JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Sock: A Clock-Based Modeling Language to Specify Secure Periodic Real-Time Tasks

Stéphanie Challita* **and Benoit Combemale***
*IRISA, University of Rennes, France

ABSTRACT Nowadays, global connectivity generates tremendous amounts of data and increases the need for *Real-Time Systems* (RTS). These systems consist of hardware and software components that execute tasks under strict timing constraints and are used in a wide variety of applications, such as connected vehicles, smart homes, e-health, and Industry 4.0. While RTS offer enormous opportunities to improve our lives, they also introduce significant security risks. However, the literature has largely neglected time-aware security in periodic task-based RTS, even though time is a crucial factor that directly affects system security. To address this gap, we propose Sock, a clock-based modeling language and toolchain that enables developers to design, execute, and reason about periodic task-based RTS. The operational semantics of Sock are specified using finite state machines equipped with logical clocks for tasks and resource operations. Sock allows the analysis of schedulability and the observation of security-related properties. It integrates confidentiality-preserving mechanisms and supports the simulation of temporal attacks and countermeasures. We evaluate Sock through a set of synthetic task sets and a realistic automotive-inspired case study on which we executed temporal attacks, and show that Sock preserves task schedulability and confidentiality, and reveals abnormal system behavior.

KEYWORDS Real-Time Systems, Periodic Tasks, Domain-Specific Modeling Language, EMF, Logical Time

1. Introduction

Global connectivity generates massive amounts of data that, once collected and processed, can significantly improve human life. *Real-Time Systems* (RTS) have become crucial in this context for domains such as connected vehicles, smart homes, e-health, and Industry 4.0. RTS are information-processing systems, consisting of hardware and software components, that must execute tasks within specific time constraints. A simple example is a connected heater that periodically measures the room temperature and adjusts its power according to the latest measurement and a target temperature; when working correctly, such a heater can substantially reduce energy consumption.

RTS are capable of performing critical tasks with high accuracy and reliability, but this potential comes with significant security risks. A compromised connected heater, for instance, may waste energy instead of saving it, while attacks on safety-critical systems (e.g., aircraft engine control software) can cause failures leading to injuries or loss of life. As a result, RTS must be designed and implemented with particular care, taking both functional and non-functional properties into account.

The research community has proposed numerous mechanisms to improve the security of RTS (Zhao & Ge 2013; Xu et al. 2014; Wolf & Serpanos 2018; Hasan et al. 2024). However, many of these approaches overlook timing, even though temporal behavior is a defining characteristic of RTS (Hasan et al. 2024). At the same time, the literature reports many attacks that exploit the relation between time and RTS (Kocher 1996; Völp et al. 2008; Chen et al. 2015; Fournaris et al. 2017; Mahfouzi et al. 2019). This motivates a stronger focus on time-aware security for developers of such systems, particularly with respect to scheduler-level behavior and periodic task execution (Hasan et al. 2024). In RTS, time enters the picture through periodic tasks and event-driven tasks: periodic tasks execute repeatedly with a fixed period, whereas event-driven tasks react to specific

JOT reference format:
Stéphanie Challita and Benoit Combemale. *Sock: A Clock-Based Modeling Language to Specify Secure Periodic Real-Time Tasks.* Journal of Object Technology. Vol. 25, No. 3, 2026. Licensed under Attribution 4.0 International (CC BY 4.0) http://dx.doi.org/10.5381/jot.2026.25.3.a5

events in the environment. In this article, the focus is on periodic task-based systems, and the goal is to reason about the time-related security aspects of these systems. This work targets the early design and analysis phases of systems that exhibit soft real-time characteristics, where timing behavior primarily impacts performance, predictability, and potential security leakage rather than constituting a strict functional correctness condition. Sock is therefore intended to support design-space exploration and the identification of timing-related vulnerabilities, rather than to provide certification-oriented guarantees as required in firm or hard real-time contexts. Addressing such settings would require additional assumptions, such as sound worst-case execution time models and fully deterministic scheduling analyses, which are beyond the present scope. This deliberate minimalism is a design choice that isolates timing effects, allowing Sock to expose and analyze temporal security behaviors without interference from unrelated functional complexity.

To address this need, we propose **Sock**, a clock-based *Domain-Specific Modeling Language* (DSML) and supporting toolchain that helps developers design, verify, edit, and execute periodic task-based RTS. Sock is grounded in *Model-Driven Engineering* (MDE), which provides suitable abstraction mechanisms for specifying periodic task-based RTS, defining their requirements, and reasoning about them (Ciccozzi et al. 2017). At its core, Sock features a task scheduling mechanism that regulates the execution of periodic tasks while ensuring that timing constraints are respected throughout the system. In Sock, time is a central concern: each concept is associated with a set of logical clocks (Lamport 1978). Each logical clock is attached to a specific operation and ticks, *i.e.* becomes enabled, whenever that operation occurs during model execution. The operational semantics of Sock are defined by temporal constraints over these logical clocks. Building on this formalization, Sock supports the verification of confidentiality-preserving mechanisms and enables execution-level observations to detect abnormal system behaviors induced by temporal attacks, thereby allowing developers to assess whether mitigation mechanisms are required.

The contributions of this article are summarized as follows:

- *Sock*, a model-based approach for the design and analysis of periodic task-based systems with temporal security concerns,
- a schedulability-aware task scheduler to orchestrate task execution and verify deadline satisfaction,
- the integration of confidentiality-preserving mechanisms and execution-level observations to detect abnormal system behaviors induced by temporal attacks,
- an experimental evaluation demonstrating the applicability and foundations of Sock,
- an open-source implementation of Sock and its experimental data, ensuring reproducibility[1].

The remainder of this article is organized as follows. In Section 2, a motivating example and the background of the work are presented. Section 3 introduces Sock, the proposed clock-based DSML for secure periodic task-based systems, and details its concepts and semantics. Section 4 describes the properties that Sock enforces. Section 5 presents the implementation of Sock and the results of the evaluation. Section 6 discusses the limitations of Sock and of the evaluation. Section 7 reviews related work. Finally, Section 8 concludes the article and outlines future directions.

[1] https://github.com/stephaniechallita/Sock

2. Motivating Example & Background

To ground our work in a concrete setting, this section first presents a simple connected heater system that we use as a running example throughout the paper. We then describe the attacker model applied to our example, focusing on adversaries who manipulate the timing behavior of periodic real-time tasks.

2.1. Real-world Context: A Simple Connected Heater System

Modern connected devices often integrate several software components that must periodically execute tasks on limited hardware resources. To illustrate this context, we consider a simplified connected heater whose objective is to maintain a stable room temperature with minimal energy consumption. The system includes three software components:

- A Thermometer, responsible for periodically measuring the current room temperature and storing it in a local database.
- A Heating Device, responsible for reading the latest temperature and increasing or decreasing the heating power accordingly.
- A Monitor, responsible for periodically generating a visual display of the temperature evolution for the end user.

Although these components perform different functions, they are not required to run concurrently. In RTS, they often share a single processing unit (e.g., a low-power CPU). This makes the system lightweight and cost-effective, but it also requires careful coordination of the tasks' execution.

To describe the temporal behavior of these components, we rely on the model of periodic real-time tasks as defined by Liu and Layland (Liu & Layland 1973). In this model, an application is a set of periodic tasks $\tau = T_1, T_2, ..., T_n$, where each task T_i is characterized by a tuple: P_i, C_i, R_i where P_i is the task period, C_i is the time to complete the task and R_i is the resource assigned to the task.

This model makes the following assumptions:

- Each task executes exactly once per period, and its period is constant.
- Tasks are independent: there is no precedence relation, and no task depends on the completion of another.
- The time to complete the task C_i is constant and known in advance.

Representing these temporal aspects directly in physical time is challenging, especially in distributed or resource-constrained systems. For this reason, we adopt a representation based on logical time (Fidge 1991), a standard abstraction for reasoning about causal and chronological relationships without relying on real clocks. This heater example will serve as a running

scenario throughout the paper. It highlights the need for a modeling approach capable of capturing periodic behaviors, resource sharing, and timing constraints, all of which are central to our definition of Sock's syntax and semantics in the subsequent sections.

2.2. Threat Model: Time-Aware Attacker

Real-time systems are vulnerable not only to functional failures but also to time-aware attacks, where an adversary manipulates temporal parameters rather than data or control flow. In the context of the connected heater presented earlier, we consider the following threat scenario.

An attacker successfully performs a *Man-in-The-Middle* (MiTM) attack. Through this intrusion, the attacker eventually obtains elevated privileges on the device's single processing unit. With such access, the attacker can arbitrarily modify the period of any periodic task in the system.

