of smart adversaries and it becomes easier to carry out adversarial actions such as side-channel attacks [55,56], DoS (making critical resources unavailable at important times) or even the recently developed timing-inference attacks [55]. *TaskShuffler* can reduce the determinism that is visible to external entities while still meeting real-time guarantees. With such randomization, even if an observer is able to capture the exact schedule for a (limited) period of time (for instance, for a few hyperperiods), *TaskShuffler* will schedule tasks in a way that succeeding hyperperiod will show different orders (and timing) of execution for the tasks.

4.2.3. Integrating Security for Legacy RT-IoT

As we have described in Section 3.2.1, an adversary can extract important information while still remaining undetected and it is essential to have a layered defense and integrated resilience against such attacks into the design of RT-IoT. However, any security mechanisms have to *co-exist* with real-time tasks in the system and have to operate without impacting the timing and safety constraints of the control logic. Besides, the embedded nature of these systems limits the availability of computational power (*e.g.,* memory or processor) required for resource-extensive monitoring mechanisms. This creates an apparent tension between security requirements (*e.g.,* having enough cycles for effective monitoring and detection) and the timing and safety requirements. For example, a critical parameter is to determine how often and how long should a monitoring and intrusion detection task execution to be effective but not interfere with real-time control or other safety-critical tasks. While this tension could potentially be addressed for newer systems at design time, this is especially challenging for retrofitting *legacy* systems where the control tasks are already in place and perhaps *cannot be modified*. Any hardware and/or software-level modifications to those legacy system parameters is costly since it will go through several verification and validation steps and may increase system downtime [15]. Most of the security solutions for RT-IoT proposed in literature either require custom hardware [27–32,49,57], modification of the existing schedulers [46,47], extra instrumentations [57] or may need to change the tasks parameters (*e.g.,* execution order and/or run-time) [33,44,50] and therefore *not* suitable for legacy systems. Integrating monitoring and detection tasks for RT-IoT *without* custom hardware support is an open problem.

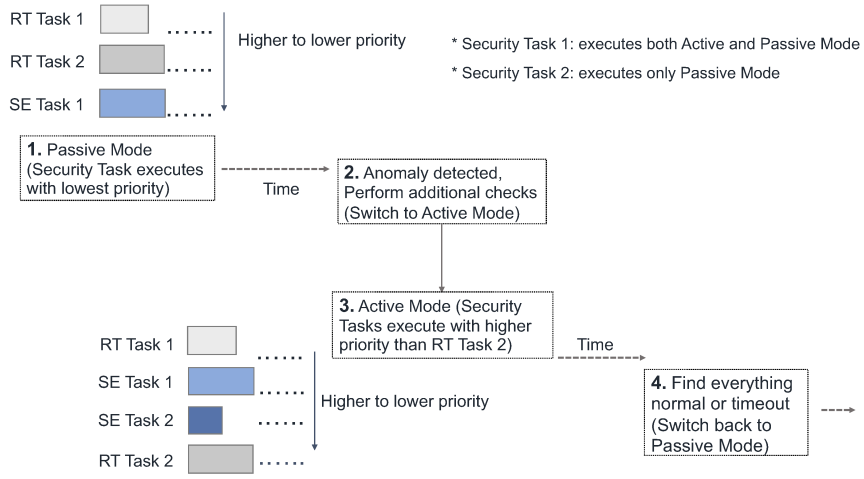


Figure 9. Flow of operations in *Contego* depicting different modes for the security tasks.

Given the tension between security and timing requirements, while integrating security mechanisms into a practical system, finding the *frequency of execution* of the monitoring tasks is an important design parameter that trades security requirements with timing constraints. If the interval between consecutive monitoring events is too large, the adversary may harm the system (and remain undetected) between two invocations of the security task. In contrast, if the security tasks are executed very frequently then it may impact the schedulability of the real-time tasks.

