POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Intrusion Detection System (IDS) for Real-Time applications

Supervisors

Prof. Alessandro SAVINO

Prof. Stefano DI CARLO

Franco OBERTI

Candidate

Simone COSIMO

October 2023

Summary

As technology advances, so do the risks associated with it. The need for Intrusion Detection Systems (IDS) in modern embedded systems applications is becoming increasingly crucial. This threat is especially true in fields such as the automotive industry, where products are more connected than ever. These systems require a high level of resiliency to external attacks. That is where IDS comes in. By serving as the first line of defense, IDS helps to identify any alterations in software's nominal behavior that could lead to harm. It is essential in ensuring the safety and security of modern embedded systems.

This study proposes a new approach to developing an IDS that classifies running software based on Hardware Performance Counters (HPC). Initially designed for performance optimizations and safety, these special registers are now extensively utilized in cybersecurity-related applications. Previous research primarily focuses on side-channel attacks and identifying the optimal set of HPCs to detect specific malicious activities, such as software manipulation malware, with a particular emphasis on Real-Time Operating System (RTOS) system I/O tampering scenarios. Our approach to testing the solution's efficiency involves using two boards with CAN communication capabilities. The boards simulate data transfer between a sensor and its corresponding Electronic Control Unit (ECU). The receiving board acts as an ECU emulator and receives data from the sensor. It then performs data-dependent operations, such as the Moving Average Filter (MAF), a digital filter designed to reduce random noise. This filter is commonly used in the automotive industry to process the NOx concentration level provided by the NOx sensor via CAN protocol.
In addition, our system monitors Hardware Performance Counter (HPC) values. These values are utilized to train a one-class classifier that serves as the core of our IDS. This classifier distinguishes between legitimate and malicious data. The malicious data is obtained by reprogramming the board that acts as a NOX CAN sensor into an adversary NOX CAN Sensor emulator box, which is commonly available on the tuning online market and allows for the simulation of a real CAN tampering sensor on the CAN bus.

The obtained results underscore the viability of utilizing Performance Counters

as a consistent and novel option to construct an IDS that is closely tied to the activities of running tasks. This capability is particularly useful for systems with strict time requirements, such as those used in safety-critical applications. The tests also showed that the classification model could be trained with a limited dataset, making it more compact and suitable for resource-constrained environments like real-time embedded systems. Despite its compact size, the model maintained its precision in outlier detection, making it an ideal solution for present and future endeavors. Looking ahead, several potential future initiatives could build upon the work outlined in this report and contribute to a deeper understanding of real-time IDS classification. For example, researchers could experiment with different architectures and sets of HPCs to see how they impact the accuracy and speed of the classification process. These experiments could generate valuable insights to inform the development of more effective IDS systems.

Acknowledgements

In un lungo percorso, come quello universitario, costanza e determinazione sono le cose che di più contano. Sebbene all'inizio ne fossi un po' carente, ho ritrovato il fuoco di quelle qualità nelle persone che mi circondano. È una gioia poter condividere questo traguardo con i miei affetti più cari.

Ai miei genitori il ringraziamento più grande, che sarà sempre riduttivo rispetto al sostegno quotidiano e incondinizionato che mi hanno regalato.
Ai miei nonni, tutti, che ogni giorno, senza che me ne accorgessi, hanno lasciato in me insegnamenti che porterò per sempre nell'anima.
Ai miei zii e i miei cugini, da sempre per me sinonimo di festa e spensieratezza, stati d'animo vitali che mi hanno aiutato ad alleggerire il peso di un percorso così importante.
Alla mia fidanzata, generatrice costante della mia motivazione, che mi ha sempre spinto a superare i miei limiti, credendo in me più di quanto io stesso potessi mai fare. Per avermi liberato, per avermi spinto: Grazie.

In questo traguardo ci sono tutto Io, ma ci siete anche tutti Voi.
Vi ringrazio di cuore.

"Fa bene a restare all'ascolto, magari non sente ancora niente, ma la prego, non si arrenda, continui a impegnarsi. Perché io credo che qualsiasi siano i nostri sogni, prima o poi troveremo per forza ciò che cerchiamo, grazie alla voce che ci guida. La vita di un essere umano non è mai uniforme: ci sono momenti in cui il colore cambia di colpo."

Tokue, Le Ricette della Signora Tokue by Durian Sukegawa

Table of Contents

List of Figures

Acronyms

AI
: Artificial Intelligence

RTOS
: Real-Time Operating System

OS
: Operating System

HPC
: Hardware Performance Counters

IDS
: Intrusion Detection System

ECU
: Electronic Control Unit

ISR
: Interrupt Service Routine

Chapter 1

Introduction

In today's everyday life, embedded systems are crucial in almost every commonly used technology. From smartphones to cars, home appliances to smart cities, their presence is vital for correctly functioning the features that made life easier for the world population. Even if, most of the time, they're hidden behind a more extensive system, often used without the knowledge of being operated by electronic components with the job of performing a precise operation in a schema where a lot of other entities are present, their existence is essential not only because they usually make life easier but also because they make life safe. This thesis work puts its focus on one of the most common applications for embedded systems, like their usage in the automotive industry: with hundreds of Electronic Control Units (ECU) present in a commonly driven vehicle, their jobs space from signalling to the user a possible problem with the car, passing through the handling of the comfort feature that a car has to offer, arriving to the more critical safety mechanism they implement, the ones that save lives by firing airbags when a crash is detected or that take action on the breaks to avoid contact with obstacles and people. Functions related to safety-critical systems usually have a significant constraint related to timings: running tasks must follow strict deadlines that cannot be missed. Otherwise, the safety of people on board could be compromised. Real-Time Operating Systems (RTOS) are utilized to realize this precise implementation. This specific operating system is built with a foundation specifically designed for the definition and execution of tasks with precise time constraints. Their use is widely spread among different fields, also thanks to ease of development concerning bare-metal implementations, which are usually more complex and error-prone. While one can think ECUs are isolated systems that cannot be accessed from the outside, the reality tells a different story, with many articles regarding cars being stolen or tampered with present in the latest news. Criminals can steal cars by connecting custom-crafted devices to wires present in the front lights of the vehicle, simulating the key signal used to open the car. Still, they could also tamper ECUs

with the same method, potentially modifying important values necessary for brake or airbag control. These two wires are the main components of the CAN bus, a shared bus heavily used in the automotive industry for its cheap and effective implementation. It is capable of satisfying common design choices for in-vehicle information exchange. Considering this scenario, the thesis aims to develop a first line of defence to prevent malicious and unauthorized information from travelling across ECUs inside the vehicle, potentially modifying the legitimate computation. An Intrusion Detection System (IDS) is the selected choice: this software is capable of classifying specific data as legitimate or malicious according to their values, but the selection of these values (used for training the IDS) is of extreme importance for realizing a sound system that is not only able to detect ongoing attacks but, at the same time, does not classify as attacks some nominal computation (which can result in catastrophic events). Hardware Performance Counters (HPC) represent good training values since they are special registers present in modern CPU architectures that count architectural events happening in the system, painting an overall picture of the ongoing computation by training the IDS with values representing a nominal scenario it is then possible to distinguish malicious activity with good precision.
Chapter 2 describes the theoretical background needed to develop the project, explaining concepts about the CAN bus, RTOSs, HPC, and IDS.
Chapter 3 explains how the development environment was set up, together with the whole system description, the connection between the used development boards, and the software setup of the HPC.
. Chapter 4 shows the executed scenario with the main gained results. Finally, chapter 5 is focused on the future improvements that can be made to the project and the conclusions.

Chapter 2

Theoretical Background

2.1 The CAN bus

2.1.1 History of the protocol

At the beginning of the 1980s, Bosch engineers started to study existing serial bus protocols with the aim of using an implementation in new passenger cars. After looking at the pros and cons of every existing solution, the company put a lot of effort into engineering a new protocol that would best suit the use case in mind. This is the story behind the creation of the Controller Area Network bus, also known as CAN, that in 1986 was first introduced at the Society of Automotive Engineers Congress [1].
After only four years, two big players in the semiconductor industry produced the first actual implementations of CAN controllers: Intel and Philips. Intel produced a controller chip called 82526, with a FullCAN implementation, while Philips opted for a BasicCAN implementation with its chip called 82C200. Their main differences were the following:

- Intel produced a chip that eased the load on the CPU of attached electronic control units, while Philips's implementation relied more on the computational power of the microcontrollers.
- BasicCAN was capable of handling a more significant number of frames and also required less silicon to be produced.

If at the beginning of the CAN bus life, these differences were notable, also under the point of view of frame handling and filtering, today this distinction is not present anymore: a mixture of the two is often used, combining the pros of both ideas.
Today, it's straightforward to find a CAN network inside road vehicles, railroads,

3D printers, medical instruments, lighting control systems, and many other devices and systems that we use daily.

2.1.2 Technological overview

To better understand why the CAN protocol is so successful, let's analyze it from the technological point of view, starting from the physical layer. The CAN bus is formed by a pair of twisted wires terminated with two resistors. The standard calls one wire CANH and the other CANL.
A twisted pair of wires was chosen to avoid signal variations caused by external factors or noise. In fact, the CAN bus uses differential signals: this means that, when arriving at a node, the information is extracted by subtracting the signals. In this way, if any external noise modifies the signals, they will be both influenced in the same way, and their difference will not vary, providing a good form of noise resistance.