A task whose period has been altered behaves incorrectly, because it no longer executes at the rate expected by the other components or by the underlying schedulability assumptions. We refer to any such task as *compromised*. In our heater example, each of the three components can be compromised independently, with distinct consequences:

- Compromised Thermometer: if the thermometer's period is increased, temperature measurements become less frequent. The heating controller operates on outdated information and may overheat or underheat the room.
- Compromised Heating Device: if the heating device executes too infrequently, it adjusts the heating power too late, reducing comfort and increasing energy consumption.
- Compromised Monitor: if the monitoring task is delayed, the user receives stale information and may manually configure the heating system incorrectly, again increasing consumption.

Although this particular example is not security-critical, similar timing manipulations in other real-time systems, such as automotive, industrial, or medical applications, could have severe consequences. This motivates the need for modeling techniques that can capture timing behavior explicitly, reason about its deviations, and support the analysis of resulting threats.

3. Sock approach

Sock is a model-based approach for the design and analysis of periodic task-based RTS with security considerations. This section presents the main foundations of Sock. We first introduce the abstract syntax in Section 3.1, which defines the core modeling concepts and their relationships. Then, in Section 3.2, we describe the concrete syntax specifying how Sock models are expressed textually. Next, Section 3.3 defines the semantic domain, characterizing the mathematical structures used to represent system executions, followed by the operational semantics in Section 3.4, which describe how models evolve over time. Then, we present the semantic mapping in Section 3.5 that links syntactic models to their formal semantics.

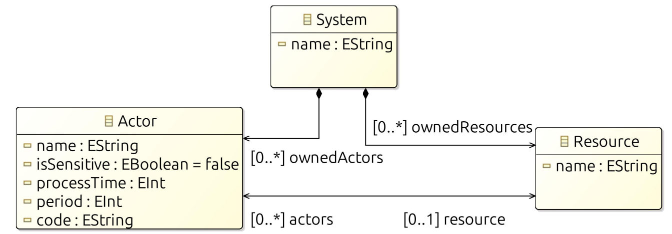

Figure 1 Sock metamodel.

3.1. Abstract syntax

As depicted in Figure 1, the Sock metamodel defines three core concepts: **System**, **Actor**, and **Resource**.

System is the root of the Sock metamodel. It has an attribute *name* (string), used as an identifier. It aggregates lists of **Actors** and **Resources**.

Actor represents a software component that executes periodic tasks using assigned resources. Each **Actor** is linked to at most one **Resource** to avoid modeling concurrency and the resulting combinatorial explosion in analysis.

The **Actor** concept defines five attributes:

- *name*: the name of the actor, used as a unique identifier.
- *isSensitive* (boolean): a flag that, if true, enforces resource cleanup after task completion to protect sensitive data (more details in Section 4.2).
- *period* (P_i, integer): the fixed time interval separating consecutive executions of the actor.
- *processTime* (C_i, integer): the execution time to complete the task.
- *code* (string): external code used during simulation to parameterize task behavior (e.g., workload or energy consumption). It is treated as an opaque function and is not part of Sock's timing semantics.

Resource is an entity that is used and required by the actors to complete their tasks. It has an attribute *name* (string), used as identifier. In Sock, **Resources** can be assimilated to hardware.

Connected Heater Example To illustrate the abstract syntax presented above, we provide a complete Sock model of the connected heater scenario described in Section 2.1. Table 1 summarizes the **Actors** composing the **System**.

Table 1 Characteristics of the actors in the connected heater model

Name	period	processTime C_i	isSensitive
Thermometer	25	5	✓
HeatingDevice	35	8	✗
Monitor	15	2	✗

Regarding the **code** attribute, Sock **Actors** should rely on a library that implements the behavior. For our example, one can imagine that the manufacturer has developed their software to manage the connected heater **Actors**. Therefore, the embedded

Actor code should rely on this existing code by wrapping it. As Sock is designed to be used at design time to ensure certain properties, such as schedulability, the developer should add to the code: 1) A way to simulate the environment; 2) A way to trace the simulation.

Below, we pursue the connected heater example by giving an example of code for each **Actor** of the System:

The Thermometer

```
int temperature = IOUtils.read(time,
    temperatures);
int power = IOUtils.read(time, powers);
int next = Computation.compute(time,
    temperature, power);
IOUtils.write(time, next, temperatures);
```

The code here measures the temperature over time by reading the file *temperatures*, which contains recorded values (line 1), and the current power of the **HeatingDevice** from the file *powers* (line 2). Both files are time-indexed, so *IOUtils.read* uses the current *time* to fetch the latest values. A mathematical model is then applied (line 3), producing a computational temperature evolution that is written back to *temperatures* (line 4).

The HeatingDevice

```
int power = IOUtils.read(time, powers);
int temperature = IOUtils.read(time,
    temperatures);
int next = temperature > TARGET
    ? power + 1
    : power - 1;
IOUtils.write(time, next, powers);
```

The code manages the power of the heating device according to a targeted temperature and the current measured temperature. Similarly to the **Thermometer**, it reads the current power (line 1) from *powers* and the temperature (line 2) from *temperatures*. It then decides whether to increase or decrease its power based on the difference between the current and target temperatures (line 3). Finally, it writes the updated power value to *powers* (line 4).

The Monitor

```
int temperature = IOUtils.read(time,
    temperatures);
Plot.streamPlot(temperature);
```

The code here traces the evolution of the temperature in a streaming plot. It reads the temperature (line 1) from *temperatures*, and plots it using a third-party library (line 2).

3.2. Concrete syntax

This section presents the concrete syntax of Sock, which defines a lightweight textual notation for specifying systems composed of periodic actors and shared resources. Implemented in Xtext and inspired by domain-specific configuration languages, this syntax provides a clear and structured way to write Sock models. The language is intentionally minimal and domain-specific: it provides constructs to declare a single **System**, a set of **Actors** with timing and sensitivity attributes, and shared **Resources** that structure contention. The complete grammar is available in the project repository[2].

Example To illustrate our concrete syntax, we describe below the connected heater scenario presented in Section 2.1.

```
System ConnectedHeater {
  ownedActor {
    Actor Thermometer {
      isSensitive true
      processTime 5
      period 25
      resource "CPU"
    },
    Actor HeatingDevice {
      isSensitive false
      processTime 8
      period 35
      resource "CPU"
    },
    Actor Monitor {
      isSensitive false
      processTime 2
      period 15
      resource "CPU"
    }
  }
  ownedResource {
    Resource CPU {
      actor (Thermometer, HeatingDevice,
    Monitor)
    }
  }
}
```

Listing 1 The connected heater example written with Sock grammar.

3.3. Semantic Domain

To assign accurate meaning to Sock models, we first define an abstract semantic domain that characterizes the class of computational behaviors that Sock intends to describe. This domain is independent of Sock's concrete or abstract syntax and provides the mathematical foundation upon which the language's semantics is constructed. Following Harel and Rumpe's recommendation (Harel & Rumpe 2004), the semantic domain formalizes the kinds of systems that Sock models denote before specifying how a model is mapped to such systems. The semantic domain is intended for design-time exploration of timing behavior and its security implications in soft real-time settings, rather than for certification-oriented analysis requiring worst-case execution-time soundness or hard deadline guarantees.

Sock targets periodic-task-based systems under resource constraints. Accordingly, the semantic domain is based on timed transition systems (TTS). A timed transition system is a tuple $(S, s_0, \tau, \rightarrow)$, in which:

[2] github.com/stephaniechallita/Sock

- S is the set of the states s_t of the **System**. A state, s_t, represents a snapshot of the entire system at a given logical time t. It includes:
    - the local control state of each actor: ready, processing, and completed;
    - the local actor variables such as remaining processing time;
    - the occupancy and internal state of each resource: free, allocated and cleaning;
    - the value of the global logical clock.
- s_0 is the initial state at time $t = 0$ where all actors are ready, all resources are free.
- τ is the Time domain. Sock operates over discrete logical time. Thus, $\tau = \mathbb{N}$, and time may advance by any non-negative integer amount subject to timing constraints (*e.g.*, actors becoming ready at their defined periods).
- $\to$ represents the transition relations. They capture the evolution of the system through the evolution of each entity: actors (*e.g.*, idle, processing, preemption) and resources (*e.g.*, being allocated, being freed, cleaning the state).

A behavior in this semantic domain is a timed execution trace, that is, a sequence:

$$(s_0, t_0) \to (s_1, t_1) \to (s_2, t_2) \to ... \quad (1)$$

where the application of a transition relation ($\to$) to a system state s_n at time t_n, producing a successor state s_{n+1} at time t_{n+1} is defined as a *step*: $(s_n, t_n) \to (s_{n+1}, t_{n+1})$. Each trace represents one admissible execution of the system under some scheduling policy. At this level, the scheduling policy is intentionally left unspecified; the semantic domain accommodates any policy that respects the resource constraints and timing conditions of the model.