In preliminary work [35] we address the problem of determining the frequency of execution (*e.g.*, periods or inter-monitoring interval) of the security tasks. Our approach to integrate security without perturbing real-time scheduling order is to execute security tasks at a *lower priority* tasks than real-time tasks. We refer this scheme as *opportunistic execution* since the security tasks are only allowed to execute opportunistically only during *slack times* when no other real-time tasks are running.

We propose to measure the security of the system by means of the *achievable periodic monitoring*. Let T_i be the period of the security task that needs to be determined. Our goal here is to minimize the perturbation between the achievable (*i.e.*, unknown) period T_i and the desired (*e.g.*, designer provided) period T_i^{des}. We formulate a constraint optimization problem and develop a polynomial-time solution that allows us to execute security tasks with a frequency closer to the desired values while respecting the temporal constraints of the other real-time tasks.

If the security tasks always execute with lowest priority, they suffer more interference (*i.e.*, preemption from high-priority real-time tasks) and the consequent longer detection time (due to poor response time) will make the security mechanisms less effective. In order to provide *better responsiveness* and increase the effectiveness of monitoring and detection mechanisms, we then proposed a multi-mode model [36]. This framework (called *Contego*) allows the security policies/tasks to execute in different modes in different *modes* (*i.e.*, passive monitoring with lowest priority as well as exhaustive checking with higher priority). By using this approach (see Fig. 9), for instance, security routines can execute opportunistically when the system is deemed to be clean (*i.e.*, not compromised). However if any anomaly or unusual behavior is suspected, the security policy may switch to a *fine-grained checking mode* and execute with higher priority. The security routines may go back to normal mode if: *i*) no anomalous activity is found; or *ii*) the intrusion is detected and malicious entities are removed.

The aforementioned works however are developed for single core systems only – integrating security mechanisms for legacy multicore platforms (where designers have less flexibility for changing system architecture/parameter) is also a challenging problem. In recent work [37] we developed a scheme for multicore RT-IoT and find a suitable assignment of security tasks that ensures they can execute with a frequency close to what a designer expects. We considered a multicore

platform comprised of M identical cores. One fundamental problem while integrating security mechanisms in multicore platforms is to determine *which security tasks will be assigned to which core and executed when*. Although security tasks can execute in any of the M available cores and any period $T_i^{des} \leq T_i \leq T_i^{max}$ is acceptable, the actual task-to-core assignment and the periods of the security tasks are not known apriori. The goal of this scheme therefore is to jointly find the *core-to-task assignment and suitable periods* for the security tasks. However, finding such an assignment is NP-hard due to combinatorial nature of the problem. Therefore we developed a *near-optimal low-complexity solution* (called *HYDRA*) that jointly finds the security tasks' period and core assignments. From our experiment we found that on average *HYDRA* (that distributes security tasks across all available cores) can provide 27.23% faster intrusion detection rate (on a quad core system) compared to the case when the security tasks are allocated a dedicated core for while the real-time tasks are assigned to the remaining cores.

5. Discussion and Research Opportunities

5.1. Securing Legacy RT-IoT Systems

Since most RT-IoT nodes are resource-constrained embedded devices, resource-intensive processing and complex protocols (*e.g.,* heavy cryptographic operations) for securing those systems is unrealistic and may threaten the safety of such systems – for instance a safety-critical task may miss deadline in order to run computation-heavy security tasks. In addition to execution frequency, another important consideration is to determine how quickly can intrusions be detected. Thus *responsiveness vs. schedulability* of critical tasks is another important trade-off. This in itself is a research challenge that needs to be investigated.

