Temporal-Based Intrusion Detection for IoV

Article *in* it - Information Technology · December 2020

DOI: 10.1515/itit-2020-0009

Mohammad Hamad*, Zain A. H. Hammadeh, Selma Saidi, and Vassilis Prevelakis

Temporal-based intrusion detection for IoV

https://doi.org/10.1515/itit-2020-0009
Received March 14, 2020; revised October 19, 2020; accepted October 20, 2020

Abstract: The Internet of Vehicle (IoV) is an extension of Vehicle-to-Vehicle (V2V) communication that can improve vehicles' fully autonomous driving capabilities. However, these communications are vulnerable to many attacks. Therefore, it is critical to provide run-time mechanisms to detect malware and stop the attackers before they manage to gain a foothold in the system. Anomaly-based detection techniques are convenient and capable of detecting off-nominal behavior by the component caused by zero-day attacks. One significant critical aspect when using anomaly-based techniques is ensuring the correct definition of the observed component's normal behavior. In this paper, we propose using the task's temporal specification as a baseline to define its normal behavior and identify temporal thresholds that give the system the ability to predict malicious tasks. By applying our solution on one use-case, we got temporal thresholds 20–40 % less than the one usually used to alarm the system about security violations. Using our boundaries ensures the early detection of off-nominal temporal behavior and provides the system with a sufficient amount of time to initiate recovery actions.

Keywords: Real-time systems, Security, Safety, Intrusion Detection

ACM CCS: Security and privacy → Intrusion/anomaly detection and malware mitigation → Intrusion detection systems

***Corresponding author: Mohammad Hamad,** Technical University of Munich, Department of Electrical and Computer Engineering, Munich, Germany, e-mail: mohammad.hamad@tum.de, ORCID: https://orcid.org/0000-0002-9049-7254
Zain A. H. Hammadeh, Institute for Software Technology, German Aerospace Center (DLR), Cologne, Germany, e-mail: zain.hajhammadeh@dlr.de
Selma Saidi, Technical University of Dortmund, Department of Electrical Engineering and Information Technology, Dortmund, Germany, e-mail: selma.saidi@tu-dortmund.de
Vassilis Prevelakis, Technical University of Braunschweig, Institute of Computer and Network Engineering, Braunschweig, Germany, e-mail: prevelakis@ida.ing.tu-bs.de

1 Introduction

A modern car contains 70–100 small embedded computers, known as Electronic Control Units (ECUs). Most of these ECUs rely on a set of sensors and actuators to serve one or more of the Electrical and Electronic (E/E) systems or subsystems in the vehicle. These (sub)systems range from the mundane, such as controlling courtesy lights, to highly critical applications, such as engine control. These ECUs are grouped into various sub-networks based on their functions. The sub-networks are interconnected via a central gateway. The ECUs within each sub-network communicate via different bus systems. Modern vehicles are also equipped with various technologies, such as WiFi, 5G, GPS, and Bluetooth, giving them the capability to collaborate and communicate with roadside units to ensure a safe and comfortable journey for drivers and passengers. The increase in vehicle connectivity is a double-edged sword. On the one hand, it extends the vehicle's functionalities and capabilities. On the other hand, it opens the door to several cybersecurity threats and makes the car a more attractive target for adversaries.

Using the traditional mechanisms to eliminate vulnerabilities and to ensure the vehicular system security is insufficient by itself. As vehicle complexity grows, the likelihood of hidden vulnerabilities and threats increases. Therefore, it is critical to provide run-time mechanisms to detect the malware and stop the attackers before they manage to gain a foothold in the system. E. g., Using signature-based detection mechanisms is inadequate since it cannot keep up with the successive and unknown attacks which may target the vehicle during its operational life.

Figure 1 depicts an exemplary distributed system of in-vehicle components communication. In an idealistic world, each ECU is running a Run-Time Environment (RTE) that hosts multiple (interacting) software components. These components are safety and non-safety critical applications. The ECUs are sharing the bus system. Traditionally, internal vehicle bus systems have been developed to fulfill safety, cost, and efficiency requirements without considering any security risks. Ignoring the security issues left the vehicle under a huge risk whenever an attacker can access the bus system. An attacker can: interfere the communication and try to inject malicious messages (e.g, m_4 and m_5) and makes the receiv-

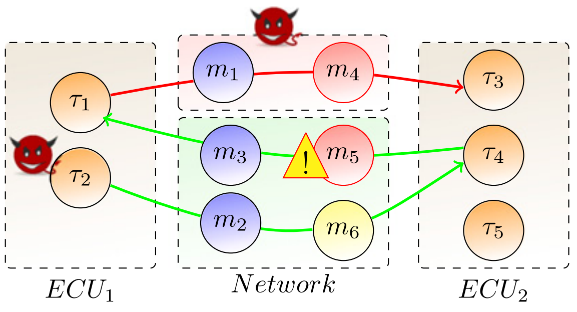

Figure 1: System and threat model of in-vehicle communication.

ing node think that messages are originated from a legitimate ECU (i. e., *masquerade attack*), intercept all the messages exchanged between the different ECUs (i. e., *man-in-the-middle attack*), manipulate the transmitted data (i. e., *spoofing attack*), or fraudulently delay or re-transmit previous messages (i. e., *replay attack*). Besides, an attacker can flood the bus with fake high priority messages to disrupt other communications (i. e., *Denial of Service (DoS) attack*).

Adopting security mechanisms that ensure data integrity, confidentiality, authenticity, and authorization can protect the in-vehicle network system from most of these attacks and facilitate the detection of infected messages (such as m_5) [9, 8]. However, a malicious software component (such as τ_2) might remain able to emit malicious packets (e. g., m_6) to its remote peer(s), if it is authorized. Some solutions proposed using the period of the transmitted messages over the bus as a reference to detect any off-nominal behavior, including dropping and injecting malicious messages [21, 19]. But it is crucial to know that the in-vehicle messages do not follow consistent timing intervals all the time. The changes in regular driving operation can change these timing intervals [24]. Also, sporadic messages, by its nature, do not have such a fixed period between any two consecutive messages. These two cases make many of the network-based IDSs that depend on the period not reliable. Therefore, in this paper, we decided to consider the component's behavior that emits the malicious messages instead of looking at the messages' behavior over the network.

Anomaly-based detection techniques seem to be more convenient and capable of detecting the off-nominal behavior, which could be caused by zero-day attacks. Anomaly-based mechanisms monitor the component's behavior to identify any misuse that falls outside the predefined profile representing the nominal behavior of that component. A software component's nominal behavior can be determined based on different component properties such as system calls, memory consumption, etc. In this paper, we propose using the component's temporal specification a baseline to define its nominal behavior. Attacks such as code injection or DoS attacks will usually cause a breach of this temporal specification, and thus it will be detected.