At the terminations of the twisted pair, two small resistors are positioned. In the ISO 11898-2 standard, they are defined as two 120 ohms resistors, and their job is to avoid any reflections on the twisted pairs to not corrupt any signal already present on the wires and return the bus to its recessive state. In fact, when talking about the possible signaling states, we identify:

- a dominant state that corresponds to a logical 0. In this scenario, the CANH and CANL voltages are driven towards 3.5V and 1.5V (these voltages depend on the standard in use. The mentioned ones are specified in the 11898-2 standard). If this logical value is present on the bus, all the other nodes trying to transmit a recessive value will fail.

- a recessive state that corresponds to a logical 1. In this scenario, the CANH and CANL voltages are stationary at around 2.5V. This state is called recessive because it is the default state on the bus when no node is transmitting, meaning that the transmission of a 0 will always prevail and change the bus status. Therefore, a logical 1 cannot be actively written on the bus itself: to signal it, a node should wait for two wires to return to their default voltages.

The CAN frame

On top of the described physical layer, the CAN messages, called frames, are transmitted from one node to another, carrying useful data. The protocol defines four types of frames:

- Data frame, used to exchange actual data between nodes

- Remote frame, used when a node requests transmission of a specific identifier

- Error frame, sent by a node when it detects an error
- Overload frame, used to create a delay between frames

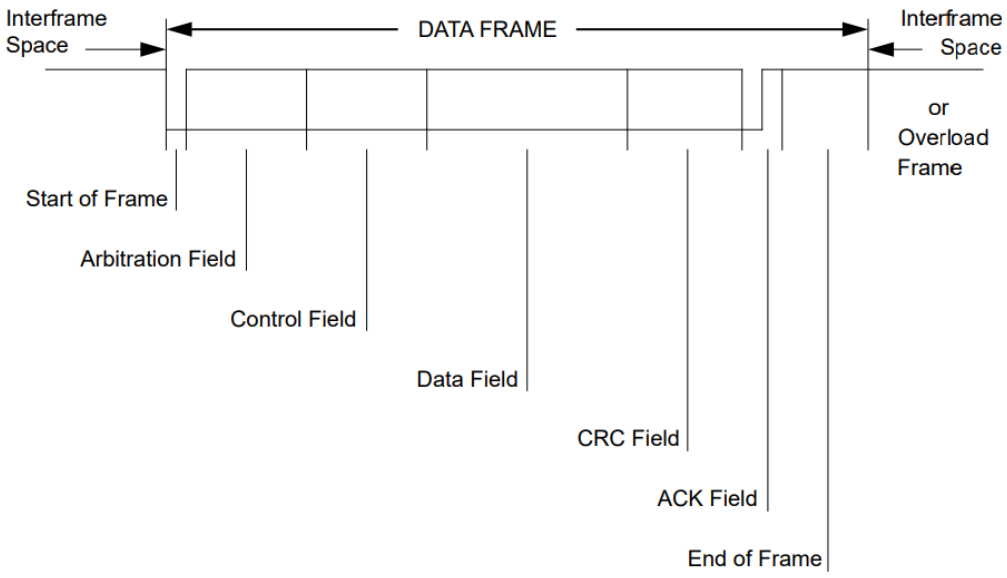


Figure 2.1: Data Frame composition [1]

Focusing on the Data frame, its structure is represented in image 2.1. We can distinguish different sections that represent values useful for the transmission of the message, such as:

- SOF (Start of frame): It is a single bit representing the start of the frame transmission.
- Arbitration field: It contains both the ID and RTR bits. These values are essential for the arbitration phase, used by the bus to decide who can transmit on the bus when collisions happen. According to the standard, the ID can have a length of 11 or 29 bits, representing the frame priority. The RTR (Remote transmission request) bit is always set to 0 for data frames and 1 for Remote frames.
- Control field: It defines the IDE (Identifier extension bit) that must be set to 0 if the transmitted frame uses an 11-bit identifier. The DLC (Data length code) is the other important information carried in this field, and, as the name suggests, it is a 4-bit information about the length in bytes of the carried data.
- Data field: it's where the actual data is contained. It can contain a maximum of 8 bytes of data.

- CRC (Cyclic redundancy check): 15-bit field for error detection against accidental data changes.
- ACK (Acknowledge slot): used to signal the reception of a valid CAN frame. If a node receives a valid frame, it responds with another frame with the ACK bit set to 0. On the other hand, if the transceiver doesn't detect a dominant level on the ACK bit, no node has correctly received the frame (this can lead to re-transmission).
- EOF (End of frame): This field marks the end of the data frame.

It is essential to notice that bit stuffing is used when creating and transmitting the Data frame. Every 5 bits with the same logical value is inserted a bit with opposite polarity. This is needed because the CAN bus uses a non-return-to-zero (NRZ) policy, thus not admitting any rest condition. All the frame is subject to bit stuffing, with the exceptions of the CRC, ACK, and EOF fields, which are of fixed size and convey crucial information such as integrity check over data and response signal that cannot be modified. The receiver will have the job of de-stuffing the frame and extrapolating the correct values.

Arbitration phase

Modern vehicles have Electronic Control Units (ECUs) that manage and control various automotive systems, such as engine performance optimization, fuel injection, emissions control, airbags, and stability control. These ECUs collect data from sensors throughout the vehicle, process it in real-time, and make immediate decisions to ensure the vehicle operates efficiently, safely, and in compliance with environmental regulations. The integration of ECUs has significantly improved vehicle performance, fuel efficiency, and overall safety, revolutionizing the automotive industry. This advantage is made possible by the practical data exchange among the ECUs facilitated by the Controller Area Network (CAN) bus. The CAN bus acts as the circulatory system of a vehicle's electronic architecture, allowing all ECUs to share information quickly and efficiently. Each ECU is strategically positioned to oversee specific subsystems, and they collaborate through the CAN bus to create a network that enables seamless coordination between vehicle components. For instance, one ECU monitors engine performance, another manages the transmission, and others control functions like climate and advanced driver assistance systems (ADAS). This continuous data exchange ensures that the vehicle operates smoothly and safely. While collisions on the CAN bus may happen frequently due to the hundreds of ECUs in a commercial vehicle, the network can handle them by deciding which conflicting frames must be transmitted.

The arbitration phase is performed to grant access to the bus to one of the many frames that try to access it. The arbitration performed by the CAN network

requires the nodes to be synchronized to sample the bits on the bus at the exact moment when nodes start sending the frame (as it can be seen in image 2.2), one of the first values they will share will be the ID in the Arbitration field: the CAN network uses this field to specify the priority of the frame, favoring low-value IDs. This results in an interruption of the transmission (with re-scheduling) for the nodes that are sending a '1' bit and detecting a synchronized '0' bit on the bus simultaneously, coming from another node in the network.

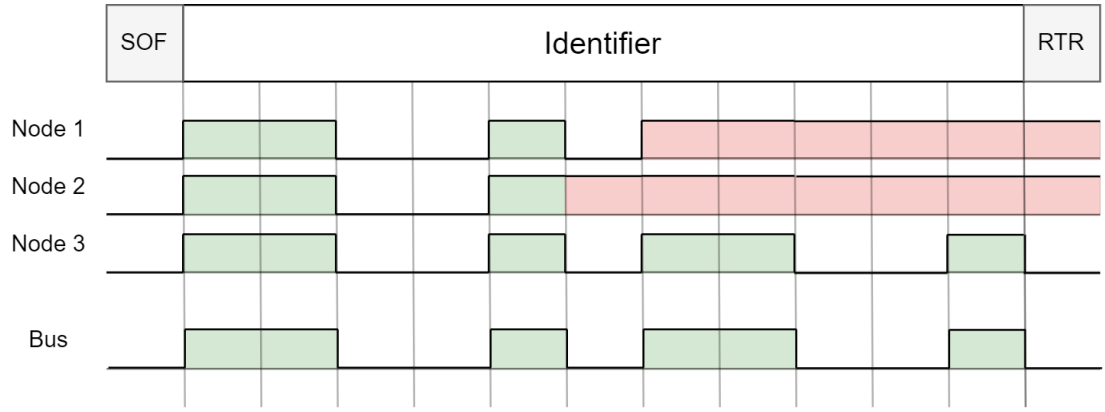


Figure 2.2: Example of arbitration phase, with interruption of transmission for frames with higher ID

2.2 Real-Time Operating Systems

A critical component in today's digital infrastructure is, without a doubt, a Real-Time Operating System (RTOS). An RTOS represents a critical component in most complex systems that surround us in everyday life in sectors like the automotive, medical, aerospace, and others. As its name suggests, an RTOS is involved in time-critical applications. It guarantees real-time responsiveness in scenarios where a slight deadline delay could result in catastrophic consequences.

The most known and used Operating Systems (OS) are the ones that are usually installed on regular computer electronics, like laptops. When smartphones: Windows, Linux, Android, MacOS, and iOS are just some examples of general-purpose OS. Their work is to create an abstraction level over the device hardware and ease the user's interaction with it: the general terminology indicates that the OS should be accommodating to satisfy every need that may not be known as a priority, focusing on prioritizing multitasking and resource-sharing. An RTOS works differently by having, as a critical characteristic, a deterministic nature: the OS is tailored over predefined and never-changing tasks that do not alter their

behavior, granting that time constraints can be reliably predicted. When defining a task, a process in the RTOS context representing the fundamental unit of work, the developer needs to specify the function that will act as the entry point and that will be executed, together with parameters like necessary initial data, stack size, and priority.