So far we have assumed that we are given a set of security tasks and that each security task has a desired frequency of execution for better security coverage. Security tasks so far have been treated as being independent and preemptible. But in practice, as previously discussed some security monitoring may need atomicity or non-preemptive execution. Further, *security tasks may have dependencies where one task depends on the output from one or more other tasks*. For example, an anomaly detection task may depend on the outputs of multiple scanning tasks. Or, the scheduling framework may need to follow certain *precedence constraints* for security tasks. For example, in order to ensure integrity of monitoring security, the security application's own binary may need to be examined first before checking the system binary files. In such cases we can not independently execute the security task and we need to consider the problem of integrating security tasks with dependencies between them. One approach could be use a directed acyclic graph (DAG) to capture the dependencies and constraints among security tasks. In this case, tightness of achievable periodic monitoring described in Section 4.2.3 may no longer be a reasonable metric. Constraints to ensure that the entire DAG is executed often enough should be included and the optimization problem reformulated and evaluated with different metrics.

5.2. Security for Multicore based RT-IoT Platforms

Most of the work [33,35,36,44,48] presented so far has been in the context of single core processors – they are the most common types of processors being used in RT-IoT systems. However, as mentioned earlier, due to increasing computational demands, multi-core processors are becoming increasingly relevant to real-time systems [23,58]. With the increased number of cores, more computation can be packed into a single chip – thus reducing power and weight requirements – both of which might be relevant to many RT-IoT systems. However multicore processors can *increase attack vectors*, especially for side-channel attacks. First, two or more tasks are running together and (most likely) sharing low-level resources (*e.g.,* last level caches). Hence, a task running on one core can snoop on the other – and not just when tasks follow each other. In fact, it has been shown that leakage can occur with a much higher bandwidth in the case of shared resources in multi-core processors

[59]. Second, when tasks execute together, a malicious task can increase the "interference" faced by a critical task – for instance, the malicious task can flood the cache/bus with memory references just when an important task (say, one that computes the control loop) is running. This could cause the critical task to get delayed and *even miss its deadline*. To prevent such problems, designers of the systems need to enforce constraints that protected tasks do not execute simultaneously with unprotected ones on the multi-core chip.

The problem of integrating security tasks into legacy RT-IoT systems is also interesting in the multicore context – perhaps the security tasks can always be running (say on one of the dedicated cores) instead of running opportunistically as is the case for single core systems. Also it may be possible to to take up more cores and execute *fine-grained sanity checks* (*e.g.,* a complete system-wide scan) as it detects malicious activity. Analyzing the impact of integrating security tasks in a multicore legacy RT-IoT is an open problem worth investigating.

5.3. Secure Communication with Timing Constraints

With the rise of RT-IoT, the edge devices are more frequently exchanging control messages and data often with unreliable mediums like the Internet. Therefore, in addition to the host-based approaches [27–31,33,44,50] described earlier, there is a requirement for securing communication channels to ensure authenticity and integrity of control messages. While some of our previous work [32,35] can also be used to deal with network-level attacks, designing a unified framework to protect edge devices as well as communication messages (given the stringent end-to-end delay requirements for high-critical traffics) is still an open problem.

Most safety-critical RT-IoT systems often have separate networks (hardware and software) for each of the different types of flows for safety (and security) reasons. This leads to significant overheads (equipment, management, weight, *etc.*) and also potential for errors/faults and even increased attack surface and vectors. Network-level nondeterminism, *i.e.,* unpredictability in sensor reading, packet delivery/forwarding/processing further complicates the management of RT-IoT systems. Existing protocols, *e.g.,* avionics full-duplex switched Ethernet (AFDX) [60], controller area network (CAN) [61], *etc.* that are in use in many of real-time domains are either proprietary, complex, expensive, require custom hardware or they are also exposed to known vulnerabilities [62].