The most critical aspect when using anomaly-based techniques is ensuring the correct definition of the component's nominal behavior. Using inaccurate or wrong behavior as a reference for off-nominal behavior detection may cause a high rate of false-positive and false-negative errors. One example of an improper nominal behavior definition, as we will see in this paper, is the use of safety-driven boundaries that are used to define nominal behavior from a safety perspective to identify security-based nominal behavior. This paper extends our previous work [7].

Contribution: in this paper:
- We introduce the Red-Zone principle, which aims to relax of a predefined boundary in case they are very tight or to tight them in case they are over-approximated (see section 2.1). Also, we present the required steps to adopt this principle (see section 2.2).
- We show how a safety-based temporal specification is not useful to detect the malicious behavior of the observed software component (see section 3).
- We identify the proper temporal thresholds which can be used to adopt Red-Zone Principle which give the system the ability to predict the presence of malicious tasks (see sections 4 and 5).
- We present the design and the implementation of the intrusion prediction schema based on the Red-Zone principle (see section 6).
- We explain the adoption of our proposed system using a real use case (see section 7).

2 Foundation

2.1 Red-Zone principle

Every task, in a given system, is usually designed to accomplish a particular work and behave in a well-defined way. However, at run-time, a task may not exercise all the intended functionality and, in some circumstances, a task may stray outside its planned operational profile for many reasons such as it could be under an attack or it is already taken over and it is carrying out the attacker's commands,

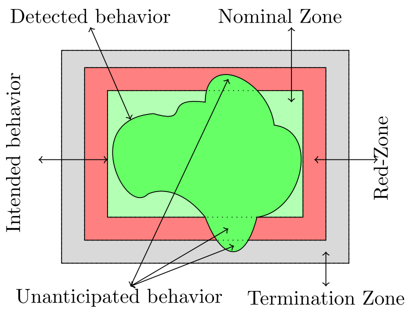

Figure 2: Red-Zone principle and the different possible operational zones of a task [7].

the task has encountered an error due to a hidden software bug as a result of insufficient testing, etc.

However, straying outside the predefined policy is not always a sign of malicious behavior. The task may attempt to perform an action that is legal but not permitted by the existing policy. This circumstance occurs when an incomplete security policy is adopted. Such security policies are typically defined using under-approximated or very limited rules due to the preliminary test, which did not cover all the implemented functionalities. The security policy in many systems is defined as binary actions (authorized, not authorized). The system keeps a task running while it behaves correctly by staying within its designated operational profile; otherwise, it is terminated. Terminating a perfectly functional component because of a violation, such as an overflow, may have spectacularly unintended consequences.

Therefore, rather than terminating the task, we introduce the *Red-Zone* principle to permit the task to overrun its designed operational profile until an ultimate limit but in an observed mode. This window of observation is called the Red-Zone. Whenever the task exceeds the ultimate limit, it will be terminated (or other option can be adopted based on recovery policy), leaving behind it an audit trace with potential information about the vulnerability or the fault. Figure 2 illustrates the possible operational behaviors of a task. The four main areas represent: the intended behavior, the actual behavior, the Red-Zone, and the termination zone for a given task.

An essential benefit of red-zoning is that, by assuming the attack is detected, it allows an attack to be monitored in real-time while the subverted application cannot cause problems to the system. Another critical goal of red-zoning is to provide early detection of off-nominal behaviors that will give the system the ability to respond effectively.

2.2 Adopting the Red-Zone principle

To adopt the Red-Zone principle, the following three-stage process needs to be accomplished for each task:

- Establish the nominal behavior of the task.
- Formulate a security policy which defines the various execution areas for the task based on observations made in the previous step. Moreover, the security policy should determine the appropriate actions within each zone.
- Implement a framework to enforce the security policy while the state of the task is changing, reflecting the execution zones (nominal, red-zone, off-nominal).

Many mechanisms were used to monitor the nominal behavior of a task based on different program properties such as power consumption [13], system call distribution [23], system calls sequences [1], and so forth. Applying these mechanisms in the vehicular domain entails several challenges, including the limited computing and energy resources of most ECUs. In this work, we investigate the use of another property. The modern vehicle contains safety-critical and non-safety-critical applications. One of the main characteristics of safety-critical applications is that they have strict temporal constraints. The safety-critical system's timing behavior could be used by an anomaly-based detection mechanism to detect various attacks against such systems.

2.3 Properties of real-time systems

In real-time systems the correctness of the system depends not only on the functional results of the computations, but also on the time at which the results are produced [20]. A reaction that occurs too late could be useless or even dangerous. For example, the anti-lock braking system (ABS) on a vehicle needs to react to the sensor on the wheels and prevent the wheels from being locked up within a specified period; after this period, the vehicle may skid or crash into something.

Real-time systems' performance has to be validated against timing constraints to guarantee that the results are available when they are needed. Timing analysis is often used to provide off-line safe upper bounds on the tasks execution and response times to cover the set of all possible executions, including corner-cases, which are the smallest (best) and largest (worst) values that can be observed at runtime. Before we go further, let us look in details at real-time tasks and their different properties and timing constraints.

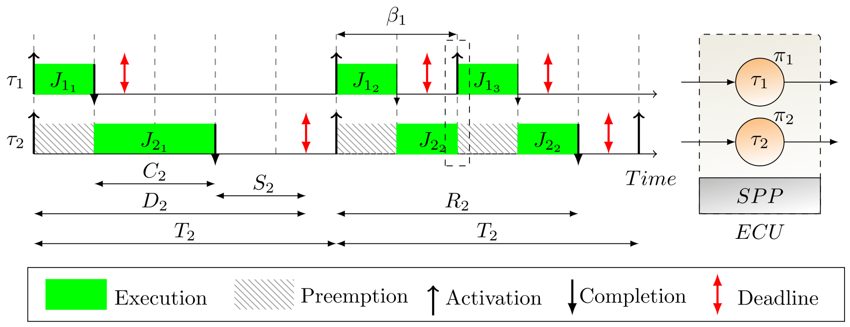

Figure 3: Timing properties of real-time tasks. Dashed rectangle represents the effect of SPP scheduling policy [3] where τ_1 (has the highest priority) takes over the CPU previously allocated to τ_2 (has the lowest priority), thereby interrupting its execution.