2.2.1 RTOS Scheduling

Scheduling is the operation of deciding which tasks that are marked as "ready to run" can access the CPU. The component that operates this decision is called the Scheduler, and it executes its job based on scheduling strategies. When trying to understand which of the pending tasks should access the computational resources, a Scheduler needs to consider different aspects that often reflect the system's necessities. Without a perfect solution to fit every application, different policies were designed to satisfy every need possible. Here just some of the most used scheduling policies are described:

- Priority-Based: This scheduling algorithm is one of the most utilized. Every task has an integer value representing its priority, meaning a higher priority grants access to the CPU more easily. This strategy can also be differentiated into fixed-priority and dynamic-priority: in the first policy, values indicating the priority of tasks cannot be changed at run time, which is possible in the second approach. The implementation of this algorithm is fairly simple, but it sees some problem that needs to be dealt with. The main one is the starvation of tasks with lower priorities when adopting a preemptive scheme. Preemption is the ability that permits higher tasks to interrupt a current lower-priority task being executed. As it can be deduced, in this situation, tasks of higher importance could be continuously run, leaving the low-priority ones in a permanent waiting state. To fix this problem different strategies can be adopted to fix this problem, like Aging in dynamic-priority contexts, which increments the priority of tasks that wait too long.

- Round-Robin: this policy assigns a fixed time slice, called Quantum, to each task in the system. Tasks are allowed to run only in their time slot, and once they terminate, they are immediately put in a ready queue. This strategy is highly "fair": no task will overcome others and for this reason, this algorithm is often used in systems where tasks have similar importance, avoiding monopolization of the CPU.

- Earliest Deadline First (EDF): contrary to the priority-based policy, in this algorithm, a deadline for each task has to be specified, together with its execution time. The Scheduler will then grant CPU access to the task with

the earliest deadline to respect. EDF is then considered a dynamic scheduling policy since the task is chosen based on the task's urgency.

2.2.2 The OSEK/VDX standard

As explained in chapter 3, the thesis work is developed thinking about an automotive scenario. In this work field, different companies from Germany and France came together to try to increase the portability and reusability of the application software. Thanks to this conjoined effort by significant players in the industry, in 1994, the OSEK/VDX standard was created. With time a lot of companies such as BMW, Renault, Siemens AG, Robert Bosch GmbH, FIAT, Volkswagen, and many others joined the project succeeding in creating interoperability of control units made by different distributors. The name of the standards comes from the union of the names of the two main consortia that took part in this journey: OSEK, a German acronym that means "Open Systems and their Corresponding Interfaces for Automotive Controllers", and VDX, "Vehicle Distributed eXecutive". Thanks to the design and development of a standard API, code re-utilization in creating Electronic Control Units (ECU) is now possible. With that, the definition of a software architecture is now the standard among the developers of real-time systems.

Tasks Management and Performance Classes

Let's now dive into the architecture of an OSEK/VDX-compliant RTOS, starting from the definition of the different classes of tasks:

- Basic Tasks: as the name suggests. This class of tasks is the most simple one. A Basic Task can be in three different states, where each one influences the choices of the scheduler:
  - Running state: in this state, the task is the possessor of the CPU, and the system is running it. There can be one and only one task with this state at a given time
  - Ready state: this is the state of the tasks waiting to be executed. The ready queue is the container in which all the tasks with the ready state are stored, and this queue can be handled in different ways by the operating system. OSEK RTOS, basing its scheduling on a fixed-priority algorithm, puts higher-priority tasks in the first positions of the queue. In contrast, the ones with the same priority level are handled by a simple First In, First Out (FIFO) policy.
  - Suspended state: finally, a task that does not want to be executed is associated with this particular state.

This class has the significant advantage of requiring fewer system resources to be handled. Still, it implies some downsides that must be considered: a primary task in the running state cannot voluntarily interrupt its execution. This ability is given to the scheduler that performs a content switch with a higher priority task or to some Interru. Still, it implies (ISR) is being fired. Therefore, it can be deduced that synchronization between tasks is impossible.

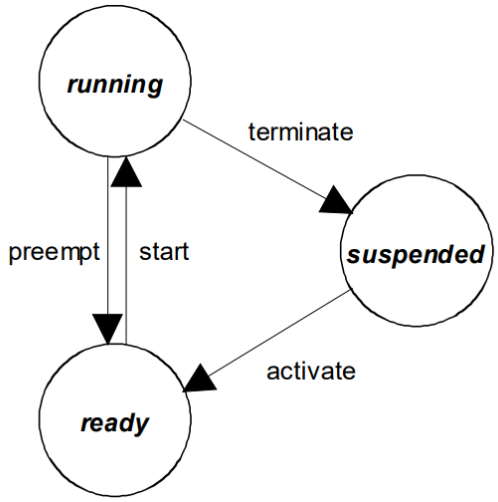


Figure 2.3: Basic Task state diagram [2]

- Extended Tasks: Everything said for the Basic Tasks is valid in this definition, too. The only big difference is that another state associated with the tasks is present in this class, called the waiting state. A task in the running state can interrupt its execution while waiting for a specific Event. This means that a path from the running state to the waiting state is present, and when the monitored Event rises, the task is moved to the ready state, where it can wait its turn to resume the execution. This synchronization mechanism increases OS complexity by using synchronization primitives such as Semaphores.

The introduction of conformance classes was deemed necessary to address the issue of varying system requirements among different application software and to facilitate the utilization of the OSEK operating system across a broad spectrum of hardware. The following classes, of which there are four, are designed to be upwardly compatible:

- BCC1: this profile only permits the use of Basic Tasks. This task cannot handle a request to activate a task already running. When defining priorities,

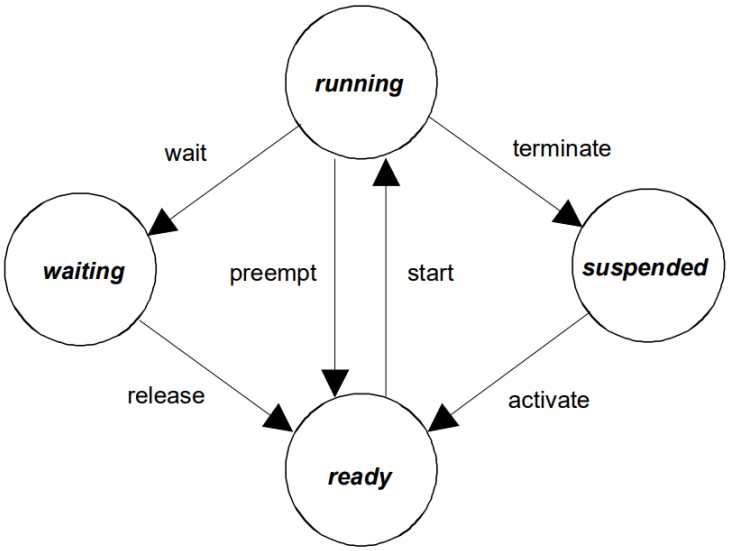


Figure 2.4: Extended Task state diagram [2]

every task has to have a different value, meaning that priority levels have a maximum population of one.

- BCC2: Similar to BCC1, with the possibility of satisfying one or more activation requests for an already active task. Moreover, priority levels are not unique, so it is possible to define multiple tasks with the same priority.
- ECC1: like BCC1, with support for Extended Tasks.
- ECC2: It's impossible to handle multiple task activation requests like BCC2 with support for Extended Tasks.

Scheduling

As hinted before, each task is assigned a priority, with zero representing the lowest priority and more significant numbers indicating higher priorities. These priorities are established statically and cannot be altered during runtime. During a context switch, the task with the highest priority is selected for execution. In the event of identical priorities, tasks are activated in a First In, First Out (FIFO) order. OSEK/VDX supports various scheduling policies, including non-preemptive scheduling, complete preemptive scheduling, groups of tasks, and mixed preemptive scheduling. Let's consider the first two policies:

- Full preemptive scheduling: In the context of complete preemptive scheduling, it is possible for a task with a higher priority to preempt the currently running

task at any given instruction. As a result, the latency period remains unaffected by the run time of lower-priority tasks. In a fully preemptive system, it is imperative to anticipate preemption by another task at all times. However, it is feasible to prevent the preemption of a critical region by explicitly blocking the scheduler.

- Non-preemptive scheduling: in this scenario, points of re-scheduling are fixed in correspondence with the natural termination of a task, with the terminated task directly calling another one, with an explicit call to the scheduler, and with a running extended task transitioning to the waiting state. Deductively, a task with a higher priority has to wait for the end of the lower-priority one being run.

OSEK Implementation Language

The objective of OSEK Implementation Language (OIL) is to provide a means of configuring an OSEK application within a specific CPU. This implies that for each CPU, there exists a single OIL description (a *.oil* file). All OSEK system objects are described utilizing the OIL format. The OIL description of the OSEK application is deemed to be comprised of a collection of system objects. A CPU serves as a container for these application system objects. OIL establishes standard types for system objects. A set of attributes and references characterizes each object. OIL explicitly defines all standard attributes for each standard system object. Each OSEK implementation can define additional specific attributes and references, but only attributes may be appended to the existing objects. The creation of new objects or any other modifications to the grammar are strictly prohibited. All non-standard attributes, also known as optional attributes, are considered fully implementation-specific and lack a standard interpretation. Each OSEK implementation can restrict the given set of values for object attributes, such as limiting the possible value range for priorities.