This semantic foundation provides the structure upon which Sock's semantic mapping is defined in the next subsections.

3.4. Operational semantics

This section defines how Sock models evolve through a collection of operations. These operations describe, in a stepwise manner, how actors are activated, how they acquire and release resources, how time progresses, and how contention is resolved. Each operation is formally mapped to transitions in the semantic domain, giving a precise execution semantics for all actors and resources. The operational semantics provides a concrete execution model that serves as the basis for the semantic mapping of Section 3.5 and for the schedulability and security analyses presented later in Section 4.

The **System** has one operation: *time()*, which advances time by one step.

Actors have seven operations:

- *ready()* is the operation that starts the actor period. This operation is called every *period*.
- *allocate()* is the operation to enter the resource.
- *process()* is the operation to complete the task using the resource. According to the field *processTime*, this operation must be called *processTime* times to complete the task of the actor.
- *free()* is the operation to exit a resource.
- *idle()* is the operation that makes the actor wait.
- *takeOver()* is the operation that preempts an actor from using the resource.
- *isTakenOver()* is the operation called when an actor is preempted from a resource.

Resources have seven operations:

- *isAllocated()* is the operation that is called when an **Actor** enters the **Resource**.
- *isProcessed()* is the operation that is called when an **Actor** processes in the **Resource**.
- *isFreed()* is the operation that is called when an **Actor** frees the **Resource**.
- *isFreedForClean()* is the operation that is called when a sensitive **Actor** frees the **Resource**.
- *actorsSwap()* is the operation that is called when an **Actor** takes over the **Actor** that allocated the **Resource**.
- *clean()* is the operation that flushes the data out of the resource in order to avoid any leak of sensitive information.
- *idle()* is the operation that makes the resource wait when there is no actor using the resource. For instance, when every actor has completed their task for the current period, the resource is no longer occupied by any actor, thus the resource goes idle. While in Sock this operation only makes the resource wait, traditionally systems implement this operation to lower the energy consumption of the resource (Wysocki n.d.).

FSMs Using the action primitives defined above, we specify the behavioral semantics of each entity via synchronous *Finite State Machine* (FSM)s, which give Sock its state-based operational meaning. The transitions are triggered by the operations and reflect the coordinated evolution of actors and resources along the shared logical time. This is detailed in Section 3.5.

The associated FSM to the **System** has one state, **Running**, and one recursive transition named *time*.

We detail now the **Actor** FSM depicted in Figure 2. The initial state is **Starting**. Initially, there is one transition: **Starting** – *start* $\to$ **Ready**. From the **Ready** state, there are three transitions:

- **Ready** – *idle* $\to$ **Ready**
- **Ready** – *allocate* $\to$ **Processing**
- **Ready** – *takeOver* $\to$ **Processing**

From the **Processing** state, there are three transitions:

- **Processing** – *isTakenOver* $\to$ **Ready**
- **Processing** – *process* $\to$ **Processing**
- **Processing** – *free* $\to$ **Finished**

From the **Finished** state, there are two transitions:

- **Finished** – *idle* $\to$ **Finished**

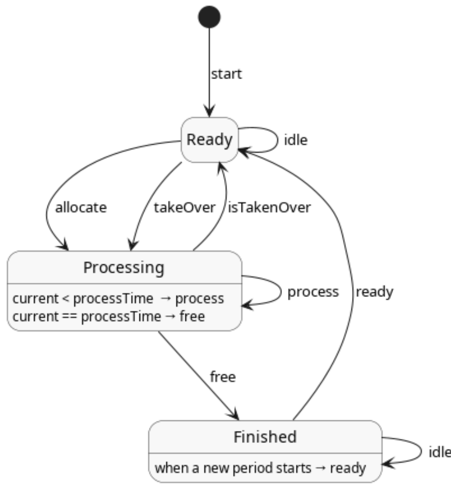

Figure 2 Actor FSM.

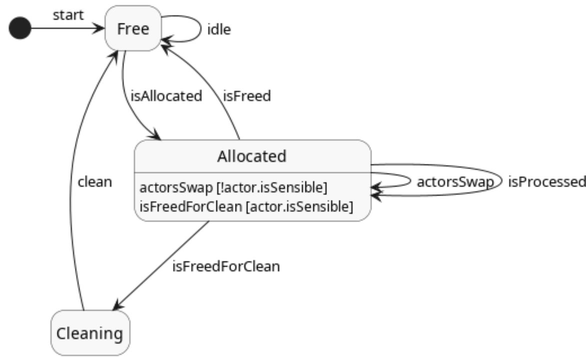

Figure 3 Resource FSM.

- **Finished** – *ready* → **Ready**

The *ready* transition relation is applied when the **Actor** starts a new period.

We detail now the **Resource** FSM depicted in Figure 3. The initial state is **Starting**. Initially, there is one transition: **Starting** – *start* → **Free**. From the **Free** state, there are three transitions:

- **Free** – *idle* → **Free**
- **Free** – *isAllocated* → **Allocated**

From the **Allocated** state, there are three transitions:

- **Allocated** – *isProcessed* → **Allocated**
- **Allocated** – *actorsSwap* → **Allocated**
- **Allocated** – *isFreed* → **Free**
- **Allocated** – *isFreedForClean* → **Cleaning**

3.5. Semantic mapping

We map here the semantic domain described in Section 3.3, to Sock's entities, namely the **System**, **Actors** and **Resources**. To do this mapping, Sock relies on FSM entities described in Section 3.4. The state s_t, at a given time, is the set of the current state of each FSM, representing the local control state for **Actors** and the occupancy for **Resources**. Sock links each FSM transition to a logical clock and to the entity operation. Eventually, this means that when a FSM transition is executed, the associated logical clock ticks, and the associated operation is executed. FSM transitions, the associated logical clocks, and the associated operations are mapped to the transition relations → (defined in Section 3.3) that evolve the state of the entire **System**. Sock temporally constrains the transitions relation by making associated logical clocks **coincide**, which is a synchronous temporal relation in which the clocks tick simultaneously (DeAntoni & Mallet 2012). Below, we say that two transition relations **coincide** when their logical clocks coincide. Note that when two transition relations **coincide**, only one of each transition relation must be applied, across all entities of the **System**. This means that the FSMs, operations and logical clocks are synchronized across the **System**.

It is the **System** that is responsible for making the time progress in a step-wise manner for the whole system, *i.e.*, for the **Actors** and the **Resource**. The **System** executes its *time* transition at each step. Every FSM is bound to the *time* transition, meaning that each time the *time* transition relation is applied, every FSM must apply a transition relation. In this context, the **idle** transition relation, present for both **Actors** and **Resources**, allows triggering a transition relation at each step, even if the entity has nothing to do. For instance, an **Actor** has **Finished** its task, hence waits for the next period to begin. In case no transition relation can be applied, Sock will crash the system and report the issue to the developer.

We now describe the constraints on the transition relations (→), broken down according to the temporal operations of Sock: *task processing*, *periodicity*, and *preemption*.

Regarding *task processing*, both **allocate** from **Actor** and **isAllocated** for **Resource** coincide, and mutual exclusion prevents multiple actors from allocating the same resource simultaneously. Then, **process** from **Actor** and **isProcessed** from **Resource** coincide and encode the fact that an **Actor** is processing using a **Resource**. Eventually, **free** from **Actor** coincides with either **isFreed** or **isFreedForClean** from **Resource**, depending on the **Actor** sensitivity. If the **Actor** is sensitive, the **isFreedForClean** must be applied; otherwise it is **isFreed**.

Then, for **Actors** *periodicity*, the **ready** transition relation is applied every count of the **Actor** period, *i.e.* the global time is a multiple of the **Actor** period. However, the **ready** transition relation can be applied only if the **Actor** is in the **Finished** state. If the **Actor** has not reached the **Finished** state when its **ready** transition relation must be applied, it means the **Actor** missed its deadline, and Sock crashes the whole system.

Eventually, *preemption* is operated by both **takeOver** and **isTakenOver** transition relations from two distinct **Actors** coincide. In addition, **actorsSwap** transition relation from the shared **Resource** coincides with these two **Actor** transition relations. Note that the **actorsSwap** transition relation can only be applied if the current **Actor** is not sensitive.

4. Property Definition

Building on the formal semantic domain and operational semantics defined in Section 3, we present here the three properties

that can be analyzed with Sock: task schedulability (Section 4.1) and two security-related properties (Section 4.2).