Given the limitations of existing protocols, leveraging the benefits of *software-defined networking* (SDN) can also be effective for RT-IoT systems. The advantage of using SDN is that it is compatible with COTS components (and thus suitable for legacy RT-IoT systems) and provides a centralized mechanism for developing and managing the system. The global view is useful to ensure QoS (*e.g.,* bandwidth and delay) and enforce security mechanisms (such as remote attestations, secure key/message exchange, remote monitoring, *etc.*). While SDNs provide a global view of the network and high-level management capabilities (including resource allocation), current standards used in traditional SDN (*e.g.,* OpenFlow [63]) do not consider inherent timing and safety-critical nature of the RT-IoT systems. In recent work [64] we tried to address this problem through static flow allocation and routing – we used static path allocation and over-provisioning hardware resources (*e.g.,* dedicating one queue per real-time flow) for meeting the end-to-end delay requirements and providing isolation. This limited the number of flows that could be admitted and resulted in underutilized network resources. Retrofitting the capabilities of SDN in the RT-IoT domain requires further research. We also need mechanisms to *prioritize between flows* (say between the critical real-time flows or even across real-time and non real-time flows) and also schemes for *multiplexing flows* on the same queues in the SDN switches (to improve the efficiency of the network) while still meeting the real-time constraints.

6. Related Work

There exists work that has investigated security in real-time systems [46,47,65]. Many researchers have studied this research area from different aspects. Information leakage via side channels has

been discussed in many works. Kadloor *et al.* [66] and Gong *et al.* [67] introduced analysis and methodology for quantifying side-channel leakage. Kelsey *et al.* [39], Osvik *et al.* [68] and Page *et al.* [69] demonstrated the usability of cache-based side-channels. Son *et al.* [41] and Völp *et al.* [70] examined the exploitation of timing channels in real-time scheduling. Bao *et al.* [71] introduce a scheduling algorithm for soft real-time systems (where some tasks can miss deadlines) and provide a trade-off between thermal side-channel information leakage and the number of deadline misses. Their exists other work [56] that studies the robustness of AES secret keys against differential power analysis (DPA) [72] attacks.

While the work above focuses on exploring vulnerabilities, there exist work that aims to provide security to real-time systems. Ghassami *et al.* [73] and Völp *et al.* [74] proposed techniques to address leakage via shared resources. An online job randomization scheme [75] is proposed by Krüger *et al.* for time-triggered real-time systems. Xie *et al.* [46] and Lin *et al.* [47] presented security in real-time systems by encrypting communication messages. Similar to the hardware-assisted security mechanisms like ours (*e.g., S3A*, *SecureCore*, *ReSecure*, *etc.*) researchers also propose architectural frameworks [57] that dynamically utilizes slack times (*e.g.,* the time instance when no other real-time tasks is executing) for run-time monitoring. There exists recent work [76,77] where authors proposed schemes to secure systems from man-in-the-middle attacks, where an attacker can compromise communication between system sensors and controllers.

Some recent work has raised security awareness in IoT applications [10,13,78–80]. Some researchers aim to add security properties to IoT. Pacheco *et al.* [81] introduced a security framework that offers security solutions with smart infrastructures. Kuusijärvi *et al.* [82] proposed to mitigate IoT security threats with using trusted networks. Those work primarily focuses on generic IoT applications, and do not consider the additional real-time constraints required for RT-IoT systems.

7. Conclusion

The sophistication of recent attacks on RT-IoT requires rethinking of security solutions for such systems. The goal of this paper is to raise the awareness of real-time security and bridge missing gaps in the current IoT context – securing the IoT systems with real-time constraints. The techniques and methodology presented here vary from different perspectives – from hardware-assisted security to scheduler-level as well as those for legacy systems. The designers of the systems and research community will now be able to integrate and develop upon these frameworks required to secure safety-critical RT-IoT systems. We believe that the real-time and IoT worlds are closely connected and will become inseparable in the near future.

Acknowledgement

The multiple security frameworks presented in this paper is the result of a team effort. The authors would like to acknowledge the contributions from our team members and collaborators: Man-Ki Yoon, Fardin Abdi Taghi Abad, Rodolfo Pellizzoni, Rakesh B Bobba, Lui Sha and Marco Caccamo. This work is supported in part by grants from the National Science Foundation (NSF CNS 14-23334, NSF CPS 1544901 and NSF SaTC 1718952), ONR (N00014-13-1-0707) and the Dept. of Energy. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of sponsors.