2.3 Hardware Performance Counters

2.3.1 The hardware way to optimize software

In the early days of computing, performance analysis was a relatively easy task: Computers were straightforward, and understanding the performances of running software was as straightforward as measuring execution times. With time, computers increasingly became more advanced, and processors and memory architectures evolved to achieve better performances. With the introduction of multi-core processors, pipelined architectures, caches, and other innovations, the ease of measuring

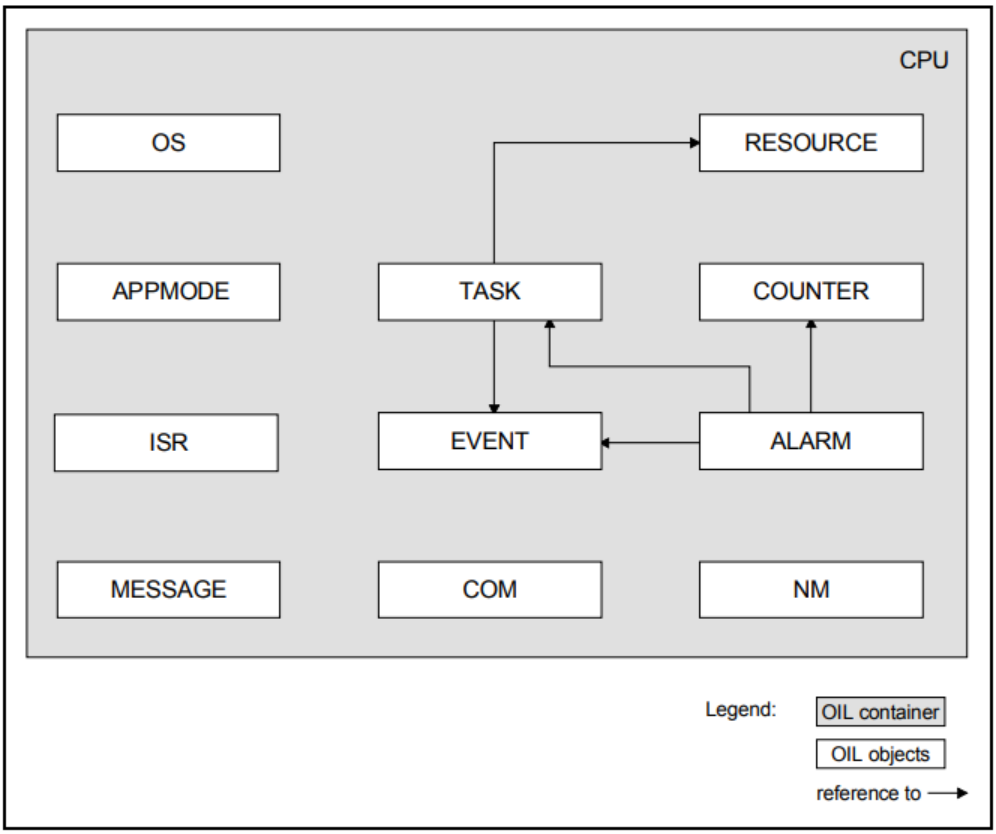


Figure 2.5: Standard OIL objects [3]

software performances rapidly increased, and the necessity for more precise and accurate methods of observing software behaviors for optimization purposes arose.

In the early 1980s, researchers started exploring hardware-based performance monitoring planning, creating special processor registers that specifically count architectural events like the number of executed instructions, cache hits and misses, branch predictions, and more. The architectural events recorded by these registers became higher since the innovation of out-of-order processors introduced complexity, and performing optimization on software became more challenging. Moreover, different vendors started to implement solutions with different hardware performance counters: the lack of standardization made the development of cross-platform software (and analysis tools) more challenging.

2.3.2 Computer Architectures Overview

Like in industrial assembly lines, where multiple vehicles are assembled in parallel, passing through different steps, processors execute instructions using a pipeline

[4]. Pipelining is a technique where instructions are executed in parallel, taking advantage of different stages that can be executed simultaneously without generating problems or errors. It is today one of the most used implementations when it realizes fast processors. Continuing the metaphor with an assembly line, the different steps an instruction has to go through are called pipe stages (or segments). A starting stage is linked to the following one in a chain-like fashion. A middle stage continues the work done by the previous one, passing it later to the successor. A final stage will do its job and terminate the pipeline, meaning that the instruction is now wholly executed. Modern processors, especially the ones with a Reduced Instruction Set Computer (RISC) architecture, usually implement a five-stage pipeline design:

- IF: Instruction Fetch stage, the instruction is retrieved from memory by reading the address of the Program Counter (PC), which is then incremented to point at the next instruction
- ID: Instruction Decode stage, where the previously loaded instruction is now decoded and eventual operands are obtained
- EX: Execution stage, where the ALU operates on the operands, whether they are both registers or immediate values
- MEM: Memory access stage, where any possible memory operation is performed (i.e. when dealing with load/store operations)
- WB: Write-back stage, the eventual result of the computation is stored in a specified memory location

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

Figure 2.6: The classic five-stage pipeline model [4]

The time necessary to move instruction from one stage to another is called the processor cycle, and the slowest pipe stage logically determines it. The described design was a revolution when confronted with non-pipeline processors, but sadly, the theoretical improvements are not always reachable since pipeline processors are prone to hazards.

Pipeline Hazards

Hazards are situations that prevent the next instruction from stepping into the next pipe stage, which is still occupied by the current instruction. In this scenario, following instructions that were fetched after the one that is stalling a stage are also stalled, leading to the impossibility of fetching a new one and improving performance. Hazards can be generated for various reasons, such as data dependencies (an instruction needs to operate on the result of a previous instruction) or PC modification caused by pipelining branch-related instruction. Understanding why and where they are generated is a matter of absolute interest to programmers who aim to have their software run as well as possible, and HPC is one of the instruments that developers can use to understand if pipeline stalls are the ones holding back the application from running at its maximum potential.

Caches

Another crucial architectural event that is the cause of non-optimized software is the slow access to data stored in memory. Accessing data saved in RAM is often considered a slow operation for a processor. For this reason, small but fast memories called caches were introduced inside the processor to mitigate excessive latency. They have the fantastic characteristic of being placed in very close proximity to the computational units of the processor, making access times small, but with the downside of not being able to store a large quantity of data. Consequently, different strategies of cache population can be implemented. The main idea behind caches is to store frequently used data or instructions to avoid accessing RAM and decrease the overall access time. When accessing the cache successfully retrieves the object of interest, it is called a Cache Hit. It is the best possible event, leading to a slight resource access latency. On the contrary, a Cache Miss is the opposite event where the population strategy did not succeed in loading the object into the cache itself (or maybe it is the first time accessing that specific piece of data or instruction), and the processor is forced to perform slow RAM access. Hits and Misses are another architectural event that HPC monitors since they can become valuable information in detecting cache-based attacks.

2.3.3 HPC Cybersecurity applications

Performance analysis is the primary application for hardware performance counters, as their name suggests. However, recent years have unveiled additional, noteworthy applications, including cybersecurity, which is the central focus of this thesis. Some of the alternative uses are as follows:

- Hardware performance counters (HPCs) can monitor events such as instructions and cache-related activities. This data enables software profiling based

on these values, thereby facilitating the detection of potential malicious executions. Malware often exhibits distinct execution patterns compared to legitimate software.

- Side-Channel Attack detection: HPCs are instrumental in detecting side-channel attacks, a class of attacks seeking to gather information about running software through observed side effects. Cache-based side-channel attacks, like "Meltdown"[5] [6] and "Spectre," can be identified by analyzing cache behavior for anomalies in access. ,
- Memory-based Attacks detection: HPCs play a supportive role in enhancing security features, such as stack canaries, used to detect buffer overflows or Return-Oriented Programming (ROP) attacks. Performance counters monitor memory activities, thereby strengthening these techniques.
- Forensics and Incident Response: Storing HPC values and post-incident analysis can aid in reconstructing attack sequences and gathering evidence, helping to understand how specific incidents occurred.
- Intrusion Detection and Prevention: The central focus of this thesis lies in utilizing HPCs for intrusion detection and prevention. Analyzing performance counter values for specific software or routines provides valuable insights into identifying potential malicious executions compared to a "normal" scenario defined during the software's design phase. Intrusion Detection Systems (IDS) can make proactive decisions to mitigate ongoing attacks, further emphasizing the importance of HPCs in this context.

  These diverse applications demonstrate that hardware performance counters have evolved beyond their initial purpose, playing a pivotal role in enhancing cybersecurity and the broader field of system analysis and security.

2.4 Intrusion Detection Systems

Monitoring events within a computer system or network and analyzing them for indications of intrusions, defined as efforts to compromise the confidentiality, integrity, availability, or circumvent the security mechanisms of a computer or network, is known as intrusion detection. Intrusions may be instigated by attackers who gain access to systems, authorized system users who attempt to obtain additional privileges beyond their authorization, or authorized users who misuse their granted privileges. Intrusion Detection Systems are software or hardware products that automate this monitoring and analysis process [7]. If, usually, IDS are developed to protect communication via the Internet by trying to analyze incoming data to spot possible intrusions, the embedded systems context sees a

different implementation of the technology since every single application can be different: the core concept is the one of extracting some form of metric from the running system or the exchanged data and develop a strategy that will be the base of the intrusion detection. In the automotive field, IDS was developed to monitor CAN bus traffic [8], while this thesis work approached the problem from the running software side, hypothesizing a system with different communication techniques with the outside world, leading to a more general approach under this point of view. As can be deduced from this description, IDS is highly tied to the application they need to monitor because the metrics and the algorithms to make decisions and categorize the events as intrusions may differ from system to system according to the implementation and the software design, and Embedded Systems are no exceptions.

While they serve as the first line of defence, they are usually considered passive systems, meaning that they do not take any action towards actively protecting the system they're monitoring, relegating this function to other specialized software. This definition often holds, but exceptions are made when the system is of particular importance, such as those considered in this work, often responsible for real-time computation with a direct connection with people's safety.