4.1. Task schedulability

Sock allows developers to define a periodic task-based system as a set of **Actors** that execute on shared **Resources**. To reason about timing feasibility, Sock adopts the classical periodic task model introduced by Liu and Layland (Liu & Layland 1973), as illustrated in Section 2.1. This model provides a well-established sufficient condition for schedulability under fixed-priority scheduling.

For a system composed of n **Actors**, where C_i denotes the computation time (process time) of actor i and P_i its period, the system is deemed schedulable if the following condition holds:

$$\forall\ resource\ r \mid \sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{\frac{1}{n}} - 1) \qquad (2)$$

This condition is used in Sock as a design-time feasibility criterion to establish a baseline configuration; it is not intended to provide a sound schedulability guarantee under all modeled security mechanisms. In this context, schedulability means that, under the assumptions of Liu and Layland (Liu & Layland 1973), each **Actor** can complete its execution within its period. This check serves as a conservative baseline that establishes the timing feasibility before considering additional security-related properties.

Beyond this analytical check, Sock relies on an executable semantics in which system behavior is described as a sequence of state transitions, see Equation 1. System evolution is driven by a *scheduler* responsible for selecting which transition relations are applied at each step, subject to the temporal and resource constraints defined in Section 3.5. For instance, we recall that the **allocate** transition of an **Actor** must coincide with the **isAllocated** transition of the corresponding **Resource**.

When multiple transitions are enabled at the same time, the Sock scheduler resolves conflicts based on actor priorities. These priorities are derived from task periods using a rate-monotonic policy: actors with shorter periods are given higher priority. Accordingly, the scheduler orders enabled transitions in decreasing priority order, while still respecting the semantic constraints imposed by shared resources and timing conditions. Algorithm 1 shows the algorithm that we implemented in the Sock scheduler to handle this responsibility. This algorithm takes as input a **System** s and sets of transition relations to apply. It also has access to the $select()$ function, which returns the transition relation that is related to the highest priority **Actor**. Algorithm 1 is explained as follows. First, it filters the transition relations sets to only keep the ones that have **allocate**in them, named $allocates$. This is done thanks to the method $getAllocates(trSets)$ (line 1), which returns sets that contain the **allocate**. If $allocates$ is not empty (line 2), the algorithm returns the set of transition relations that contains the transition relation **allocate** of the highest priority actor (line 3). If $allocates$ is empty, the algorithm filters transition relation sets to only keep the ones that have **takeOver** in them, named $takeovers$. This is done thanks to the method $getTakeovers(trSets)$ (line 4), which returns sets that contain the **takeOver**. If $takeovers$ is empty (line 5), it means that the system is in one of the following cases: 1) The resource is **Cleaning** its state; 2) All the actors are **Finished**, waiting to be *ready()* for their next period; 3) There is an actor in the resource, processing, *i.e.* the resource is **Allocated**, and all the other actors are **Finished**, waiting to be *ready()* for their next period (this case is a special instance of the second one). In all these three cases, there is only one transition relation set and the algorithm returns it (line 6). If there is at least one set that has the operation **takeOver**, the algorithm retrieves the one of the highest priority actor (line 7), named $selectedTakeOver$. Since there are possible **takeOver**, it means that there is an **Actor** inside the **Resource**, processing. Thus, there is a set that has the **process**. At this stage, the algorithm filters transition relation sets to only keep the ones that have **process** in them, named $processes$ (line 8). Then, the algorithm retrieves the one of the highest priority actor (line 9), named $selectedProcess$. It compares the actor priority of both selected sets, *i.e.* $selectedTakeOver$ and $selectedProcess$ (line 10), and returns the set that contains the **Actor** with the highest priority (line 11 and line 13).

Algorithm 1 Sock scheduler algorithm

Require: System := s
Require: Transition Relations Sets := $trSets$
Require: Function to select Transition Relations with highest priority := $select()$
Ensure: Next set of Transition Relations to apply
1: $allocates \leftarrow getAllocates(trSets)$
2: **if** $allocates \neq \emptyset$ **then**
3:     **return** $select(allocates)$
4: $takeovers \leftarrow getTakeovers(trSets)$
5: **if** $takeovers = \emptyset$ **then**
6:     **return** $trSets.get(0)$
7: $selectedTakeOver \leftarrow select(takeovers)$
8: $processes \leftarrow getProcess(trSets)$
9: $selectedProcess \leftarrow select(processes)$
10: **if** $selectedProcess > selectedTakeOver$ **then**
11:     **return** $selectedProcess$
12: **else**
13:     **return** $selectedTakeOver$

Schedulability of the connected heater example For a model with $n = 3$ actors, the resource bound is $3\left(2^{\frac{1}{3}} - 1\right) \approx 0.79$. The sum of the schedulability rates is $\frac{5}{25} + \frac{8}{35} + \frac{2}{15} = 0.20 + 0.23 + 0.13 = 0.60$. For this particular model, we have $0.60 < 0.79$, which means that this model is schedulable. The order of the actors' execution is Monitor, Thermometer, and HeatingDevice, with respect to their schedulability rates.

Figure 4 illustrates the execution of the connected heater example over 60 steps, which is sufficient to highlight the system's behavior. Each actor is represented by a separate line: blue for the Thermometer, green for the HeatingDevice, and red for the Monitor. A rising edge indicates that an actor allocates the resource, while the falling edge marks when the actor exits

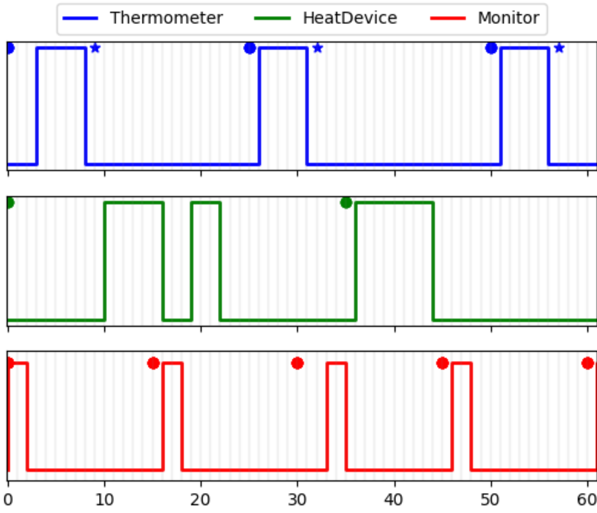

Figure 4 Binary plot representing the execution of the example.

and frees the resource. While the line remains high, the actor is actively processing. Dots indicate the beginning of an actor's period, and stars denote when the resource state is cleaned.

From the graph, we observe that the first period of each actor starts at $t = 0$. The resource usage cycle then begins with the execution of the Monitor, followed by the Thermometer. After the Thermometer completes its execution, a star appears, indicating that the resource cleans its state. This cleanup occurs because the Thermometer manipulates sensitive information, *i.e.* $S_{\text{Thermometer}} = 3$. Subsequently, the HeatingDevice enters the resource. However, it is preempted by the Monitor, which has a higher priority. Once the Monitor completes its task, the HeatingDevice resumes execution to finish its own task.

4.2. Security properties

To illustrate how Sock can be used to model and analyze timing-related threats, we instantiate two representative security properties drawn from the literature. These properties serve as demonstrative scenarios rather than an exhaustive treatment of real-time system security.

Confidentiality of sensitive data Sock enforces the confidentiality of sensitive data through a resource-level cleansing mechanism. When an **Actor** manipulates sensitive information, *i.e.*, when $isSensitive = true$, the developer specifies that this information must not be observable by any other actor. To enforce this constraint, Sock associates a *clean()* operation to the corresponding **Resource**, as introduced in Section 3.4, which is executed when the actor releases the resource.

We emphasize that the sensitivity of the data manipulated by an actor is independent of its priority. Sensitivity is a data-related concern, whereas priority is derived from timing parameters and is used for schedulability purposes (see Section 4.1). As a result, a sensitive **Actor** may have lower priority than an insensitive one. However, the temporal constraints encoded in the transition relations prevent a sensitive actor from being preempted while holding a resource, ensuring that sensitive data cannot be exposed during execution.

The execution of the *clean()* operation introduces a temporary unavailability of the resource. During this cleaning phase, the resource cannot be allocated by any actor, which effectively induces a blocking interval for other tasks. This blocking represents an additional execution cost that is not captured by the classical Liu and Layland utilization bound. Rather than modifying the theoretical schedulability condition, Sock accounts for this effect by extending the computational cost of sensitive actors.