2.3.1 Real-time tasks

Each real-time task has number of execution requirements illustrated in Figure 3. Every real-time task τ_i is defined by a tuple $\tau_i = \langle C_i^+; D_i; T_i \rangle$ where C_i^+ refers to the worst case execution time (WCET), D_i refers to the relative deadline, and T_i refers to the minimum interarrival time with $D_i \leq T_i$. Each task is assigned a *priority* π_i, if a task is activated regularly, we call it *periodic*, otherwise we call it *sporadic*, and β_i is used to characterizing the minimum distance between two consecutive activation. Every instance of τ_i is called a *job* J_{ij}. When a job of τ_i is *activated* and becomes ready for execution, it occupies the resource (typically the CPU) for a duration no longer than C_i^+ of execution. The longest execution time (i. e., C_i^+) can be derived using worst-case execution time analysis. The point of time when the task finishes its execution called *completion* time. The idle time between the completion of a job of τ_i and its deadline is called a *slack* S_i. Such a slack can be used for the computation of that job without causing it to miss its deadline [6]. Real-time task executions are constrained in time to guarantee the correctness of the system. This time constraint is the relative *deadline* D_i which is the maximum allowable time for a given task to complete its execution. The consequences of missing the deadline can be used to distinguish between two types of real time tasks:

- ***Soft* real-time tasks:** Missing a deadline of such tasks is tolerable. It may cause a degradation in the quality of service. In most cases, non-safety-critical tasks are considered as soft real-time tasks.
- ***Hard* real-time tasks:** Missing a deadline of such tasks is intolerable. It could lead to catastrophic situations. All the safety-critical tasks are considered as hard real-time tasks.

Similar to that, a (sub-)system of a vehicle is considered as a *Hard* real-time system when it contains at least one *Hard* real-time task. On the other hand, when all the (sub)system's tasks are *Soft* real-time task, we refer to that (sub-)system as a *Soft* real-time system.

2.3.2 Worst-case response time

When multiple real-time tasks are mapped to the same ECU, they share and compete for the same hardware resource, e. g., CPU. The execution of these tasks is controlled by a scheduler which decides at each time which task can be executed on the CPU. Different scheduling policies are adopted by the scheduler to arbitrate between the various tasks [4]. SPP scheduling [3] is a well-known scheduling algorithm in which the scheduler will always allocate the resource to the ready job of the highest priority task (see Figure 3). Therefore, a lower priority task may be preempted by a higher priority one.

The *response time* R_i then characterizes the time which spans between the activation of a job and its completion. Response time analysis [18] is used to determine the worst-case response time (WCRT), it computes an upper bound on the response time of a task and validate it against its deadline D_i. The busy-window analysis is a well established technique for computing the WCRT. It computes the maximum processing time $B_i^+(q)$ which is necessary to process q jobs of a task τ_i. $B_i^+(q)$ under an SPP scheduler is defined as follows:

$$B_i^+(q) := q.C_i^+ + \sum_{j \in hp(\tau_i)} \left\lceil \frac{B_i^+(q)}{T_j} \right\rceil .C_j^+ \quad (1)$$

where, $hp(\tau_i)$ is the set of tasks with higher priority than π_i. $\left\lceil \frac{B_i^+(q)}{T_j} \right\rceil$ is the maximum number of jobs from interfering tasks τ_j during a time window of size $B_i^+(q)$. For sporadic tasks, T_j should be replaced by β_j. Note that, the computation of the busy-window is a fixed point computation as $B_i^+(q)$ appears in both sides of the equation, which can be solved iteratively, starting with $B_i^+(q) := q.C_i^+$. The computation of $B_i^+(q)$ is proven to converge for some $Q_i \in \mathbb{N}$ and the length of the busy-window is computed as $B_i^+(Q_i)$ [3]. The q-th job is activated at $(q-1)T_i$, considering $t = 0$ at the beginning of the busy-window. Therefore, the response time of the q-th job is $B_i^+(q) - (q-1)T_i$. R_i^+, the WCRT of task τ_i, is then computed as follows:

$$R_i^+ := \max_{1 \le q \le Q_i} \{B_i^+(q) - (q-1)T_i\} \tag{2}$$

We will use a motivational example which represents a hard real-time system that includes three periodic tasks $\tau_2 = (1,4,4)$, $\tau_3 = (1,5,5)$, and $\tau_4 = (2,8,8)$ ordered from the highest to the lowest priority. The system also contains one sporadic task τ_1 with execution time $C_1^+ = 1\,\text{ms}$ and $\beta_1 = 20$. The task τ_1 has the highest priority but is rarely activated. By using Equation (2), we can compute the WCRT for every task of the system as follow (see Figure 7-b): $R_1 = 1$, $R_2 = 2$, $R_3 = 3$, $R_4 = 7$.

This work considers that the studied system is a hard real-time system with sporadic and periodic tasks and uses a uni-processor and SPP scheduling policy. Also, determining and exploring the required action that needs to be applied when a task's execution exceeds its deadline is beyond the paper's scope.

3 Time-based intrusion detection approach

As the vehicle increasingly becomes more interconnected with the Internet, the different hard real-time (sub-)systems become an attractive target of various attacks. However, these systems have the advantage that they are predictable by design, as they should comply with temporal constraints.

Therefore, from the security point of view, we refer to any malicious activity which alters the predefined temporal behavior of a real-time task and causes it to miss its deadline as a temporal attack. The temporal attack could be caused by deliberate attacks, such as the code injection attack, when an attacker tries to exploit an existing vulnerability (e. g., buffer overflow [2]) to execute injected or preexisting malicious code to break or force the system to do malicious actions [14]. The deadline miss may be the ultimate objective of the attack (e. g., a denial-of-service attack (DoS)), or only a side-effect of the attack as the process now needs to carry out both its normal tasks and the actions of the attacker. If the resources allocated are tight,

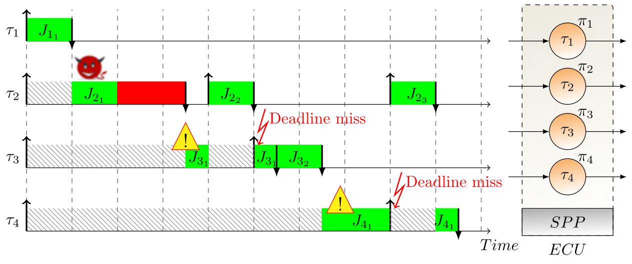

Figure 4: Using the WCRT and deadline miss as an indicator of compromise could lead to false identification of malicious tasks.

the extra effort will result in slower response time and potentially missed deadlines.

In hard real-time systems, a task is considered off-nominal whenever it violates its temporal constraint (i. e., deadline). Therefore, an intuitive approach is to use the deadline as a demarcation line that indicates the task's off-nominal temporal behavior. However, using the deadline miss as a sign to predict the off-nominal behavior is insufficient for many reasons. Firstly, waiting for the task to exceed its deadline to trigger an alarm about the off-nominal state is useless, especially in critical-safety hard real-time systems, because it turns to be too late for the system to avoid the attack, we refer to this as *belated detection*. Secondly, the deadline miss of one task may cause a cascading deadline misses for lower priority tasks as shown in Figure 4, where the deadline miss of τ_3 causes τ_4 to miss its deadline too. Finally, the use of deadline-based monitoring may not always identify the off-nominal task correctly. Indeed, the interference on a high priority task may lead the lower priority tasks to miss their deadline. In this case, the deadline-based detection will report an issue in the lower priority tasks without any sign about the high priority task's misbehaving. In Figure 4, monitoring system will report both τ_3 and τ_4 as malicious tasks since they miss their deadline although τ_2 is the one should be identified since it is the actual task which causes this issue.