Chapter 3

Configuration of the Environment

In this chapter, the steps and considerations taken to develop the project are described in detail with the help of code snippets. Before diving deep into the implementation details, the adopted workflow needs to be explained to make the general picture clear and understandable.

After a thorough scouting job to choose a suitable hardware base to work on, the development starts by either emulating an RTOS by writing a basic bare-metal scheduler implementation (described in section 3.2) or running an actual RTOS (described in section 5.1.1): in case the second option is the preferred one, it is suggested first to identify the OS and then the architecture since it needs to be supported. The developed scenario is then composed of two running tasks with different timing constraints, which execution is eventually paused when interrupts related to the CAN bus are fired. The project sees the involvement of another device (an Arduino in this case) that sends via CAN bus some CAN frames with specific IDs and data (described in section 3.3) that the main board will then use to execute the functions (described in section 4.1 that hardware performance counters will monitor. The Arduino acts as a NOx sensor that communicates with an ECU (the Infineon TC2xx Evaluation Board) inside a vehicle and can emulate both legitimate and malicious entities.
The HPC values that are measured over the selected functions are divided into two classes:

- Legit values. HPC monitors the scenario of legit execution of the function, meaning that data coming from the CAN sender follows the constraints specified during the design phase (for example, they're contained in a particular range)

- Malicious values. HPC monitors the scenario of malicious execution of the function, meaning that data coming from the CAN sender does not follow the original constraints (for example, they're out of a particular range)

The first class of values will be then used to train the One-Class Classifier (described in section 4.2) that will generate a model capable of classifying new data as legit that comes from a nominal execution of the function or as malicious. This model will be the core of the Intrusion Detection Systems.

3.1 Architecture Selection and Developing Environment

The first step was choosing the architecture and the hardware to implement our solution. To make a wise decision, different considerations were made to avoid making a simple exercise demonstration: the objective was to emulate as much as possible a real production environment, with hardware, communication protocol, and software choices that reflect industry standards.

For this reason, the chosen architecture was TriCore by Infineon. TriCore is a 32-bit architecture with a RISC design and unique capabilities of digital signal processing, thanks to the inclusion of a Digital Signal Processor (DSP) in the same package as the MCU and the processor core itself: from here, the name TriCore. This particular architecture is one of the most used in the automotive world. Infineon, one of the big companies in the field, develops ECUs for combustion engines, electrical and hybrid vehicles, braking systems, airbags, connectivity, and others. Infineon offers a wide selection of development boards and starter kits with an equipped TriCore package and different on-chip peripherals. The Infineon TC299 Evaluation Board was used to develop the project because of its multi-core design, availability of documentation, and ease of development, thanks to an easy-to-use Integrated Development Environment and availability of working low-level drivers for the peripherals collected in the iLLD Infineon library. Nonetheless, the selected board supports both CAN protocol communications and performance monitoring thanks to hardware performance counters, key characteristics to develop a meaningful implementation of the IDS.

The board was programmed using Aurix Development Studio (ADS), freely available from Infineon's website. The IDE comes with an embedded version of the TASKER compiler that makes possible the cross-compilation needed to produce the correct binary file to be flashed on the board. Both the compilation, the flashing process, and the debug capabilities are performed by ADS and require no setup.

Since a crucial part of the scenario considered in the project is the exchange of data through a heavily used protocol in the automotive field like the CAN protocol, a board that will send information is needed, and an Arduino UNO does the

job when paired with a CAN transceiver like the MCP2515. The communication between the Arduino and the MCP2515 is performed via SPI.

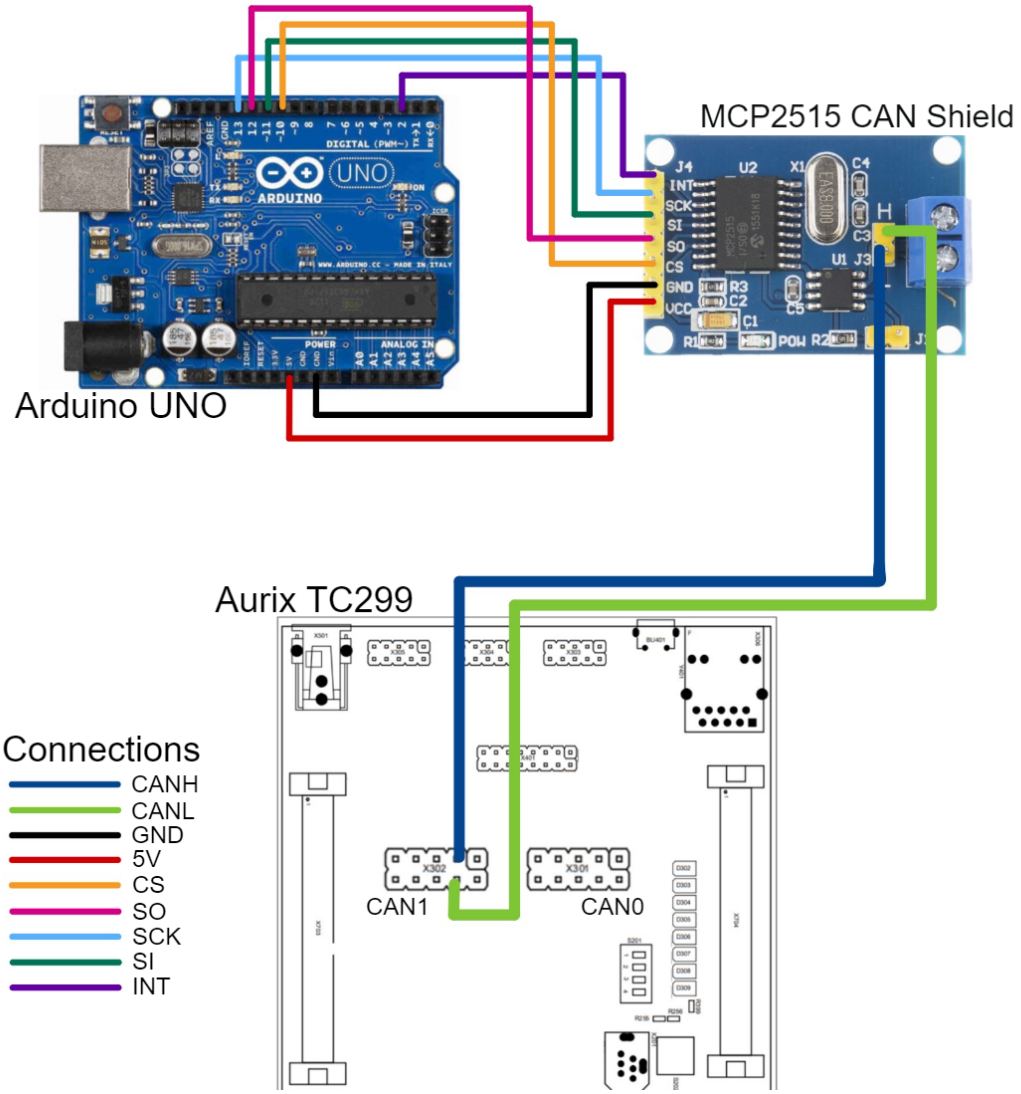


Figure 3.1: Connections between Arduino UNO, MCP2515 CAN shield and Aurix TC299 board

3.2 RTOS Emulation

In the implementation of the project, the decision to develop a simple bare-metal scheduler was made. In this way, the ease of development around the main objective

of the thesis, that is, measuring HPC values and classifying them, increased. Still, the effort to port the work on an RTOS like ERIKA Enterprise was made and described in 5.1.1.

To replicate an RTOS scheduler capable of precisely choosing the right tasks to execute to grant the timing constraints defined during the coding of the tasks themselves, a scenario where two dummy tasks are executed was defined.
To be precise, the tasks are represented by the following functions:

- A first task (called TASK25, for future references), executed every 25ms, of the approximate duration of 3ms
- A second task (called TASK100, for future references), executed every 100ms, of the approximate duration of 3ms

```
1 void TASK(void) {
2     uint32 i;
3     for(i = 0; i < 100000; i++) { }
4     for(i = 0; i < 100000; i++) { }
5 }
```

As seen in the above listing, the code of the tasks consists of two empty For loops. The specific number of iterations complies with the constraints of 3ms duration of execution: to check the correctness of timings, the HPC measuring clock cycles can be used, or a specific function could be coded taking advantage of the on-board timer and the clock frequency information.

These two tasks are periodically executed by the simple bare-metal scheduler listed below. This loop is the main execution of one of the three cores in the board and is constantly run since the stopping condition will always be false, the so-called infinite loop. The first instruction saves into the *ticksFor25ms* variable the amount of ticks corresponding to 25 milliseconds. Embedded systems usually require a periodic time source called the clock tick, and each tick is generated according to the clock frequency that often can be set by the software designer to fit the needs of the application. In this case, the necessity is to understand, with the clock configuration of the device, how many ticks correspond to 25 milliseconds. This information is critical for the scheduler to perform a correct *waitTime()* operation to follow the time constraints.
The procedure is the following:

- At the beginning of the loop, save the number of ticks the timer has counted up to this point in the execution. The call to the *IfxStm_get()* function provided by iLLD lets the developer specify as a parameter the timer to use (in this case, the one called *STM0*, there may be other ones) and returns the number of ticks, that are saved into the *t1* variable.