References

1. Kolias, C.; Kambourakis, G.; Stavrou, A.; Voas, J. DDoS in the IoT: Mirai and other botnets. *Computer* **2017**, *50*, 80–84.
2. Westling, J. Future of the Internet of Things in Mission Critical Applications **2016**.
3. Falliere, N.; Murchu, L.O.; Chien, E. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response* **2011**, *5*, 6.

4. Lee, R.M.; Assante, M.J.; Conway, T. Analysis of the cyber attack on the Ukrainian power grid. *SANS Industrial Control Systems* **2016**.
5. Koscher, K.; Czeskis, A.; Roesner, F.; Patel, S.; Kohno, T.; Checkoway, S.; McCoy, D.; Kantor, B.; Anderson, D.; Shacham, H.; others. Experimental security analysis of a modern automobile. IEEE S&P, 2010, pp. 447–462.
6. Checkoway, S.; McCoy, D.; Kantor, B.; Anderson, D.; Shacham, H.; Savage, S.; Koscher, K.; Czeskis, A.; Roesner, F.; Kohno, T.; others. Comprehensive Experimental Analyses of Automotive Attack Surfaces. USENIX Sec. Symp., 2011.
7. Clark, S.S.; Fu, K. Recent results in computer security for medical devices. MobiHealth, 2011, pp. 111–118.
8. Abrams, M.; Weiss, J. Malicious control system cyber security attack case study–Maroochy Water Services, Australia. *McLean, VA: The MITRE Corporation* **2008**.
9. Sadeghi, A.R.; Wachsmann, C.; Waidner, M. Security and privacy challenges in industrial Internet of things. ACM/EDAC/IEEE DAC, 2015, pp. 1–6.
10. Ida, I.B.; Jemai, A.; Loukil, A. A survey on security of IoT in the context of eHealth and clouds. IEEE IDT, 2016, pp. 25–30.
11. Weber, R.H. Internet of Things–New security and privacy challenges. *Comp. law & sec. rev.* **2010**, *26*, 23–30.
12. Fink, G.A.; Zarzhitsky, D.V.; Carroll, T.E.; Farquhar, E.D. Security and privacy grand challenges for the Internet of Things. IEEE CTS, 2015, pp. 27–34.
13. Kraijak, S.; Tuwanut, P. A survey on IoT architectures, protocols, applications, security, privacy, real-world implementation and future trends. IET WiCOM 2015, 2015, pp. 1–6.
14. Gubbi, J.; Buyya, R.; Marusic, S.; Palaniswami, M. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future gen. comp. sys.* **2013**, *29*, 1645–1660.
15. Chiang, M.; Zhang, T. Fog and IoT: An overview of research opportunities. *IEEE IoT Journal* **2016**, *3*, 854–864.
16. Liu, J.W.S. *Real-Time Systems*; Prentice Hall, 2000.
17. Ge, Y.; Liang, X.; Zhou, Y.C.; Pan, Z.; Zhao, G.T.; Zheng, Y.L. Adaptive analytic service for real-time Internet of things applications. IEEE ICWS, 2016, pp. 484–491.
18. Kim, J.E.; Abdelzaher, T.; Sha, L.; Bar-Noy, A.; Hobbs, R.; Dron, W. On maximizing quality of information for the Internet of things: A real-time scheduling perspective. IEEE RTCSA, 2016, pp. 202–211.
19. Buttazzo, G. *Hard real-time computing systems: predictable scheduling algorithms and applications*; Vol. 24, 2011.
20. Mok, A.K. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Massachusetts Institute of Technology, 1983.
21. Liu, C.L.; Layland, J.W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *JACM* **1973**, *20*, 46–61.
22. Davis, R.I. A review of fixed priority and EDF scheduling for hard real-time uniprocessor systems. *ACM SIGBED Review* **2014**, *11*, 8–19.
23. Davis, R.I.; Burns, A. A survey of hard real-time scheduling for multiprocessor systems. *ACM CSUR* **2011**, *43*, 35.
24. Joseph, M.; Pandya, P. Finding response times in a real-time system. *The Comp. J.* **1986**, *29*, 390–395.
25. Audsley, N.; Burns, A.; Richardson, M.; Tindell, K.; Wellings, A.J. Applying new scheduling theory to static priority pre-emptive scheduling. *SE Journal* **1993**, *8*, 284–292.
26. Bini, E.; Buttazzo, G.C. Schedulability analysis of periodic fixed priority systems. *IEEE Trans. on Comp.* **2004**, *53*, 1462–1473.
27. Mohan, S.; Bak, S.; Betti, E.; Yun, H.; Sha, L.; Caccamo, M. S3A: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems. ACM HiCoNS, 2013, pp. 65–74.
28. Yoon, M.K.; Mohan, S.; Choi, J.; Kim, J.E.; Sha, L. SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems. IEEE RTAS, 2013, pp. 21–32.
29. Yoon, M.K.; Mohan, S.; Choi, J.; Sha, L. Memory heat map: anomaly detection in real-time embedded systems using memory behavior. ACM/EDAC/IEEE DAC, 2015, pp. 1–6.
30. Yoon, M.K.; Mohan, S.; Choi, J.; Christodorescu, M.; Sha, L. Learning Execution Contexts from System Call Distribution for Anomaly Detection in Smart Embedded System. ACM/IEEE IoTDI, 2017, pp. 191–196.