Consider the process time C_i of an actor. Its computational cost C_i^s equals C_i for non-sensitive actors, and $C_i + 1$ for sensitive actors, where the additional unit represents a fixed design-time increment that accounts for the temporary blocking induced by the cleaning operation. This approach is inspired by the work of Mohan *et al.* (Mohan et al. 2014), who argue that protecting sensitive data through cleansing mechanisms necessarily incurs execution overhead. In line with this observation, Sock adopts a minimal and conservative abstraction in which the cost of cleaning is fixed to one step. While this abstraction does not provide a tight schedulability guarantee, it enables developers to reason explicitly about the security–performance trade-off at design time. Allowing developers to specify actor- or resource-specific cleaning costs is left for future work.

In addition to resource cleaning, Sock supports confidentiality by randomizing the scheduling decisions in order to limit information leakage through timing observations. A deterministic scheduler may allow an attacker to predict which **Actor** will access a **Resource** at a given time, which increases the effectiveness of timing-based side-channel attacks, such as those described by Kocher (Kocher 1996). To mitigate this risk, Sock provides an optional randomized variant of its scheduler, inspired by schedule randomization protocols such as *TaskShuffler* (Yoon et al. 2016). The principle is to introduce controlled deviations from strict fixed-priority scheduling while preserving deadline satisfaction. Concretely, instead of always selecting the highest-priority ready **Actor**, the randomized scheduler may select a lower-priority **Actor** among the eligible ones.

This randomization is subject to a runtime admissibility check. Before committing to a randomized choice, the scheduler verifies that delaying the execution of the highest-priority **Actor** does not cause it to miss its deadline. This is achieved by computing, in a step-wise manner, the latest start time at which the highest-priority **Actor** can still complete before its deadline, assuming the execution of another **Actor** beforehand. If this condition is not satisfied, the scheduler falls back to the deterministic, highest-priority choice. This local verification ensures that, at each scheduling step, the system remains schedulable under the current execution. The randomized scheduler does not modify the task model nor the schedulability condition defined in Section 4.1, but rather refines the scheduling policy used to apply transition relations. As such, randomization is an optional execution-time mechanism that trades predictability for increased resistance to timing-based information leakage.

We acknowledge that the step-wise admissibility check used to preserve schedulability does not constitute a formal proof of deadline preservation under all possible random schedules.

A comprehensive theoretical analysis of schedulability under scheduler randomization is outside the scope of this paper and discussed further in Section 6. The purpose of this mechanism is to demonstrate how Sock can be used to prototype and evaluate security-oriented scheduling strategies, and to pave the way for future extensions, such as customizable or mixed-criticality–aware schedulers.

Abnormal Behavior Detection The second security property integrated into Sock is the **detection of abnormal behavior of a component**. This abnormal behavior can be detected when the energy cost of a component tends towards infinity (Lincoln & Cervin 2002). In Sock, the responsibility of the energy cost is given to the *code* attribute of the **Actor**. This *code* attribute, which is either plain Java code or Groovy script, should return an integer that indicates how much the task has consumed energy. With this, when an actor completes its task once, it also reports the energy consumed to do so. According to the task that might be influenced by the environment or others actors, the energy consumed by the actor varies. Sock monitors the energy consumption of each **Actor** and reports it to the developer. The developer can then analyze this energy consumption and decide if there is a problem with stability for each component. In case the developer detects a problem, they could adapt their model in order to avoid such a situation before the real deployment, preventing potential damages and extra cost. The objective is to support parametric exploration and scenario comparison, not cycle-accurate or hardware-faithful energy estimation.

Security of the connected heater example Regarding the manipulation of sensitive data, we recall that the Thermometer is marked as sensitive (see Table 1). When it releases the **CPU** (falling edge at $t = 31$ in Figure 4), the **Resource** performs a state cleaning operation (blue star at $t = 32$ in Figure 4), during which no other **Actor** may allocate the **Resource**. Although a new period of the Monitor begins earlier (red dot at $t = 30$ in Figure 4), it cannot allocate the **Resource** until the cleaning completes, and therefore only starts execution at $t = 33$.

Regarding energy consumption, we modify the *code* attribute described in Section 3.1 as follows. For the Thermometer and the Monitor, energy consumption is modeled as a constant, since their tasks (temperature sensing and logging) do not induce significant variation. In contrast, the energy consumption of the HeatingDevice depends on its current power level. As the heating power increases or decreases according to the difference between the measured and target temperatures (see the *next* variable in the code), the returned energy consumption directly reflects this power level. Therefore, the *code* attribute of the HeatingDevice would return the variable *next* to report its energy consumption.

5. Evaluation of the Sock approach

To evaluate Sock, we design an experimental protocol to answer the following research questions:

RQ1 Does Sock provide sufficient semantic coverage to represent and execute diverse periodic task-system configurations?

RQ2 How does Sock support the preservation of confidentiality against timing-based attacks in periodic task-based systems?

RQ3 Can Sock reveal abnormal system behavior induced by temporal attacks through execution-level observations?

The goal of this evaluation is to demonstrate the feasibility of Sock as a modeling and exploration framework and to exercise its constructs on varied scenarios, rather than to establish statistical representativeness over a class of real-time systems.

We first present the implementation of the Sock approach using GEMOC Studio (Bousse et al. 2016) (Section 5.1). Second, we explain the protocol that we applied to answer each research question (Section 5.2). Later on, we expose the results of the experimentation and our answers to the research questions (Section 5.3).

5.1. Implementation

Sock is implemented as a set of Eclipse plugins using the GEMOC Studio (Bousse et al. 2016). GEMOC Studio yields generic components to develop and use executable DSMLs. These DSMLs are based on *Eclipse Modeling Framework* (EMF). In addition to this, GEMOC Studio provides a framework to apply clock-based constraints to the concepts of the model. The semantic mapping of Sock (see Section 3.5) is implemented using *Clock Constraint Specification Language* (CCSL) (André 2009) and *Model of Concurrency and Communication Modeling Language* (MoCCML) (Deantoni et al. 2015). The former is a modeling language to specify relations between logical clocks, while the latter is used to formally specify the concurrency aspect of operational semantics. The scheduling algorithm (see Section 4.1) is implemented as a *Step Decider* within GEMOC Studio. At runtime, the Step Decider chooses which set of clocks must tick, hence the set of transition relations to apply. We provide Sock and the results of our experimentation in open access on GitHub[3].

5.2. Evaluation protocol

5.2.1. Protocol to answer RQ1 To evaluate whether Sock can systematically model and execute periodic task-based RTS within a broad and well-defined class, we adopt a generative evaluation approach. We define a parameter space characterizing periodic task systems, including: 1. the number of resources, 2. the number of actors assigned to each resource, 3. actor periods, and 4. actor process times. All actors assigned to the same resource share that resource exclusively, and resources are independent. From this parameter space, we randomly generate multiple independent system instances expressed directly in Sock's concrete syntax. Since executions of periodic task systems are potentially infinite, each model is executed for a duration equal to the hyper-period of the resource, defined as the least common multiple of the periods of all actors assigned to that resource. Executing one hyper-period is sufficient, as the behavior of periodic actors repeats identically in subsequent hyper-periods.

[3] https://github.com/stephaniechallita/Sock

By generating and executing multiple system instances covering diverse combinations of periods, workloads, and actor configurations, this protocol evaluates Sock's ability to represent and execute arbitrary instances within the considered class of periodic task-based RTS. The absence of execution failures or semantic inconsistencies across all generated models provides evidence that Sock supports systematic modeling and execution across the defined parameter space.

5.2.2. Protocol to answer RQ2

To evaluate Sock's ability to mitigate timing-based confidentiality leaks, we select a representative schedule inference attack from the literature that exploits temporal behavior to recover task schedules. We then randomly generate 10 Sock models, each composed of a single shared resource and multiple periodic actors with heterogeneous timing parameters and sensitivity attributes. For each model, we execute the system for one hyper-period using the default Sock scheduler and apply the selected attack. We repeat the experiment using the same models while enabling scheduler randomization in Sock. Finally, we compare the attack's effectiveness, with and without randomization, by comparing how much exploitable temporal information can be extracted from the system's execution traces.

5.2.3. Protocol to answer RQ3

We select a temporal attack from the state of the art whose objective is to alter the functional behavior of a system without violating its schedulability. The attack must exploit the temporal dynamics of periodic task-based RTS. We then model both the system and the attack in Sock and execute two configurations: (i) a reference execution without the attack, and (ii) an execution in which the attack is enabled. For both executions, we collect execution traces and observable system metrics. We compare these observations to determine whether the presence of the attack induces abnormal behavior that can be exposed at the system level. In particular, we focus on control-related indicators, such as the evolution of physical variables and the energy consumption of components.