- TASK25, the task with the execution period of 25 milliseconds is executed.
- An integer variable is incremented: this will be the counter that symbolizes the number of times the loop has been run. Since TASK100 is executed once every 100 milliseconds, it will be run when the fourth execution of TASK25 has just finished.
- The if statement follows the reasoning explained in the previous point. It will be ignored in the first three iterations and executed during the fourth, running the TASK100 and resetting the counter variable.
- A new *IfxStm_get()* call is executed with the same timer as the first one. The return value is saved in the *t2* variable this time.
- At this point, the execution of the tasks has been performed. What is now necessary is to understand how much time (or how many ticks) passed between the two measurements, and to do this, a simple difference is performed and saved into the *tdelta* variable. This data is essential when calling the *waitTime()* function because the loop has to be restarted after 25 milliseconds from the start of the current iteration, meaning that the number of ticks to be specified as a parameter in the wait function will need to take into account the time passed executing one or both tasks. That being said, the difference between *ticksFor25ms* and *tdelta* is the one of interest.

```
1  Ifx_TickTime ticksFor25ms = IfxStm_getTicksFromMilliseconds(&
       MODULE_STM0, 25);
2  uint8 i = 0;
3  uint64 t1, t2, tdelta;
4  while(1) {
5      t1 = IfxStm_get(&MODULE_STM0);
6      TASK();
7      i++;
8
9      if(i == 4) {
10         TASK();
11         i = 0;
12     }
13     t2 = IfxStm_get(&MODULE_STM0);
14
15     // ticks passed since the beginning of the execution
16     tdelta = t2 - t1;
17
18     waitTime(ticksFor25ms - tdelta);
19 }
```

3.3 CAN Communication

Emulating an automotive is key for successfully implementing the project. CAN is the main bus and protocol that ECUs use to exchange information with each other, and in this work, it is used to send data from the Arduino to the TC299 board: sent information is used by the Infineon board to execute a function, explained in section 4.1. In this section, the configuration of both the sender and the receiver are described with code samples.

3.3.1 Arduino setup: The Sender

An Arduino UNO board is the entity that generates and sends the data to the TC299. The Arduino UNO itself does not support the CAN bus, so the MCP2515 CAN module was used: a cheap but effective extension board mounting the MCP2515 and pin headers to connect it with the Arduino itself via SPI and others for the CAN communication (CAM_L/CAM_H). The sender's job will be generating the data and sending them to the receiver: three CAN frames are populated with random data that changes after every successful send operation, with two out of three frames strictly necessary for executing the function at the TC299 side. Forged data will follow two kinds of profiles: legit data is the one that is sent when the wanted execution should reflect the nominal one. In contrast, malicious data is sent if an emulation of an ongoing attack needs to be simulated. Let's see the description of each part of the script.

CAN module setup

The first step is finding a way to program the board, and using the *arduino-mcp2515* library made that job easy, as seen by the code presented in this section. First, it is necessary to call the constructor function of the *MCP2515* class, specifying as a parameter the pin number of where the SPI CS is connected (10 in the example). In the *setup()* function, are specified and created the structs that contain the data that need to be sent: for each one of them, the assignment of the frame CAN ID (*can_id* field), the length of the data (*can_dlc* field), and the actual data to be sent expressed as an array of 8 bytes (*data* field). Since data is dynamic and based on random computation, in the *setup()* function, only the static information regarding the frames, such as the CAN ID and length, is defined. Besides that, the necessary settings for the serial communication are set by specifying the baud rate used to communicate with the host computer to display the data. Also, some functions for defining the CAN module bitrate and mode of operation are called. To quickly switch between legit data and malicious data, a global variable called *WINDOWATTACK* is created, with boolean values 0 or 1.

Message creation and delivery

The application sends three different CAN frames; here are listed the functions that create the frames by populating the necessary struct fields:

- *sendSignalArray()*Thison is in charge of generating 8 bytes of data for the CAN frame with ID *0x722*. The for loop iterates eight times, each cycle generating a new random value fitting in one byte (numerical value between 0 and 255). Some messages are sent on the serial monitor to signal a successful delivery. The necessity for this specific data composition is explained in section 4.1.

```
1     void sendSignalArray() {
2       for(int i = 0; i < 8; i++) {
3         uint8_signal_data.data[i] = random(256);
4         Serial.print("MovAvg: byte #");
5         Serial.print(i);
6         Serial.print(" is ");
7         Serial.println(uint8_signal_data.data[i]);
8       }
9       mcp2515.sendMessage(&uint8_signal_data);
10      Serial.println("MovAvg: sent 8 byte signal");
11    }
12
```

- *sendWindowSize()*: The chosen function described in section 4.1, to be executed, needs to receive a one-byte value. This function is selected for generating the data, filling a single byte of the *movAvg_window_size* struct with ID *0x700*. The usual *random()* function is called. Still, this time, the generation is conditional. It depends on the value of the global *WINDOWATTACK* variable: if its value is 0, a legit value between 1 and 8 (both included) is generated;e if its value is 1, a malicious value outside of the range is created and sent.

```
1     void sendWindowSize() {
2       movAvg_window_size.data[0] =
3         (WINDOWATTACK) ? random(9, 100) : (1 + random(8));
4       mcp2515.sendMessage(&movAvg_window_size);
5       Serial.print("MovAvg: sent Window Size of ");
6       Serial.println(movAvg_window_size.data[0]);
7     }
8
```

- *sendLedToggle()*: This dummy function is created to send a simple eight-byte constant signal besides its last value, which is randomly generated each time.

This frame is sent to the TC299 board to emulate a more general scenario, where the board executes the HPC monitored function and toggles LEDs, as explained in section 4.1.

```
1     void sendLedToggle() {
2       for(int i = 0; i < 7; i++) {
3         led_index.data[i] = 0xCA;
4       }
5       led_index.data[7] = random(8);
6       mcp2515.sendMessage(&led_index);
7       Serial.print("Led: toggling led #");
8       Serial.println(led_index.data[7]);
9     }
10
```

All of these functions are called in sequence in the *loop()* function of the Arduino. Still, they are separated by a *delay()* function that waits for the number of milliseconds passed as a parameter before continuing with the execution. In the written implementation, the *DELAY* global variable specifies the wait time, and it is set to 250ms.

3.3.2 TC299 setup: The Receiver

The Aurix TC299 board is the receiver of the Arduino's data and the executor of the function that the HPC will monitor. This section describes how the CAN interface of the board was set up using iLLD, a library provided by Infineon containing all the low-level drivers for managing the board and its communication interfaces, timers, peripherals, etc...

The intended scenario sees the TC299 acting as the ECU in a vehicle, receiving data from a NOx sensor via the CAN bus. All the CAN frames that arrive through the bus are handled with interrupt service routines: precisely, frames will arrive in a burst of three, and each one of the frames will be inserted in a buffer. When this buffer of size three is filled, the interrupt will be fired, and the function specified to handle it will be run. In this function, other than checking the correct arrival of the frames, the different data will be split and managed based on the arriving frame CAN ID, and after this first phase, it is used as input to the considered function monitored by HPC.

Interrupt and CAN Transceiver configuration

Let's look at the setup of the CAN module on the board and how the interrupts are specified and handled. This In the iLLD context, in a bare-metal environment, the interrupts are declared by the following lines of code:

```
#define ISR_PRIORITY_CAN_OVERFLOW    1
IFX_INTERRUPT(
    canIsrOverflowHandler,
    0,
    ISR_PRIORITY_CAN_OVEFLOW
);
```

The first instruction defines a constant value that represents the priority of the interrupt: this is necessary because in case two ISRs are fired concurrently, the system needs to know which one to execute, meaning that in this definition scheme, two ISRs cannot have the same priority. The *IFX_INTERRUPT()* function is the one that will specify the actual ISR, and its parameters are the following:

- The first one (*canISrOverflowHandler* in the code snippet) is the pointer to the function that will be executed when the interrupt is fired. This function must be specified separately and have a *void* return type.
- The second parameter is an integer value that defines on what interrupt vector table (IVT) the function in the first parameter has to be positioned. The Aurix TC299 board has three separate cores that can be used to execute instructions. This also means that each one of the cores possesses an IVT, making this parameter necessary to distinguish which one of the cores will be the one to handle this precise interrupt request when it rises.IVTs are simple tables where a specific interrupt number is bound to a pointer indicating the function to call. In the example code, *0* means that the IVT of core number zero will be used to handle the interrupt.
- The third and last parameter is the one that sets the priority of the interrupt itself.