AUTOSAR, for instance, also recognized the problems associated with the deadline-based monitoring approach. Therefore, it chose to monitor the execution time of tasks rather than the deadline to ensure the time protection. This mechanism's goal is to prevent fault propagation by killing the task which exceeds the WECT [5]. However, as mentioned previously, this extreme solution may not be suitable for hard-real-time systems, as it may cause the system to enter an unstable state. Zimmer et al. [25] use the WCET timing bound to detect code injection attacks by comparing the WCET value with the elapsed time along the return path. Petal et al. [15] proposed an approach to monitor the execution time of the running tasks. They use a dedicated security processor to supervise the application processors. They change the task's binary file by inserting instructions at the start and the end of each block to instrument the execution time for each block. At run time, the security processor will determine whether a given block takes longer than its defined WCET.

Therefore, defining other reasonable temporal boundaries to delimit the off-nominal behavior of real-time tasks is required to prevent the propagation of the temporal constraints violations between tasks and support the early detection of the system's malicious tasks. We aim to define these boundaries by adopting the Red-Zone principle.

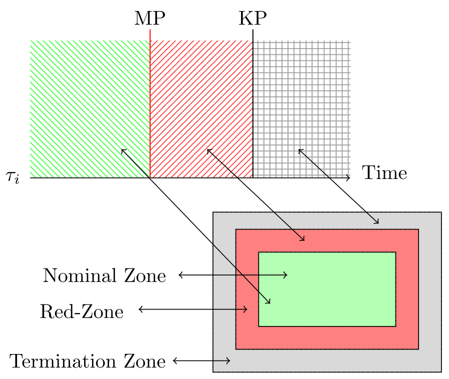

Figure 5: Linking the Red-Zone areas to the temporal line of the real-time task [7].

4 Red-zone boundaries for real-time systems

We extend the use of the Red-Zone principle to establish appropriate prediction configurations that can be adopted by real-time systems. These configurations define the monitoring boundaries that are used later by the prediction scheme to detect the task's off-nominal behavior. The adoption of the Red-Zone principle is achieved by an offline analysis to define for each real-time task the next properties:
- The **nominal temporal behavior** of each task.
- A **Monitoring Point (MP)**: it is a point in time that represents the border line between the nominal temporal behavior and the potential off-nominal one; see Figure 5. The temporal behavior of a task becomes suspicious whenever this point is passed.
- A **Killing Point (KP)**: it is another point in time (see Figure 5); the temporal behavior of a task becomes certainly off-nominal whenever this point is exceeded.

Within each zone, different response strategies can be adopted such as terminating the malicious task whenever it exceeds the KP [10].

Two key features determine the efficacy of any proposed mechanism for off-nominal behavior detection, namely the False Positives Rate (FPR) and the precision values. FPR, which is defined in Equation (3), identifies the number of suspicious tasks that complete their jobs before the killing point over the number of total monitored ones. Any effective prediction technique should ensure a minimum value of its FPR.

$$FPR = \frac{\#\{R_i \mid R_i \geq MP \quad \& \quad R_i < KP\}}{\#\{R_i \mid R_i \leq KP\}} \tag{3}$$

The precision, which is defined in Equation (4), is used to determine the accuracy of the off-nominal behavior detection scheme. The higher value of this factor, the more accurate the prediction schema. The next equation defines the precision:

$$Precision = \frac{\#\{R_i \mid R_i \geq KP\}}{\#\{R_i \mid R_i \geq MP\}} \leq 1 \tag{4}$$

As mentioned previously, the Red-Zone principle defines a time zone where we can start monitoring the system to safely detect a possible malicious behavior and act on-time accordingly, before we consider the extreme solution of terminating a task or a system.

5 Response time-based prediction configuration

As we described in Section 4, we need to define the response time-based temporal nominal behavior of the hard real-time task. Consequently, we need to determine both MP and KP.

5.1 Defining temporal nominal behavior and MP

In real-time systems, the WCRT can be used to characterize the system. An intuitive approach to designate the Red-Zone is to use WCRT as an MP. Starting the monitoring at the WCRT allows detecting attacks that solely push the response time of the system to be larger than the worst-case. However, waiting until the WCRT to start tracing the task is, in many cases, too late. Allowing the response time to exceed the WCRT may cause a cascading deadline miss

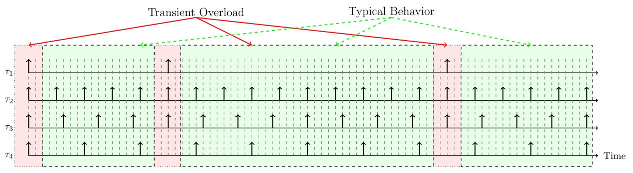

Figure 6: Execution trace of τ_1, τ_2, τ_3, and τ_4.

for lower priority tasks as shown in Figure 4 where the response time of τ_2 has exceeded R_2^+ which causes both τ_3 and τ_4 to miss their deadlines sequentially.

Furthermore, in a real-time system, which contains periodic and sporadic tasks, the WCRT does not reflect the ordinary nominal behavior boundary of the system's tasks. WCRT analysis takes into account the worst-case scenario where the maximum interference (i. e., load) is considered. However, in many cases, particularly in the presence of a transient overload (i. e., sporadic tasks), the frequency of activation of these tasks is very low, and therefore the worst-case scenario with maximum load may never happen. But, it is nevertheless considered, which results in very large (pessimistic) bounds on the response time.

Figure 6 shows the execution trace of the four tasks in our example for a period. The figure shows that the temporal transient overload does not exist (as in the dashed, green rectangle) most of the time.

Typical Worst-Case Analysis [17] (TWCA) was introduced to take into account the activation frequency of sporadic tasks and therefore define a much smaller/tighter bound on the response time accompanied by the frequency at which this bound might be violated based on the frequency of the sporadic load [11]. Consequently, the purpose of the typical worst-case is to get as close as possible to the most probable case to occur at the run-time leading to typical (and not worst-case) performance. In other words, it is used to define the nominal behavior boundary of the tasks.

By applying TWCA analysis for every task τ_i in the system, we get:

– a typical worst-case response time bound $TWCRT_i$ which could be used as MP_i for the task τ_i, along with
– for every $k \in \mathbb{N}^+$, a bound on the number of executions of τ_i (let us say m) which may have a response time larger than $TWCRT_i$ in a window of k consecutive executions. We refer to it as *m out of k*. This bound represents the maximum permitted FPR_i of our prediction schema for the task τ_i.