31. Abad, F.A.T.; Van Der Woude, J.; Lu, Y.; Bak, S.; Caccamo, M.; Sha, L.; Mancuso, R.; Mohan, S. On-chip control flow integrity check for real time embedded systems. IEEE CPSNA, 2013, pp. 26–31.
32. Abdi, F.; Hasan, M.; Mohan, S.; Agarwal, D.; Caccamo, M. ReSecure: A Restart-Based Security Protocol for Tightly Actuated Hard Real-Time Systems. IEEE CERTS, 2016, pp. 47–54.
33. Pellizzoni, R.; Paryab, N.; Yoon, M.K.; Bak, S.; Mohan, S.; Bobba, R.B. A generalized model for preventing information leakage in hard real-time systems. IEEE RTAS, 2015, pp. 271–282.
34. Francillon, A.; Castelluccia, C. Code Injection Attacks on Harvard-architecture Devices. ACM CCS; ACM: New York, NY, USA, 2008; pp. 15–26.
35. Hasan, M.; Mohan, S.; Bobba, R.B.; Pellizzoni, R. Exploring Opportunistic Execution for Integrating Security into Legacy Hard Real-Time Systems. IEEE RTSS, 2016, pp. 123–134.
36. Hasan, M.; Mohan, S.; Pellizzoni, R.; Bobba, R.B. Contego: An Adaptive Framework for Integrating Security Tasks in Real-Time Systems. IEEE ECRTS, 2017.
37. Hasan, M.; Mohan, S.; Pellizzoni, R.; Bobba, R.B. A design-space exploration for allocating security tasks in multicore real-time systems. IEEE DATE, 2018, pp. 225–230.
38. Zhou, Y.; Feng, D. Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing. *IACR Cryptology ePrint Archive* **2005**, *2005*, 388.
39. Kelsey, J.; Schneier, B.; Wagner, D.; Hall, C. Side channel cryptanalysis of product ciphers. ESORICS, 1998, pp. 97–110.
40. Jiang, K.; Batina, L.; Eles, P.; Peng, Z. Robustness analysis of real-time scheduling against differential power analysis attacks. IEEE ISVLSI, 2014, pp. 450–455.
41. Son, J.; Alves-Foss, J. Covert timing channel analysis of rate monotonic real-time scheduling algorithm in MLS systems. IEEE IAW, 2006, pp. 361–368.
42. Agrawal, D.; Archambeault, B.; Rao, J.R.; Rohatgi, P. The EM side—channel (s). International Workshop on Cryptographic Hardware and Embedded Systems. Springer, 2002, pp. 29–45.
43. Bar-El, H.; Choukri, H.; Naccache, D.; Tunstall, M.; Whelan, C. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE* **2006**, *94*, 370–382.