Configuring the CAN module is a laborious process that sees the setting of different values in a specific data structure offered by iLLD. A struct called *AppMulticanType* with the following definition is used for the purpose:

```
typedef struct {
    IfxMultican_Can      can;
    /* CAN module handle to HW module SFR set */
    IfxMultican_Can_Config      canConfig;
    /* CAN module configuration structure */
    IfxMultican_Can_Node        canNode0;
    /* CAN node 0 handle data structure */
    IfxMultican_Can_Node        canNode1;
    /* CAN node 1 handle data structure */
```

```
    IfxMultican_Can_NodeConfig       canNodeConfig;
    /* CAN node configuration structure */
    IfxMultican_Can_MsgObj       canSrcMsgObj;
    /* CAN source standard message object */
    IfxMultican_Can_MsgObj       canDstMsgObj;
    /* CAN destination standard message object */
    IfxMultican_Can_MsgObjConfig     canMsgObjConfig;
    /* CAN message object configuration structure */
    IfxMultican_Message      txMsg;
    /* Transmitted CAN message structure */
    IfxMultican_Message      rxMsg[NUMBER_OF_CAN_MESSAGES];
    /* Received CAN messages array */
} AppMulticanType;
```

As can be seen, different CAN modules are present on the board, so different fields inside the structure can be populated to configure each node separately. This data structure also contains settings regarding the message object in both cases where the board acts as a sender and a receiver. Each configuration is defined as a specific data type, each different depending on the purpose. Most of these settings can have the default configuration in the proposed implementation. Still, the following fields need to be adapted to make the board follow its role in the designed scenario:

- *canConfig*: This field is mainly used to configure values to make the module aware of the interrupt to be fired. The first step is loading a default configuration struct, followed by the specification of the interrupt priority for the interrupt node pointer (should correspond to the priority defined in the *IFX_INTERRUPT()* call). Moreover, the service provider that should handle the ISR is specified, and it is set to 0 since it's the core number that will compute the interrupt routurned offnally, the CAN module is initialized thanks to the *IfxMultican_Can_initModule()* function, using the previously defined configurations.

```
    // g_multican is a struct of type AppMulticanType
    IfxMultican_Can_initModuleConfig(
                &g_multican.canConfig,
                &MODULE_CAN
    );
    g_multican.canConfig.nodePointer[OVERFLOW_INTERRUPT_SRC_ID]
                .priority = ISR_PRIORITY_CAN_OVERFLOW;
    g_multican.canConfig.nodePointer[OVERFLOW_INTERRUPT_SRC_ID]
                .typeOfService = 0;
    IfxMultican_Can_initModule(
                &g_multican.can,
                &g_multican.canConfig
    );
```

```
14
```

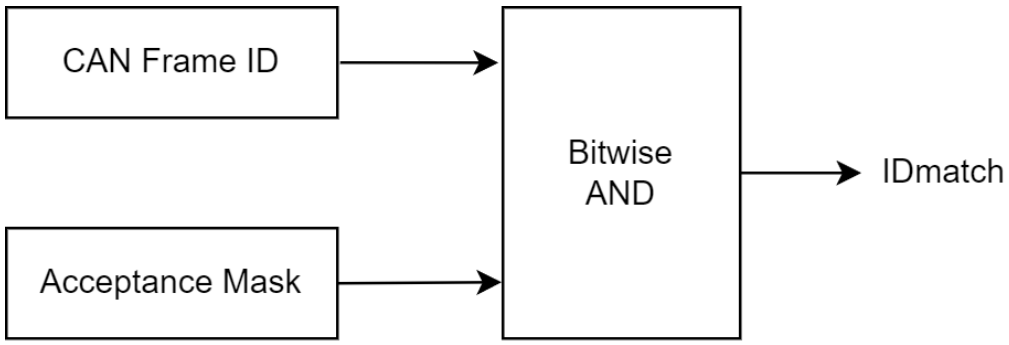
- *canNodeConfig*: This step is crucial since it defines what physical CAN node present on the board is used. It is important to remember that the physical layout of the board could change even in the same board typology: Aurix development kits can be modified by mounting/removing electronic components such as resistors and capacitors, and this could lead to a change of the PINs position or the detach of the module in its entirety. Like with the previous point, a function generating a default configuration data structure is called, and its fields are set, in this case, by specifying the physical node ID to use (in the example, node 1) and the PINs, which iLLD specify in a stand-alone file labelling them with both pin number (the schema is *P<port_number>_<pin_number>*) and their role (*TX* or *RX*). Finally, finalize the configuration by applying them to the right *g_multican* field. Another important boolean field that must be set to the *FALSE* value is specifying if the CAN module is working in *loopback* mode. In this mode of operation, the CAN modules on the board will communicate, emulating the connection, and it is usually enabled for testing purposes. Since the Arduino (an external entity) will be the one generating the data, the loopback has to be turned off to receive the wanted data frames.
  correctly

```
1     IfxMultican_Can_Node_initConfig(
2                 &g_multican.canNodeConfig,
3                 &g_multican.can
4     );
5     g_multican.canNodeConfig.loopBackMode = FALSE;
6     g_multican.canNodeConfig.nodeId = IfxMultican_NodeId_1;
7     g_multican.canNodeConfig.txPin = &IfxMultican_TXD1_P14_0_OUT;
8     g_multican.canNodeConfig.rxPin = &IfxMultican_RXD1B_P14_1_IN;
9     IfxMultican_Can_Node_init(
10                &g_multican.canNode1,
11                &g_multican.canNodeConfig
12    );
13
```

- *canMsgObjConfig*: The following operations aim to define the queue data structure where the messages will be stored and the filters for the incoming CAN data frames. Expressly, some integer values are specified as the base FIFO object ID and SLAVE object ID: they are needed to define the queue for storing the messages, with the base FIFO being the one receiving the frames first, while the SLAVE object is the memory location that will store messages

that arrives when the base FIFO is complete. Another crucial setting is the one where the acceptance mask value is defined. This value is necessary for the CAN transceiver to accept or deny incoming CAN frames based on their IDs specified in the Arbitration field, as specified in section 2.1. Image 3.2 shows how the filtering is performed by simply performing a bitwise AND operation on the CAN ID specified in the incoming frame and the acceptance mask value manually configured in the initialization code. If the result of the operation is 0, the frame is accepted and saved into the FIFO queue, ready to be utilized. Otherwise, it will be discarded.

```
1     IfxMultican_Can_MsgObj_initConfig(
2                 &g_multican.canMsgObjConfig,
3                 &g_multican.canNode1
4     );
5     g_multican.canMsgObjConfig.msgObjId = RX_FIFO_BASE_OBJECT_ID;
6     g_multican.canMsgObjConfig.msgObjCount = RX_FIFO_SIZE;
7     g_multican.canMsgObjConfig.frame = IfxMultican_Frame_receive;
8     g_multican.canMsgObjConfig.firstSlaveObjId =
9                 SLAVE_MESSAGE_OBJECT_ID;
10    g_multican.canMsgObjConfig.acceptanceMask = 0x70000000UL;
11    IfxMultican_Can_MsgObj_init(
12                &g_multican.canDstMsgObj,
13                &g_multican.canMsgObjConfig
14    );
15
```



IDmatch = 0: ID of the received frame fits to message object
IDmatch > 0: ID of the received frame does not fit to message object

Figure 3.2: CAN module acceptance mask filtering

With all the previous considerations and settings, the CAN module is ready to receive and execute work on frames. Still, enabling and defining the specific kind of interrupt that must be generated is crucial. In the code snippet below, after getting the pointer to the base queue that will store frames, some instructions enable the interrupt when the queue is overflowed and also specify the ID of the salve object, explicitly routing new CAN frames to that destination when the base queue is already filled.

```
Ifx_CAN_MO *hwObj;
hwObj = IfxMultican_MsgObj_getPointer(
            g_multican.can.man,
            RX_FIFO_BASE_OBJECT_ID
);
IfxMultican_MsgObj_setOverflowInterrupt(hwObj, TRUE);
IfxMultican_MsgObj_setTransmitInterruptNodePointer(
            hwObj,
            OVERFLOW_INTERRUPT_SRC_ID
);
IfxMultican_MsgObj_setSelectObjectPointer(
            hwObj,
            SLAVE_MESSAGE_OBJECT_ID
);
```

3.3.3 Interrupt Service Routine

Now that the interrupt is set up in terms of who rises it and how, it is time to describe the function that will be called when the interrupt is received, the one that represents the first parameter of the *IFX_INTERRUPT()* function described in section 3.3.2, the entire function is listed in appendix A.
The first instruction is setting up and starting the performance counters because here is where the designated function to be monitored will be called: the explanation on this aspect is described in section 3.4. The focus here is on what the ISR performs on the arrived CAN frames, and its job is straightforward: the function cycles on the queue of received messages, extracting the information about the status of the message with the following code.

```
readStatus = IfxMultican_Can_MsgObj_readMessage(
            &g_multican.canDstMsgObj,
            &g_multican.rxMsg[currentCanMessage]
);
```

The type of the *readStatus* variable is an enumeration representing the different states that a received message can have, and based on its value, some checks are

performed to avoid working with incorrect or incomplete data. In the proposed implementation, the following checks are performed:

- The first if block checks if the status differs from the *IfxMultican_Status_newData* enum value. If it is, no new data arrived in the queue, so no action should be performed.

- The second if block checks if the status equals the *IfxMultican_Status_newDataButOneLost* enum value. If it is, it means there is new data, but one of the frames did not arrive correctly at the interface.

- The third if block checks if the data transmission was successful. Here, the received packets are analyzed and processed to prepare them for the chosen computation that HPC will then analyze. As shown in image 3.3, the Arduino emulating the NOx sensor will send a burst of three CAN frames with different IDs and, if considering the Moving Average example (explained in section 4.1.2) frames will contain data representing the 8-byte signal, the 1-byte window size value and a dummy frame containing a random value to toggle some LEDs on the board.