In more formal and generic way, we can define the *m out of k* as follows: in any sequence of k jobs of τ_i (k consecutive execution of τ_i) there are at most (at least) m jobs (do not) satisfy a given property. In the context of TWCA, the given property is $R_i^+ \leq TWCRT_i$. We look to the sequence of k as a *sliding window* where its size is finite and fixed, containing k jobs. This window slides along to the infinity and always there must be no more than m jobs in this window with $R_i^+ \leq TWCRT_i$.

Indeed, we can define the nominal temporal behavior of task τ_i based on this analysis that the nominal response time of task τ_i is between $]0, TWCRT_i]$.

TWCA refers to the response time of the task τ_i without the transient overload (i. e., ignoring the sporadic tasks) as a *typical* worst-case response time $TWCRT_i$ for this task. Figure 7 shows the two relative response times; left part of the figure shows the *TWCRT* of the tasks while the right part shows the *WCRT* which we already calculated. From the same figure we can see the difference between the *TWCRT* of τ_4, which is equal to 4 ms, and the R_4^+, which is equal to 7 ms.

By unfolding multiple executions of the periodic tasks (i. e., τ_2, τ_3, and τ_4), one can observe an upper bound of how many times out of the observed executions when those tasks may coincide with τ_1 and have a response time larger than its *TWCRT*. For example, by considering the execution trace presented in Figure 6 and by calculating the response time of the three periodic tasks during the tracing period, we find that out of 20 consecutive executions of τ_2, only 4 observed R_2 were longer than $TWCRT_2$. And, out of 16 consecutive executions of τ_3, only 2 observed R_3 were longer than $TWCRT_3$. For task τ_4, out of 10 consecutive executions, only 2 observed R_4 were longer than $TWCRT_4$.

5.2 Defining KP

After defining the MP, we need to determine the other bound which encloses the Red-Zone area (i. e., KP). In *hard* real-time systems, a task becomes malicious whenever its response time exceeds its deadline. Therefore, the deadline is one potential candidate to be adopted as KP. However, using the deadline as KP is not suitable; keeping the task running until its deadline will cause a cascading deadline miss of the low priority tasks, as we have seen before.

Another proposal is using the WCRT of the task as a KP. However, the task will be under observation as soon as it enters its Red-Zone area. This observation will introduce some overhead. We refer to it as the Monitoring Overhead (MOV).

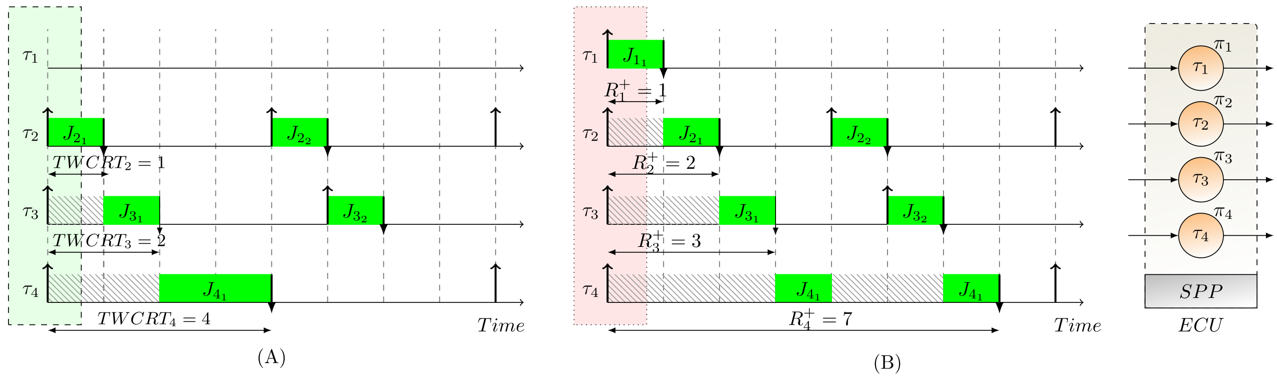

Figure 7: The TWCRT and WCRT for the τ_2, τ_3, and τ_4.

Definition 5.1 (Monitoring Overhead (MOV)). *The extra time that task needs to consume as result of being in its Red-Zone area (i. e., under monitoring).*

$$MOV_i = C_i^{rz} \times f_i \tag{5}$$

where, C_i^{rz} represents the maximum execution time that task τ_i still need to consume during its Red-Zone area (i. e., $R_i^+ - TWCRT_i$), and f_i is a factor which represents, for each task, the factorized overhead. F By considering the worst-case execution time of the task τ_i when the task is fully monitored is $\bar{C}_i^+$, then the f_i can be calculated as $f_i = \frac{\bar{C}_i^+ - C_i^+}{C_i^+}$. It is worth to mention that the real-time system's monitoring is a vast topic by itself, and there is much research about how to implement and optimize such a mechanism [16, 22].

The monitoring overhead MOV_i is different for every task τ_i and is proportional to the size of its Red-Zone. With the presence of MOV, using WCRT as a KP may cause killing (if the security policy requires the termination of the task when it passes its KP) a benign task since the WCRT of this task was calculated without considering such an overhead. Thus, we defined another time point for each task to represent its KP, called the Early-warning Deadline (ED).

Definition 5.2 (Early-warning Deadline (ED)). *ED_i of task τ_i is the maximum time for τ_i to complete its execution with the presence of the monitoring overhead MOV_j from all interfering tasks.*

$$ED_i = C_i^+ + MOV_i + \sum_{j \in hp(\tau_i)} \left\lceil \frac{ED_i}{T_j} \right\rceil .(C_j^+ + MOV_j) \tag{6}$$

with initial value $ED_i = C_i$.

We constraint the monitoring overhead by the available idle time (i. e., slack) to safely monitor the tasks without jeopardizing the schedulability of the task or lower priority tasks:

$$\sum_{j \in H} \left\lceil \frac{\max_{i \in H} D_i}{T_j} \right\rceil .MOV_j \leq \min_{l \in H} S_l \tag{7}$$

where S_l represents the slack of task τ_l, and H represents the set of all hard real-time tasks in the system.

Lemma 1.

$$R_i^+ \leq ED_i \leq D_i \tag{8}$$

Proof. $\forall i \in H : MOV_i \geq 0$. Substituting in Equation (6) we get $R_i^+ \leq ED_i$. Also we know that for each real-time task [6]:

$$R_i^+ + S_i \leq D_i$$

From Equation (7) and (6), we get $ED_i \leq D_i$. □

Corollary 1. *If τ_i and all higher priority tasks entered the Red-Zone, the presented monitoring overhead will not cause deadline misses in the lower priority task.*

Figure 8 shows the *ED* of each task in our example based on Equations (6) and (7). Note that for the sake of simplifying the calculation of ED_4, we assumed that the overheads sum is equal to the minimum slack (i. e., $\sum_{k \in H} MOV_k = \min_{k \in H} S_k = 1$) and $f = 1/5$ for all tasks.