5.3. Experimentation and results

5.3.1. Answer to RQ1

To answer *RQ1*, we evaluated Sock on a set of randomly generated periodic task-based systems expressed directly in Sock's concrete syntax. Each generated system is parameterized as follows. The number of resources R_i is chosen in the interval $[1;2]$. For each resource, the number of actors N is selected in the interval $[2;5]$. The period of each actor P_i is chosen in $[5;50]$ with a step of 5, and the corresponding process time C_i is selected in $[1;10]$. Each actor is marked as sensitive ($S_i = ✓$) with a probability of 33%; otherwise, it is marked as non-sensitive ($S_i = ✗$).

Using this parameter space, we generated 10 independent Sock models, summarized in Table 2. For each resource, the table reports its hyper-period, defined as the least common multiple of the periods of the actors assigned to that resource. Actor characteristics are denoted using the format (P_i, C_i, S_i), where P_i is the period, C_i the process time, and S_i the sensitivity flag. Each generated model was successfully designed and executed for one complete hyper-period per resource. Since the behavior of periodic actors repeats identically across hyper-periods, this execution window is representative of the system's long-term behavior.

These results demonstrate that Sock can systematically model and execute periodic task-based systems drawn from the considered parameter space. All generated models are available in our GitHub repository[4] to support reproducibility.

Table 2 Descriptions of the generated Sock models.

R	HP	Actors(P_i, C_i, S_i)
r0	150	(25, 5, ✗), (30, 8, ✗)
r1	1575	(25, 4, ✗), (45, 9, ✓), (35, 7, ✗)
r0	360	(45, 2, ✗), (20, 5, ✗), (40, 1, ✗), (45, 6, ✗)
r0	120	(40, 3, ✗), (15, 7, ✗)
r1	120	(40, 9, ✗), (15, 1, ✓), (40, 1, ✓), (40, 1, ✗)
r0	200	(25, 1, ✓), (40, 5, ✗), (25, 6, ✗)
r0	120	(40, 6, ✓), (30, 7, ✗), (40, 3, ✓)
r1	420	(30, 4, ✗), (35, 5, ✗), (20, 5, ✗)
r0	150	(25, 1, ✓), (25, 4, ✓), (30, 7, ✗)
r1	150	(25, 6, ✗), (30, 9, ✗)
r0	45	(45, 7, ✗), (15, 6, ✗)
r0	105	(35, 6, ✗), (15, 6, ✗)
r1	360	(40, 4, ✓), (45, 5, ✗), (40, 8, ✗), (45, 2, ✓)
r0	100	(25, 8, ✓), (20, 3, ✗)
r1	210	(30, 5, ✗), (35, 2, ✓), (35, 8, ✗)
r0	280	(40, 6, ✗), (20, 5, ✗), (35, 3, ✓)

5.3.2. Answer to RQ2

To evaluate whether scheduler randomization in Sock reduces the effectiveness of timing-based attacks, we consider the *ScheduLeak* attack proposed by Chen *et al.* (Chen et al. 2015). ScheduLeak is a side-channel attack that exploits the idle task of a shared resource to infer the execution schedule of co-located tasks. By observing idle intervals, the attacker reconstructs an estimation of each task's arrival time, *i.e.*, the time instants at which actors enter the resource. We selected ScheduLeak because it explicitly exploits temporal behavior, a core characteristic of periodic real-time systems, and aims at leaking schedule-related information, which may subsequently facilitate more severe attacks. The effectiveness of the attack is quantified by the percentage of actor arrival times that are correctly inferred over a system execution.

We implemented ScheduLeak within Sock and applied it to a set of randomly generated Sock models. Each model consists of a single shared resource and between 2 and 5 actors (N). Actor periods (P_i) range from 10 to 50 (step 10), processing times (C_i) from 1 to 5, and each actor is marked as sensitive (S_i) with a probability of 33%. For each model, the hyper-period (HP) is limited to 250. Table 3 summarizes the characteristics of the ten generated models and the corresponding attack results. For each model, we execute the system for one hyper-period using (i) the default Sock scheduler and (ii) the randomized scheduler.

[4] https://github.com/stephaniechallita/Sock

We then apply ScheduLeak to both executions and compare the percentage of correctly inferred arrival times.

Table 3 Results of *ScheduLeak* in the percentage of the estimated arrival times.

N	HP	Actors(P_i, C_i, S_i)	Orig. %	Rnd. %
4	150	(15, 1, ✗), (30, 2, ✗), (10, 1, ✗) (25, 1, ✗)	76.32	18.92
3	60	(30, 2, ✗), (15, 3, ✗), (20, 1, ✗)	100.00	18.18
3	100	(10, 1, ✗), (20, 1, ✓), (25, 4, ✗)	100.00	52.38
3	40	(40, 3, ✗), (10, 1, ✗), (20, 4, ✓)	100.00	50.00
4	105	(35, 3, ✗), (35, 4, ✗), (35, 3, ✓) (15, 1, ✗)	73.33	20.00
3	180	(10, 2, ✗), (20, 2, ✗), (45, 2, ✗)	72.73	12.12
3	150	(25, 4, ✗), (30, 3, ✗), (15, 1, ✗)	100.00	33.33
3	90	(30, 4, ✗), (45, 4, ✗), (10, 1, ✗)	100.00	73.33
4	120	(40, 3, ✗), (30, 1, ✓), (30, 4, ✗) (30, 3, ✓)	100.00	7.14
5	30	(30, 2, ✓), (30, 3, ✗), (30, 3, ✗) (30, 1, ✗), (30, 2, ✗)	100.00	20.00

The results show a clear reduction in attack effectiveness when scheduler randomization is enabled. While ScheduLeak reconstructs 100% of arrival times with the original scheduler, this percentage drops to a median value of 20% under scheduler randomization. **These results indicate that scheduler randomization in Sock significantly reduces the leakage of schedule-related information, thereby mitigating timing-based schedule inference attacks.**

In all evaluated cases, scheduler randomization did not lead to deadline misses. Although not a formal guarantee, this observation suggests that randomization preserves schedulability for the considered models.

5.3.3. Answer to RQ3

We selected the *Butterfly attack* by Mahfouzi *et al.* (Mahfouzi et al. 2019) for three reasons: it is recent and temporally focused, it is domain-independent and applicable to any periodic task-based system, and the authors provide sufficient implementation details to reproduce it in Sock. This attack manipulates a non-critical task to indirectly disrupt a critical task, remaining stealthy while inducing abnormal system behavior. Concretely, it drops ticks from the logical clock starting a new actor period, increasing that actor's period.

In Sock, we implemented the Butterfly attack by adding a transition from the **Ready** state to **Finished** in the actor's FSM (Figure 2). This simulates incomplete task execution while maintaining overall schedulability. As a use case, we relied on the first case study by Mahfouzi *et al.* (Mahfouzi et al. 2019), a modern automotive DC motor control system: the positioning task T_1 guides the DC motor task T_2. The tasks are defined as

$$T_1 : (C_1 = 10, P_1 = 15) \quad T_2 : (C_2 = 5, P_2 = 30)$$

When the Butterfly attack is enabled, it withdraws 3 ticks of 4, and thus the period of the positioning task is multiplied by 4, *i.e.*, $P_1 = 15 \times 4 = 60$. We report the evolution of the DC motor speed in Figure 5 for both executions, with and without the Butterfly attack enabled. The black step is the desired speed. The vehicle positioning task must indicate to the DC motor controller if it has to accelerate or slow down to reach the desired speed. The blue plot is the speed of the DC motor without the attack enabled, and the red plot is the motor speed when the attack is enabled. In this graph, we see that the speed of the motor under attack is unstable and cannot converge to the desired speed. This happens because the vehicle positioning task does not indicate sufficiently often the correct order, *i.e.* accelerate or slow down, to the DC motor. By increasing the period of the vehicle positioning task, the vehicle cannot reach the desired speed because the DC motor is still accelerating or slowing down while it should not. This is because the vehicle positioning task is not updated frequently enough, as its execution period is too long.

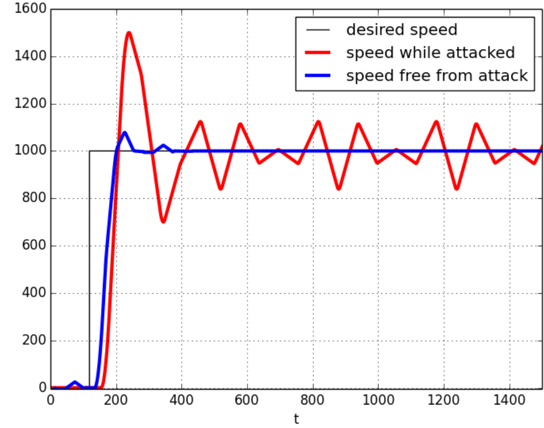

Figure 5 Evolution of the DC motor speed.