44. Mohan, S.; Yoon, M.K.; Pellizzoni, R.; Bobba, R.B. Real-time systems security through scheduler constraints. IEEE ECRTS, 2014, pp. 129–140.
45. Loukas, G. *Cyber-physical attacks: A growing invisible threat*; Butterworth-Heinemann, 2015.
46. Xie, T.; Qin, X. Improving security for periodic tasks in embedded systems through scheduling. *ACM TECS* **2007**, *6*, 20.
47. Lin, M.; Xu, L.; Yang, L.T.; Qin, X.; Zheng, N.; Wu, Z.; Qiu, M. Static security optimization for real-time systems. *IEEE Trans. on Indust. Info.* **2009**, *5*, 22–37.
48. Chen, C.Y.; Ghassami, A.; Nagy, S.; Yoon, M.K.; Mohan, S.; Kiyavash, N.; Bobba, R.B.; Pellizzoni, R. Schedule-based side-channel attack in fixed-priority real-time systems. Technical report, 2015.
49. Abdi, F.; Chen, C.Y.; Hasan, M.; Liu, S.; Mohan, S.; Caccamo, M. Guaranteed physical security with restart-based design for cyber-physical systems. ACM/IEEE ICCPS, 2018, pp. 10–21.
50. Yoon, M.K.; Mohan, S.; Chen, C.Y.; Sha, L. TaskShuffler: A Schedule Randomization Protocol for Obfuscation against Timing Inference Attacks in Real-Time Systems. IEEE RTAS, 2016, pp. 1–12.
51. Sha, L. Using simplicity to control complexity. *IEEE Software* **2001**, *18*, 20–28.
52. Candea, G.; Kiciman, E.; Zhang, S.; Keyani, P.; Fox, A. JAGR: An autonomous self-recovering application server. IEE Auto. Comp. Wkshp., 2003, pp. 168–177.
53. Candea, G.; Fox, A. Recursive restartability: Turning the reboot sledgehammer into a scalpel. IEEE HOTOS, 2001, pp. 125–130.
54. Sha, L.; Rajkumar, R.; Lehoczky, J.P. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. on comp.* **1990**, *39*, 1175–1185.
55. Chen, C.Y.; Bobba, R.B.; Mohan, S. Schedule-based side-channel attack in fixed-priority real-time systems. Technical report, University of Illinois, http://hdl.handle.net/2142/88344, 2015. [Online].
56. Jiang, K.; Batina, L.; Eles, P.; Peng, Z. Robustness analysis of real-time scheduling against differential power analysis attacks. IEEE Computer Society Annual Symposium on VLSI. IEEE, 2014, pp. 450–455.
57. Lo, D.; Ismail, M.; Chen, T.; Suh, G.E. Slack-aware opportunistic monitoring for real-time systems. IEEE RTAS, 2014, pp. 203–214.
58. Virtualization and the Internet of things. 2016.