6 Design and implementation

To implement our prediction schema, we need to consider two main concepts: (1) the new task state (i. e., Monitored Running) and (2) the monitoring policy, which is integrated into the scheduler. In the remainder of this section, we elaborate on these concepts in more detail.

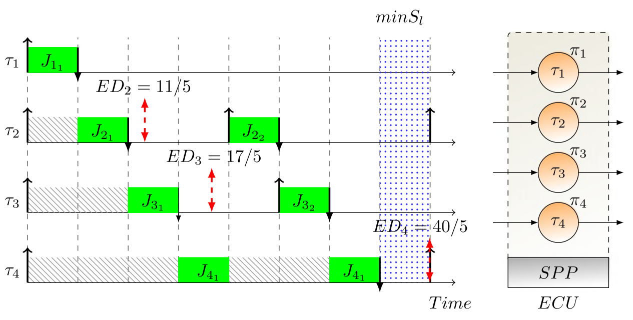

Figure 8: The ED for the τ_2, τ_3, and τ_4.

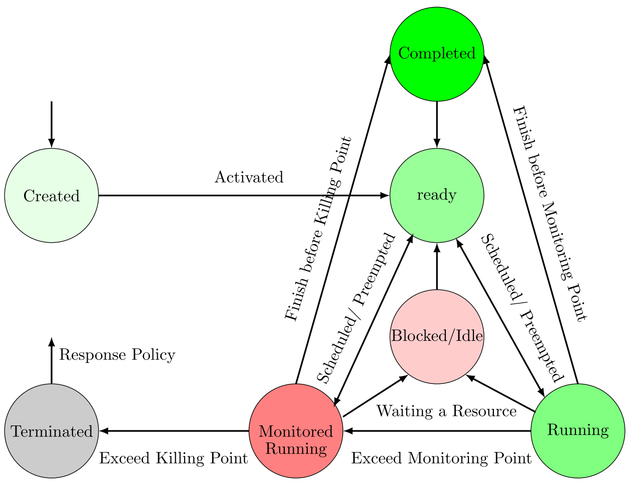

Figure 9: Monitored Running state and its relation with other states [7].

6.1 Task states

Basically, each real-time task is designed to have a recurrent nature. During its life cycle, it can be in one of three different states: either it is *Ready* to run, it is already *Running*, it is *Blocked* (i. e., it is waiting for some resources, or it is *Idle* waiting for a specific timeout). In our proposed scheme, we introduced another state called *Monitored Running* state, as shown in Figure 9. Whenever the task exceeds its monitoring point, it will enter to the *Monitored Running* state where it is running under observation. During its running in this state, the task could be blocked waiting for a resource, or another task could preempt it. It could complete its execution before the killing point (KP), or it may breach its KP, which would force it to perform certain actions based on the applied security response policy.

The implementation of such a state requires the modification of the task's TCB by adding a new field to identify the state of the task. This Boolean field will be checked whenever the task exceeds its monitoring point.

6.2 Monitoring policy

Whenever a new task is created, the defined prediction configurations for this task should be added to its TCB. Therefore, we need to modify the task's TCB to include another five fields: The first field is called Monitoring Time (MT), which holds the value of the monitoring point. The second field, called Killing Time (KT), contains the value of the killing point. The third field will hold *m*. The fourth one will hold *k*, and the last field will contain an array of *k* integer values (*task_window*), which will be used to trace how many tasks have exceeded the MT in the last *k* jobs.

All these values are used by the integrated monitoring policy, as described in Algorithm 1. Whenever the task is scheduled to run, the system checks whether the running task is a new job, so the number of jobs (*t_c*) needs to be increased (line 4). After that, the MT value is loaded into a timer (line 6). The task is kept executing normally without any tracing (lines 9–14) while its execution time does not exceed the MT (i. e., the timer is exhausted). In the meantime, if a task with higher priority shows up, the context switching takes place to replace the current task (lines 11–13).

Whenever the timer reaches its limit (i. e., task's execution time passed the MT), the task's state is changed to ‘monitored’, and the value in the *task_window* cell, which refers to the current task, is updated with value 1. Whenever the number of the cells that include 1s is larger than *m*, the system needs to be informed that an attack occurs since the task violated its *moutofk* condition (lines 19–22). The task is kept under monitoring until it finishes, or it exceeds its KT, and at that point, the system needs to be informed about the breach (lines 23–30).

7 Evaluation

To evaluate our approach, we use the Adaptive Cruise Control (ACC) system as a use case. ACC is a common and well-known automotive driver assistance that keeps the vehicle at a steady speed entered by the driver without keeping the accelerator pedal pressed down continuously and maintains a safe headway distance between cars in the same lane. Internal sensors such as LiDAR and a camera are used to provide information about the car's speed ahead, which helps the ACC system adjust its distance from it by controlling the engine and brakes.

Algorithm 1 LoadTask: using the prediction configuration while loading the ready task.

```
Require:
1:  TIMER timer
2:  procedure LOADTASK( TASK cur_t )
3:      if (cur_t is new_job) then
4:          $t\_c = t\_c + 1$
5:          $task\_window$[t_c%k] = 0
6:          $timer.value = cur\_t.TCB.MT$
7:      end if
8:      if (cur_t.TCB.monitored == False) then
9:          while ($timer.value$ >0 & cur_t not Done) do
10:             run_without_tracing(cur_t)
11:             if (Preemption) then
12:                 context_switching()
13:             end if
14:         end while
15:         $cur\_t.TCB.monitored = True$
16:         GoTo 18
17:     else
18:         $timer.value = cur\_t.TCB.KT - cur\_t.TCB.MT$
19:         $task\_window$[t_c%k] = 1
20:         if ($sumof(task\_window) > m$) then
21:             $apply\_recovery\_policy()$
22:         else
23:             while ($timer.value$ >0 & cur_t not Done) do
24:                 monitored_running(cur_t)
25:                 if (Preemption) then
26:                     context_switching()
27:                 end if
28:             end while
29:             $apply\_recovery\_policy()$
30:         end if
31:     end if
32: end procedure
```

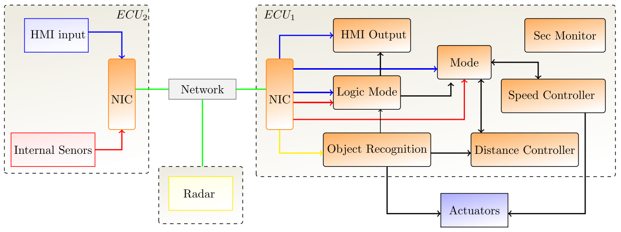

Figure 10: ACC software components based on [12]. The color of arrows within ECU_1 refer to the source of the information which has the same color and mapped to ECU_2.