Figure 6 reports the energy consumption of the DC motor controller. The blue line is the energy consumption of the DC motor controller while the Butterfly attack is disabled, and the red line is the energy consumption of the DC motor controller while the Butterfly attack is enabled. The energy consumption tends towards infinity when the Butterfly attack is enabled. This confirms the instability seen in Figure 5. **This experiment shows how Sock reports the abnormal behavior of a DC motor by monitoring its energy consumption.**

The two attacks considered for RQ2 and RQ3, *Scheduleak* and the *Butterfly attack*, serve as representative, literature-documented scenarios illustrating how Sock can model and analyze timing-based threats at design time; they are not intended to provide an exhaustive taxonomy of temporal attacks.

6. Discussion

The previous sections presented Sock and its evaluation. While our results are promising, it is important to discuss the limitations of our approach and clarify the scope of its applicability.

Scope of the Approach Sock is intended for design-time analysis in soft real-time contexts, where timing affects performance and potential information leakage rather than constituting a strict correctness requirement. Supporting firm or hard real-time guarantees would require integrating sound worst-case execution-time models and conservative schedulability analyses, which

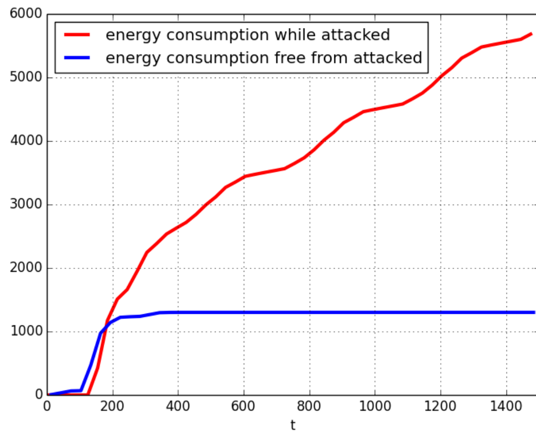

Figure 6 Accumulation of the energy consumption, indicating a security threat to the DC motor controller.

are beyond the present scope.

Scalability to Large Systems Our evaluation of Sock is based on a limited number of randomly generated models, complemented by a single model inspired by a real-world use case. Consequently, the presented results do not directly characterize Sock's behavior on very large or highly complex periodic task-based RTS. Nevertheless, many such systems are commonly structured as collections of relatively independent subsystems, each operating on a limited set of shared resources. In these settings, Sock can be applied to individual subsystems in isolation. The results obtained at the subsystem level can then be combined to reason about the overall system behavior, provided that inter-subsystem interactions are suitably constrained. While we do not evaluate this compositional approach experimentally, it suggests a practical path toward scalability and constitutes an important direction for future work.

Attack Coverage and Countermeasures Second, the set of attacks and countermeasures currently implemented in Sock is limited. In particular, the detection of instability depends on the developer providing an energy consumption function, which may be challenging to implement for complex systems. While it is not feasible to simulate every possible attack scenario, our goal is to provide a design-time exploration tool that helps developers understand system behavior both in normal operation and under attacks. Even this limited set of attacks can significantly improve engineers' awareness of potential security issues. For instance, an engineer can define a mitigation protocol triggered by an instability alert and subsequently test its effectiveness against attacks such as the Butterfly attack, thereby gaining confidence in the system's resilience.

Neglected System Aspects Third, Sock intentionally omits communication aspects of the modeled systems. In practice, periodic tasks often gather data, process it, and may communicate with the cloud or other components for further analysis or control. By excluding communication, we focus specifically on temporal security, which is the central concern of this work. Similarly, Sock does not model authentication, access control, or other non-temporal security mechanisms. While these aspects are critical in practice, they are orthogonal to the timing-focused analysis that Sock enables and can be addressed using complementary tools or methods.

Limitations of the schedulability check The schedulability condition given in Equation 2 is inherited from the Liu and Layland task model and is used in Sock as a sufficient, design-time feasibility check. This condition does not account for additional execution constraints introduced by Sock's security mechanisms, such as temporary resource unavailability due to cleaning operations or scheduling perturbations induced by optional randomization. As a consequence, while satisfaction of Equation 2 establishes baseline timing feasibility, it does not constitute a formal guarantee that all executions produced by Sock will meet deadlines under these extended semantics. Sock therefore treats schedulability as a prerequisite rather than a comprehensive proof obligation, and complementary analyses or simulations are required when security-related mechanisms are enabled. Extending classical schedulability theory to fully integrate these mechanisms is outside the scope of this work and left for future research.

7. Related Work

Research on securing periodic real-time tasks has been prolific and tackled a wide range of scientific problems. We split the existing related work into three categories. First, we present the existing *real-time languages* in the literature. Second, we review the *security for RTS* approaches. Third, we explore the existing *DSMLs that tackle security issues in RTS.*

Real-time languages Le Guernic *et al.* (LeGuernic et al. 1991) introduced *SIGNAL*, a block-diagram language for real-time development with an emphasis on signal processing and control. Halbwachs *et al.* (Halbwachs et al. 1991) proposed *LUSTRE*, a data-flow language based on sequences of values synchronized with a clock. Berry *et al.* (Berry & Gonthier 1992) presented *ESTEREL*, a textual modular language with rigorous behavioral and mathematical semantics, providing stronger formal guarantees than typical synchronous languages for reactive systems. More recently, Berry *et al.* (Berry & Serrano 2020) introduced HipHop.js, extending JavaScript with preemption and synchronous concurrency to simplify the development of complex temporal behaviors. While these languages provide formal semantics and expressive real-time modeling, they focus on system design and reactive behavior rather than security. In contrast, Sock targets real-time systems with shared resources, enabling time-aware attack simulations and analysis of countermeasures, bridging real-time modeling with security-oriented evaluation.

Security for RTS Walls *et al.* (Walls et al. 2019) address control-flow hijacking in periodic task-based applications, but their countermeasure targets a specific class of attacks and does not account for timing or scheduling effects. Pellizzoni *et al.* (Pellizzoni et al. 2015) propose a security model that captures information-leakage constraints between periodic tasks in real-time systems; however, their approach remains declarative and focuses on architectural constraints rather than executable

behavior under concrete schedules. Several works mitigate timing- and schedule-based attacks through runtime obfuscation. MAARS (Sain et al. 2025) demonstrates that fixed-priority controllers leak periodicity through repeating schedules and mitigates this via period randomization and schedule switching, while REORDER (Chen et al. 2018) randomizes EDF execution using bounded priority inversions and evaluates security through schedule entropy. While effective at reducing predictability, these techniques rely on runtime mechanisms and provide limited support for design-time reasoning or systematic exploration of adversarial behaviors. In contrast, Sock adopts a design-time perspective based on an executable DSML with explicit operational semantics for periodic real-time task-based systems, enabling schedulability analysis and the simulation of timing-based attacks under precise execution and resource contention semantics.

DSML for security in RTS Roudier *et al.* (Roudier & Apvrille 2015) propose SysML-Sec, introducing a SysML environment that customizes SysML diagrams to include security aspects. Contrary to SysML-sec, Sock models are directly executable. This execution gives faster feedback than generating code with SysML-Sec and testing it. Tang *et al.* (Tang et al. 2020) introduce a DSML for IoT applications that is translated into Lustre to enable formal verification under synchronous semantics, with a focus on statically checking functional and safety-oriented security properties of event-driven systems. In contrast, Sock targets periodic task-based systems with shared resources and supports the analysis of temporal and scheduler-induced security properties that arise from concrete execution and resource contention. Saadatmand *et al.* (Saadatmand & Leveque 2012) propose a model-based approach in which security requirements are expressed as annotations on the ProCom component model and propagated to derived artifacts, with a focus on architectural-level specification and generation of security requirements. In contrast, Sock focuses on schedulability analysis and simulation of time-based attacks to assess security properties under concrete timing and scheduling behavior. Levendovszky *et al.* (Levendovszky et al. 2014) propose a model-driven framework combining design-time modeling with a runtime platform to deploy secure applications on distributed embedded systems. Their approach enforces security through architectural mechanisms such as partitioning and controlled information flows, providing security guarantees by construction at runtime. In contrast, Sock is based on operational semantics and concentrates on schedulability checking and simulation of timing-based attacks to reason about temporal security properties and adversarial behavior prior to deployment.

8. Conclusion & Perspectives

In this article, we present **Sock**, a clock-based modeling language and toolchain for periodic task-based RTSs that explicitly captures temporal security. Sock allows developers to simulate embedded attacks, verify the confidentiality of sensitive information, and detect potential security violations.

Looking ahead, Sock is designed for extensibility. We plan to support multiple attack scenarios and countermeasures, offering flexibility to address diverse developer needs. We also aim to optimize its operational semantics by associating an FSM with each **Actor/Resource** pair rather than with each entity, reducing the computational complexity of states and transitions and enabling scalability to larger systems.