59. Ge, Q.; Yarom, Y.; Cock, D.; Heiser, G. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Crypt. Eng.* **2016**, pp. 1–27.
60. Fuchs, C.M.; others. The evolution of avionics networks from ARINC 429 to AFDX. *NW Arch. & Serv.* **2012**.
61. Farsi, M.; Ratcliff, K.; Barbosa, M. An overview of controller area network. *Comp. & Cont. Eng. Journal* **1999**, *10*, 113–120.
62. Hoppe, T.; Kiltz, S.; Dittmann, J. Security threats to automotive CAN networks–practical examples and selected short-term countermeasures. SAFECOMP, 2008, pp. 235–248.
63. McKeown, N.; Anderson, T.; Balakrishnan, H.; Parulkar, G.; Peterson, L.; Rexford, J.; Shenker, S.; Turner, J. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Comp. Comm. Rev.* **2008**, *38*, 69–74.
64. Kumar, R.; Hasan, M.; Padhy, S.; Evchenko, K.; Piramanayagam, L.; Mohan, S.; Bobba, R.B. End-to-End Network Delay Guarantees for Real-Time Systems using SDN. IEEE RTSS, 2017.
65. Son, S.; Chaney, C.; Thomlinson, N. Partial security policies to support timeliness in secure real-time databases. Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on, 1998, pp. 136–147.
66. Kadloor, S.; Kiyavash, N.; Venkitasubramaniam, P. Mitigating Timing Side Channel in Shared Schedulers. *CoRR* **2013**, *abs/1302.6123*.
67. Gong, X.; Kiyavash, N. Timing Side Channels in Shared Queues. *CoRR* **2014**, *abs/1403.1276*.
68. Osvik, D.A.; Shamir, A.; Tromer, E. Cache attacks and countermeasures: the case of AES. Crypt.' Track at the RSA Conf., 2006, pp. 1–20.
69. Page, D. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Crypt. ePrint Arc.* **2002**, *2002*, 169.
70. Völp, M.; Hamann, C.J.; Härtig, H. Avoiding timing channels in fixed-priority schedulers. ACM ASIACCS, 2008, pp. 44–55.
71. Bao, C.; Srivastava, A. A secure algorithm for task scheduling against side-channel attacks. International Workshop on Trustworthy Embedded Devices. ACM, 2014, pp. 3–12.
72. Kocher, P.; Jaffe, J.; Jun, B.; Rohatgi, P. Introduction to differential power analysis. *Journal of Cryptographic Engineering* **2011**, *1*, 5–27.
73. Ghassami, A.; Gong, X.; Kiyavash, N. Capacity limit of queueing timing channel in shared FCFS schedulers. IEEE ISIT, pp. 789–793.
74. Völp, M.; Engel, B.; Hamann, C.J.; Härtig, H. On Confidentiality Preserving Real-Time Locking Protocols. IEEE RTAS, 2013, pp. 153–162.
75. Krüger, K.; Völp, M.; Fohler, G. Vulnerability analysis and mitigation of directed timing inference based attacks on time-triggered systems. ECRTS, 2018, Vol. 106, pp. 22:1–22:17.
76. Lesi, V.; Jovanov, I.; Pajic, M. Network Scheduling for Secure Cyber-Physical Systems. IEEE RTSS, 2017, pp. 45–55.
77. Lesi, V.; Jovanov, I.; Pajic, M. Security-Aware Scheduling of Embedded Control Tasks. *ACM TECS* **2017**, *16*, 188:1–188:21.
78. Ngu, A.H.; Gutierrez, M.; Metsis, V.; Nepal, S.; Sheng, M.Z. IoT Middleware: A Survey on Issues and Enabling technologies. *IEEE IoT Journal* **2017**, *4*, 1–20.
79. Mohsin, M.; Anwar, Z.; Husari, G.; Al-Shaer, E.; Rahman, M.A. IoTSAT: A formal framework for security analysis of the Internet of things (IoT). IEEE CNS, 2016, pp. 180–188.
80. Wurm, J.; Hoang, K.; Arias, O.; Sadeghi, A.R.; Jin, Y. Security analysis on consumer and industrial IoT devices. IEEE ASP-DAC, 2016, pp. 519–524.
81. Pacheco, J.; Hariri, S. IoT Security Framework for Smart Cyber Infrastructures. IEEE FAS*, 2016, pp. 242–247.
82. smar, J.; Savola, R.; Savolainen, P.; Evesti, A. Mitigating IoT security threats with a trusted Network element. IEEE ICITST, 2016, pp. 260–265.