Table 1: WCETs and Deadlines of software components of ACC system. Values for τ_2–τ_6 are based on [12].

Task	Task Name	T	WCRT
τ_0	Sec Monitor	150	4
τ_2	Speed Controller	20	5
τ_1	NIC	10,200	6
τ_3	Distance Controller	100	20
τ_4	HMI Output	100	2
τ_5	Mode Logic	100	1
τ_6	Object Recognition	100	30

Figure 10 shows the various software components which collaborate to achieve the ACC functionality besides the security monitoring component (Sec Monitor). All these software components are mapped to the same ECU (i. e., ECU_1), and require information delivered by other components within different ECU and sensors (e. g., Radar and internal sensors). The Radar sensor is used to monitor the road ahead and inform the Object Recognition task. As long as there is no object ahead, ACC maintains the speed value was set by the driver through HMI input. Otherwise, ACC reacts based on the speed of the detected object. If an ahead vehicle slowed down, the speed controller reduces the car's speed by releasing the accelerator and/or by activating the brake control system. If the road becomes clear again, ACC accelerates to reach the desired speed. In both cases, HMI output is used to inform the driver about the current speed and other statues. The vehicle's driver can, at any time, disable or enable the system, which is captured by the Logic mode component.

Table 1 illustrates the real-time properties of ACC tasks which are mapped on the ECU1. Within this system, there are two sporadic tasks (i. e., T_0 and T_1), and the rest have periodic activations. Note that in this system, the periodic task deadlines are defined to be equal to the periods ($D_i = T_i$). The tasks are ordered in the table from the highest priority to the least one. We are interested in comparing the defined MP with the calculated WCRT for each critical task and see how early we could alarm the system. Moreover, we are concerned about how often our approach could indicate a false off-nominal behavior of each monitored task (i. e., FPR value).

In Table 2, we show the computed bounds on each task, namely: the worst-case response time (WCRT), the typical worst-case response time which is considered as the monitoring point in our scheme (i. e., MP), the killing point (overall monitoring overhead is considered to be equal to minimum slack), and the false-positive ratio. By comparing the WCRTs with the MPs, we observe that we alarm the system (putting the task in the monitoring

Table 2: Red-Zone response time based prediction configuration and FPR.

Task	Mp	WCRT	ED	FPP
τ_0		4		
τ_2	5	9	9,33	2 %
τ_1		16		
τ_3	30	51	52,75	8 %
τ_4	32	53	56,17	8 %
τ_5	33	54	58,58	8 %
τ_6	73	94	100	8 %

mode) early enough – in this example, the MP is ranging between most 20 % to 40 % less than the WCRT- with a very low FPR (in average 8 %). A one has to be aware that the killing points KP are fair enough to let the task finishing its execution in case of false positive monitoring (normal execution). Simultaneously, to prevent any deadline miss for the lower priority tasks, which may cause a degradation in the system's QoS.

One other significant concern in the proposed scheme is its performance overhead. Our monitoring algorithm is integrated into the system scheduler; thus, it is applied for each hard and critical task. The added overhead of this modified scheduler comes from the use of the timers and the required time to save its value later during the context switching. Another overhead comes from the tracer; This overhead directly proportional to the type of the logging mechanism and the logging details. In our current proposal, we traced only the application's system calls.

8 Summary and discussion

The early detection of off-nominal behaviors in safety-critical systems is an important goal. It gives the system the ability to recover efficiently and prevents fault propagation. In this paper, we proposed prediction configurations for real-time tasks with temporal constraints to predict these tasks' off-nominal behavior. The defects of using the deadline and the worst-case boundaries as abnormality signs for real-time tasks have promoted us to define other limits. In particular, we used the TWCRT to define a bound that represents the least privileged temporal bound for each real-time task. Whenever a given task's response time exceeds these bounds, an alarm is triggered to notify the system about a potential off-nominal behavior that may be caused by the execution of malicious code. Such configuration allows the task to execute outside its designated profile under carefully controlled conditions. Moreover, we discussed the prediction scheme's design, which uses the prediction configuration to monitor the taskś behavior. Our approach depends on tight temporal bounds, making it applicable to be adopted by existing schedulable hard real-time systems. It is well known that determining precise prediction bounds is always a challenge. However, the accurate definition of the Red-Zone bounds will improve the efficacy of the prediction scheme.

It is essential to understand the decoupling between the Red-Zone principle and the method to define its boundaries (mainly the monitoring point), which depends on the studied system's charismatics. Defining the monitoring bound can be achieved via multiple methods. One of these methods is using the TWCA, which considers the system sporadically overloaded (i. e., system must contain sporadic tasks), which is the case with automotive systems. Consequently, this method will not be applicable for a system without sporadic tasks. However, any other method that gives us a monitoring bound that is less than the WCRT can be adopted. For example, someone can use the probabilistic analysis, which is applicable for a full periodic system, to define such a monitoring point. As future work, we could use different methods to define Red-Zone bounds and compare the efficacy of using the produced bounds with our current method.

Funding: This work is partially supported by the European Commission through the following H2020 projects: nIoVe under Grant Agreement No. 833742, THREAT-ARREST under Grant Agreement No. 786890, CONCORDIA under Grant Agreement No. 830927, and SmartShip under Grant Agreement No. 823916.

References

1. Faraz Ahmed, Haider Hameed, M. Zubair Shafiq, and Muddassar Farooq. Using Spatio-temporal Information in API Calls with Machine Learning Algorithms for Malware Detection. In *Proceedings of the 2nd ACM Workshop on Security and Artificial Intelligence*, pages 55–62. ACM, 2009.
2. James P. Anderson. Computer Security Technology Planning Study. Volume 2. Technical report, DTIC Document, 1972.
3. Neil C. Audsley, Alan Burns, Robert I. Davis, Ken W. Tindell, and Andy J. Wellings. Fixed Priority Pre-emptive Scheduling: An Historical Perspective. *Real-Time Systems*, 8(2-3):173–198, 1995.
4. Felice Balarin, Luciano Lavagno, Praveen Murthy, Alberto Sangiovanni-Vincentelli, et al. Scheduling for Embedded Real-time Systems. *IEEE Design & Test of Computers*, 15(1):71–82, 1998.
5. Dominique Bertrand, Sébastien Faucou, and Yvon Trinquet. An Analysis of the AUTOSAR OS Timing Protection Mechanism.