In the long term, we envision supporting code generation from Sock models using MDE techniques. This would allow developers to produce executable scripts or Java bytecode from Sock **Actors**, deployable directly on physical devices represented as **Resources**.

Acknowledgments

This project was funded by Inria Challenge SPAI. We thank Robert De Simone for the insightful discussions that supported this work.

References

André, C. (2009). *Syntax and Semantics of the Clock Constraint Specification Language (CCSL)* (Tech. Rep. No. 6925). Inria.

Berry, G., & Gonthier, G. (1992). The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of computer programming*, *19*(2), 87–152.

Berry, G., & Serrano, M. (2020). HipHop. js:(A) Synchronous Reactive Web Programming. In *Pldi* (pp. 533–545).

Bousse, E., Degueule, T., Vojtisek, D., Mayerhofer, T., Deantoni, J., & Combemale, B. (2016). Execution Framework of the Gemoc Studio (tool demo). In *Acm sigplan international conference on software language engineering* (pp. 84–89).

Chen, C.-Y., Ghassami, A., Nagy, S., Yoon, M. K., Mohan, S., Kiyavash, N., ... Pellizzoni, R. (2015). *Schedule-Based Side-Channel Attack in Fixed-Priority Real-time Systems* (Tech. Rep.). University of Illinois at Urbana Champaign, USA.

Chen, C.-Y., Hasan, M., Ghassami, A., Mohan, S., & Kiyavash, N. (2018). Reorder: Securing Dynamic-Priority Real-Time Systems Using Schedule Obfuscation. *arXiv preprint arXiv:1806.01393*.

Ciccozzi, F., Crnkovic, I., Di Ruscio, D., Malavolta, I., Pelliccione, P., & Spalazzese, R. (2017). Model-Driven Engineering for Mission-Critical IoT Systems. *IEEE Software*, *34*(1), 46–53.

Deantoni, J., Diallo, I. P., Teodorov, C., Champeau, J., & Combemale, B. (2015). Towards a Meta-Language for the Concurrency Concern in DSLs. In *Design, automation & test in europe conference & exhibition (date)* (pp. 313–316).

DeAntoni, J., & Mallet, F. (2012). Timesquare: Treat your models with logical time. In C. A. Furia & S. Nanz (Eds.), *Objects, models, components, patterns* (pp. 34–41). Berlin, Heidelberg: Springer Berlin Heidelberg.

Fidge, C. (1991). Logical Time in Distributed Computing Systems. *Computer*, *24*(8), 28–33.

Fournaris, A., Pocero, L., & Koufopavlou, O. (2017, 07). Exploiting Hardware Vulnerabilities to Attack Embedded System Devices: a Survey of Potent Microarchitectural Attacks. *Electronics*, *6*, 52.

Halbwachs, N., Caspi, P., Raymond, P., & Pilaud, D. (1991). The Synchronous Data Flow Programming Language LUSTRE. *Proceedings of the IEEE*, *79*(9), 1305–1320.

Harel, D., & Rumpe, B. (2004, 11). Meaningful Modeling: What's the Semantics of "Semantics"? *Computer*, *37*, 64 - 72.

Hasan, M., Kashinath, A., Chen, C.-Y., & Mohan, S. (2024, April). SoK: Security in Real-Time Systems. *ACM Comput. Surv.*, *56*(9). Retrieved from https://doi.org/10.1145/3649499

Kocher, P. C. (1996). Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In N. Koblitz (Ed.), *Advances in cryptology — crypto '96* (pp. 104–113). Springer Berlin Heidelberg.

Lamport, L. (1978, July). Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, *21*(7), 558–565. Retrieved from https://doi.org/10.1145/359545.359563

LeGuernic, P., Gautier, T., Le Borgne, M., & Le Maire, C. (1991). Programming Real-Time Applications with SIGNAL. *Proceedings of the IEEE*, *79*(9), 1321–1336.

Levendovszky, T., Dubey, A., Otte, W. R., Balasubramanian, D., Coglio, A., Nyako, S., . . . Karsai, G. (2014). Distributed Real-Time Managed Systems: A Model-Driven Distributed Secure Information Architecture Platform for Managed Embedded Systems. *IEEE Software*, *31*(2), 62-69.

Lincoln, B., & Cervin, A. (2002). JITTERBUG: A Tool for Analysis of Real-Time Control Performance. In *Proceedings of the 41st ieee conference on decision and control* (Vol. 2, p. 1319-1324 vol.2).

Liu, C. L., & Layland, J. W. (1973, January). Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, *20*(1), 46–61. Retrieved from https://doi.org/10.1145/321738.321743

Mahfouzi, R., Aminifar, A., Samii, S., Payer, M., Eles, P., & Peng, Z. (2019). Butterfly Attack: Adversarial Manipulation of Temporal Properties of Cyber-Physical Systems. In *Ieee real-time systems symposium (rtss)* (p. 93-106).

Mohan, S., Yoon, M. K., Pellizzoni, R., & Bobba, R. (2014). Real-Time Systems Security through Scheduler Constraints. In *2014 26th euromicro conference on real-time systems* (p. 129-140).

Pellizzoni, R., Paryab, N., Yoon, M.-K., Bak, S., Mohan, S., & Bobba, R. B. (2015). A Generalized Model for Preventing Information Leakage in Hard Real-Time Systems. In *21st ieee real-time and embedded technology and applications symposium* (pp. 271–282).

Roudier, Y., & Apvrille, L. (2015). SysML-Sec: A Model Driven Approach for Designing Safe and Secure Systems. In *3rd international conference on model-driven engineering and software development (modelsward)* (pp. 655–664).

Saadatmand, M., & Leveque, T. (2012). Modeling Security Aspects in Distributed Real-Time Component-Based Embedded Systems. In *2012 ninth international conference on information technology-new generations* (pp. 437–444).

Sain, A., Adhikary, S., Koley, I., & Dey, S. (2025). MAARS: Multi-Rate Attack-Aware Randomized Scheduling for Securing Real-time Systems. In *Proceedings of the acm/ieee 16th international conference on cyber-physical systems (with cps-iot week 2025).* New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3716550.3722030

Tang, W., Feng, H., Hisazumi, K., & Fukuda, A. (2020). A Verification Method for Security and Safety of IoT Applications Through DSM Language and Lustre. In *Proceedings of the 3rd international conference on information science and systems* (p. 166–170). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3388176.3388211

Völp, M., Hamann, C.-J., & Härtig, H. (2008). Avoiding Timing Channels in Fixed-Priority Schedulers. In *Acm symposium on information, computer and communications security* (p. 44–55). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1368310.1368320

Walls, R. J., Brown, N. F., Le Baron, T., Shue, C. A., Okhravi, H., & Ward, B. C. (2019). Control-Flow Integrity for Real-Time Embedded Systems. In *31st euromicro conference on real-time systems (ecrts 2019).*

Wolf, M., & Serpanos, D. (2018). Safety and Security in Cyber-Physical Systems and Internet-of-Things Systems. *Proceedings of the IEEE*, *106*(1), 9-20.

Wysocki, R. J. (n.d.). *CPU Idle Time Management.* https://www.kernel.org/doc/html/v5.0/admin-guide/pm/cpuidle.html. (The Linux Kernel, Accessed: November 24, 2025)

Xu, T., Wendt, J. B., & Potkonjak, M. (2014). Security of IoT Systems: Design Challenges and Opportunities. In *Ieee/acm international conference on computer-aided design (iccad)* (p. 417-423).

Yoon, M., Mohan, S., Chen, C., & Sha, L. (2016). TaskShuffler: A Schedule Randomization Protocol for Obfuscation against Timing Inference Attacks in Real-Time Systems. In *2016 ieee real-time and embedded technology and applications symposium (rtas)* (p. 1-12).

Zhao, K., & Ge, L. (2013). A Survey on the Internet of Things Security. In *9th international conference on computational intelligence and security* (p. 663-667).

About the authors

Stéphanie Challita is Associate Professor of software engineering at the University of Rennes. Her work focuses on model-driven engineering, software language engineering and reverse engineering, mostly applied to distributed systems such as cloud computing, real-time systems and REST APIs. You can contact the author at stephanie.challita@irisa.fr or visit https://stephanie.challita.fr.

Benoit Combemale is Research Director at Inria, on leave from his position as Full Professor of software engineering at the University of Rennes. He is interested in software engineering, including model-driven software and systems engineering, software language engineering and software validation & verification (V&V); mostly in the context of (smart) cyber-physical systems and Internet of things. You can contact the author at benoit.combemale@irisa.fr or visit https://combemale.fr.