In *IEEE Conference on Emerging Technologies & Factory Automation, 2009 (ETFA 2009)*, pages 1–8. IEEE, 2009.
6. R. I. Davis, K. W. Tindell, and A. Burns. Scheduling Slack Time in Fixed Priority Pre-emptive Systems. In *Real-Time Systems Symposium, 1993, Proceedings*, pages 222–231, Dec. 1993.
7. Mohammad Hamad, Zain A. H. Hammadeh, Selma Saidi, Vassilis Prevelakis, and Rolf Ernst. Prediction of Abnormal Temporal Behavior in Real-time Systems. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 359–367, 2018.
8. Mohammad Hamad and Vassilis Prevelakis. Implementation and Performance Evaluation of Embedded IPsec in Microkernel OS. In *2015 World Symposium on Computer Networks and Information Security (WSCNIS)*, pages 1–7. IEEE, 2015.
9. Mohammad Hamad, Johannes Schlatow, Vassilis Prevelakis, and Rolf Ernst. A Communication Framework for Distributed Access Control in Microkernel-based Systems. In *12th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT16)*, 2016.
10. Mohammad Hamad, Marinos Tsantekidis, and Vassilis Prevelakis. Red-Zone: Towards an Intrusion Response Framework for Intra-vehicle System. In *Proceedings of the 5th International Conference on Vehicle Technology and Intelligent Transport Systems, VEHITS 2019*, Heraklion, Crete, Greece, May 3–5, 2019, pages 148–158. SciTePress, 2019.
11. Moncef Hamdaoui and Parameswaran Ramanathan. A Dynamic Priority Assignement Technique for Streams with (m, k)-Firm Deadlines. *IEEE Trans. Computers*, 44(12):1443–1451, 1995.
12. Hans Hansson, Mikael Åkerholm, Ivica Crnkovic, and Martin Torngren. SaveCCM-a Component Model for Safety-critical Real-time Systems. In *Proceedings. 30th Euromicro Conference, 2004*, pages 627–635. IEEE, 2004.
13. Grant A. Jacoby, Randy Marchany, and Nathaniel J. Davis. Battery-based Intrusion Detection a First Line of Defense. In *Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop, 2004*, pages 272–279. IEEE, 2004.
14. Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon Mccoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental Security Analysis of a Modern Automobile. In *Proceedings of IEEE Symposium on Security and Privacy*, 2010.
15. Krutartha Patel and Sri Parameswaran. SHIELD: a Software Hardware Design Methodology for Security and Reliability of MPSoCs. In *45th ACM/IEEE Design Automation Conference, 2008 (DAC 2008)*, pages 858–861. IEEE, 2008.
16. Martin Pohlack, Björn Döbel, and Adam Lackorzynski. Towards Runtime Monitoring in Real-time Systems.
17. Sophie Quinton, Matthias Hanke, and Rolf Ernst. Formal Analysis of Sporadic Overload in Real-time Systems. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE 2012)*, Dresden, Germany, March 12–16, 2012, pages 515–520, 2012.
18. Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. Real Time Scheduling Theory: A Historical Perspective. *Real-Time Systems*, 28(2-3):101–155, 2004.
19. Hyun Min Song, Ha Rang Kim, and Huy Kang Kim. Intrusion Detection System Based on the Analysis of Time Intervals of CAN Messages for In-vehicle Network. In *2016 International Conference on Information Networking (ICOIN)*, pages 63–68. IEEE, 2016.
20. John A. Stankovic and Krithi Ramamritham. What is Predictability for Real-time Systems?, 1990.
21. A. Taylor, N. Japkowicz, and S. Leblanc. Frequency-based Anomaly Detection for the Automotive CAN Bus. In *2015 World Congress on Industrial Control Systems Security (WCICSS)*, pages 45–49, Dec. 2015, doi:10.1109/WCICSS.2015.7420322.
22. Hideyuki Tokuda, Makoto Kotera, and Clifford Mercer. A Real-time Monitor for a Distributed Real-time Operating System. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 68–77, 1988.
23. Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Mihai Christodorescu, and Lui Sha. Learning Execution Contexts from System Call Distribution for Anomaly Detection in Smart Embedded System. In *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation*, pages 191–196. ACM, 2017.
24. Clinton Young, Habeeb Olufowobi, Gedare Bloom, and Joseph Zambreno. Automotive Intrusion Detection Based on Constant CAN Message Frequencies Across Vehicle Driving Modes. In *ACM Workshop on Automotive Cybersecurity (AutoSec ’19)*, 2019.
25. Christopher Zimmer, Balasubramany Bhat, Frank Mueller, and Sibin Mohan. Intrusion Detection for CPS Real-time Controllers. In *Cyber Physical Systems Approach to Smart Electric Power Grid*, pages 329–358. Springer, 2015.

Bionotes

Mohammad Hamad
Technical University of Munich, Department of Electrical and Computer Engineering, Munich, Germany
mohammad.hamad@tum.de

Dr.-Ing. Mohammad Hamad is a Postdoctoral Researcher in the Embedded Systems and Internet of Things group in the Faculty of Electrical Engineering and Information Technology at the Technical University of Munich (TUM). Mohammad received his Ph.D. from the Institute for Data Technology and Communication Networks at TU Braunschweig in 2020. Mohammad ’s research interests are in the area of Autonomous vehicle and IoT security.

Zain A. H. Hammadeh
Institute for Software Technology, German Aerospace Center (DLR), Cologne, Germany
zain.hajhammadeh@dlr.de

Dr.-Ing. Zain A. H. Hammadeh a research scientist at the German Aerospace Center (DLR). In 2019, he received his Ph.D. degree (Dr.-Ing.) in real-time systems from TU Braunschweig, Germany with Prof. Rolf Ernst. Since Feb. 2019 he joined the Institute for Software Technology as a research scientist.

Selma Saidi
Technical University of Dortmund, Department of Electrical Engineering and Information Technology, Dortmund, Germany
selma.saidi@tu-dortmund.de

Prof. Dr. Selma Saidi Selma Saidi is a Professor of Embedded Systems in TU Dortmund. Her research focus involve the design, implementation and validation of innovative intelligent embedded systems. Key aspects are the development of novel hardware and software design methods for embedded and autonomous systems where performance, predictability and self-adaptability play an important role. Domains of applications are avionics, autonomous driving and Internet of Things. Selma Saidi received in 2013 a Ph.D. degree in computer sciences from the University of Grenoble in France conducted together with STMicroelectronics. After her PhD, She joined the Technical University of Braunschweig as a Postdoctoral researcher.

Vassilis Prevelakis
Technical University of Braunschweig, Institute of Computer and Network Engineering, Braunschweig, Germany
prevelakis@ida.ing.tu-bs.de

Prof. Dr. Vassilis Prevelakis is the professor of embedded computer security at the Technical University, Braunschweig, in Germany. He holds B.Sc. degrees with Honours in Mathematics and Computer Science and M.Sc. in Computer Science from university of Kent at Canterbury, U.K. and a Ph.D. in Computer Science from university of Geneva, Switzerland. He has worked in various areas of security in Systems and Networks both in his current academic capacity and as a freelance consultant. Prevelakis current research involves issues related to vehicular automation security, secure processors, security aspects of software engineering, auto-configuration issues in secure VPNs, etc.