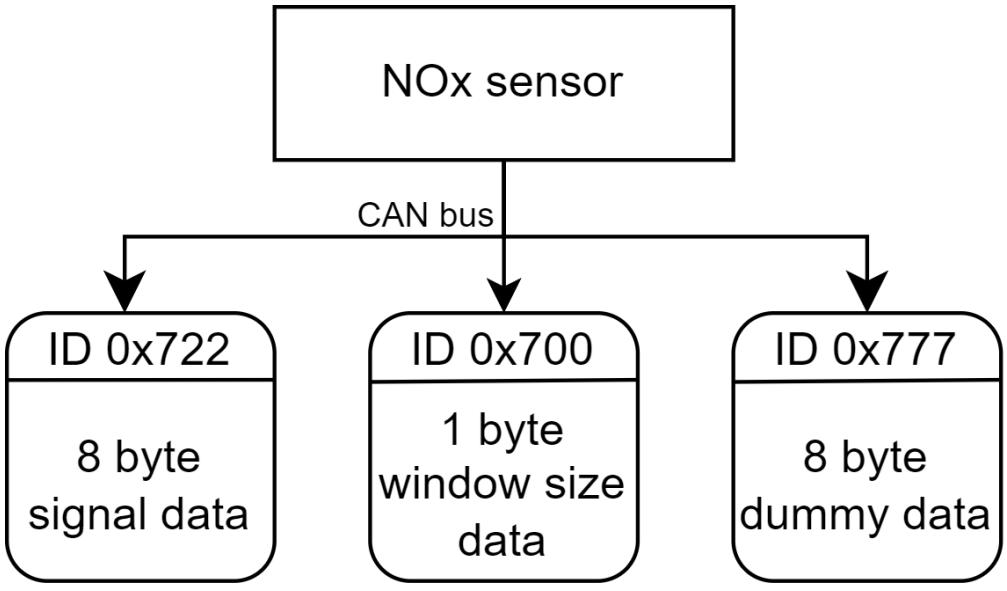


Figure 3.3: CAN frames sent by the NOx sensor in Moving Average Filter example

3.4 Hardware Performance Counters Configurations

The developed IDS is a Classifier based on data from HPC measuring a specific function of interest decided during the project's design phase. HPC are the key to training a model good enough to make a precise detection. In the TriCore Infineon architecture used in the thesis, a set of five hardware performance counters is present with the following setup:

- A register counting clock cycles
- A register counting executed instructions
- Three registers that are user-configurable and can count different architectural events by simply setting the respective control register to specific binary values

CFG	M1CNT	M2CNT	M3CNT
000	IP_DISPATCH_STALL	LS_DISPATCH_STALL	LP_DISPATCH_STALL
001	PCACHE_HIT	PCACHE_MISS	MULTI_ISSUE
010	DCACHE_HIT	DCACHE_MISS_CLEAN	DCACHE_MISS_DIRTY
011	TOTAL_BRANCH	PMEM_STALL	DMEM_STALL

Figure 3.4: HPC configurations on Aurix TC27x board family [9]

As can be seen from image 3.4, the TriCore architecture implements hardware performance counters focused on recording stalls of specific pipeline stages, cache hits and misses (both for instruction cache and data cache), and others covering branch prediction information. The flexibility of deciding which of the events to record comes with the duty of explicitly setting a specific register with a specific value to make the processor aware of the wanted setup. iLLD offers easy access to the *CPU_CCTRL* register, together with two functions called *___mfcr()* and *___mtcr()* that have to be used to get and set the current register value, respectively.

```
1 void enablePerfCountersWithMultiCount(
2                 uint32 M1CNT,
3                 uint32 M2CNT,
4                 uint32 M3CNT)
5 {
6     Ifx_CPU_CCTRL cctrl;
7     cctrl.U    = ___mfcr(CPU_CCTRL);
8     cctrl.B.M1 = M1CNT;
```

```
9       cctrl.B.M2 = M2CNT;
10      cctrl.B.M3 = M3CNT;
11      __mtcr(CPU_CCTRL, cctrl.U);
12      IfxCpu_resetAndStartCounters(IfxCpu_CounterMode_normal);
13  }
```

The above code listing defines a wrapper function that will reset and start the performance counters only after setting the wanted configuration on the *CPU_CCTRL* register, passed as parameters. As hinted before, the assignment cannot be done via the classic = operator since the considered register is related to a specific core, and it is protected in order to avoid accidental modifications. The Move From Core Register (MFCR) and Move To Core Register (MTCR) functions will handle the modification granting secure access: while getting the value thanks to MFCR is an operation runnable in every CPU state, setting the register via MTCR can only be done when in Supervisor mode. Moreover, the *__mtcr()* wrapper function contains an inline assembly definition for the *mtcr* instruction itself, plus an *isync* instruction that ensures that all other instructions after it will see the modifications made to the *CCTRL* register.
Finally, the *IfxCpu_resetAndStartCounters()* function is the one that will reset and start the performance counters with the mode specified as a parameter, which can be of two types:

- Normal mode: The counters increment on their respective triggers
- Task mode: Allow an additional gating from the debug unit which can filter data based on specific criteria

From this point on, HPC are active and with a debugging tool, their addresses can be read and studied while they continuously change. What about stopping them? For this purpose, iLLD provides the *IfxCpu_stopCounters()* function, which accepts no parameter and returns the values of the counters in a struct of type *IfxCpu_Perf* with the following definition:

```
1   typedef struct
2   {
3       uint32  counter;
4       boolean overlfow;
5   } IfxCpu_Counter;
6
7   typedef struct
8   {
9       IfxCpu_Counter instruction;
10      IfxCpu_Counter clock;
11      IfxCpu_Counter counter1;
```

```
12      IfxCpu_Counter counter2
13      IfxCpu_Counter counter3;
14  } IfxCpu_Perf;
```

The last step is finding a way to display and retrieve these values while running the system: Aurix Development Studio has some really useful debugging functions for this purpose. The easiest way to print the captured data is by taking advantage of the Simulated FileSystem (which produces the below output): with this feature, it is possible to use the *printf()* function contained in *stdio.h* that wouldn't be normally accessible. This solution implies the availability of the development environment and the debugging tools: to make the system production-ready, a more stand-alone strategy needs to be adopted, as hinted in section 5.1.

```
1  [11] * CCNT: 4258(0), ICNT: 1404(0), M1CNT: 1125(0), M2CNT: 599(0),
       M3CNT: 278(0)
2  [14] * CCNT: 4240(0), ICNT: 1386(0), M1CNT: 1116(0), M2CNT: 593(0),
       M3CNT: 281(0)
3  [17] * CCNT: 4268(0), ICNT: 1408(0), M1CNT: 1122(0), M2CNT: 599(0),
       M3CNT: 282(0)
4  [20] * CCNT: 4213(0), ICNT: 1348(0), M1CNT: 1104(0), M2CNT: 583(0),
       M3CNT: 278(0)
5  [23] * CCNT: 4259(0), ICNT: 1404(0), M1CNT: 1124(0), M2CNT: 600(0),
       M3CNT: 279(0)
6  [26] * CCNT: 4213(0), ICNT: 1348(0), M1CNT: 1104(0), M2CNT: 583(0),
       M3CNT: 278(0)
7  [29] * CCNT: 4259(0), ICNT: 1404(0), M1CNT: 1124(0), M2CNT: 600(0),
       M3CNT: 279(0)
8  [32] * CCNT: 4239(0), ICNT: 1386(0), M1CNT: 1115(0), M2CNT: 593(0),
       M3CNT: 281(0)
9  [35] * CCNT: 4213(0), ICNT: 1348(0), M1CNT: 1104(0), M2CNT: 583(0),
       M3CNT: 278(0)
10 [38] * CCNT: 4259(0), ICNT: 1404(0), M1CNT: 1124(0), M2CNT: 600(0),
       M3CNT: 279(0)
11 [41] * CCNT: 4077(0), ICNT: 1278(0), M1CNT: 1097(0), M2CNT: 569(0),
       M3CNT: 257(0)
12 [44] * CCNT: 4268(0), ICNT: 1408(0), M1CNT: 1123(0), M2CNT: 599(0),
       M3CNT: 282(0)
13 ...
```

Chapter 4

Running the experiment: HPC Capture and Model Creation

With an environment equipped with an RTOS (or an emulation of it) and capable of receiving and elaborating data from a sensor, executing the desired function while monitoring it with HPC and the ability to extract them, the last milestone is executing the system and using the HPC measurements to build the classifier, the core of the IDS. This section describes this last step, explaining the decisions made over the kind of functions run by the ECU and, more importantly, the solution adopted for the classifier used for the outlier detection.

4.1 HPC Collection over Sample Functions

HPC can depict the currently running software by counting specific architectural events during the whole execution. To make them good indicators to use for the development of a classifier, it is necessary to frame them over the specific function since they cannot distinguish any data or external signal unless this data or signal affects the function in a way they can detect, for example by slowing or speeding up the execution, by generating pipeline stalls or changes in the cache use. With this in mind, the two functions selected for the project possess these characteristics. Still, they are, at the same time, slightly different from each other in how they are launched and behave when input data changes, like in an attack scenario.

The first routine calculates the n-th element of the Fibonacci sequence, a standard computation used in computer data structures and sorting algorithms, financial engineering, audio compression and watermarking [10], and architectural

engineering. The second routine is an implementation of the Moving Average Filter (MAF). This digital filter is often utilized to remove random noise present on a signal by averaging the values of the signal itself over a sliding window of a fixed size. While Fibonacci is a function that is always run even if the received data comes from a malicious actor, MAF is an "on/off" routine executed only if incoming data is coherent with a specific format and value range.

Since three out of five performance counters are configurable, the decision was to set these three registers to the following values (can be seen in appendix A) as a result of monitoring the produced values and their change when an attack was simulated:

- *IP_DISPATCH_STALL*: enabled by setting the correspondent part of the control register to the hex 0 value (default one). This counter is incremented on every cycle in which the Integer dispatch unit is stalled for whatever reason.
- *LS_DISPATCH_STALL*: enabled by setting the correspondent part of the control register to the hex 0 value (default one). This counter is incremented on every cycle in which the Load-Store dispatch unit is stalled for whatever reason.
- *DMEM_STALL*: enabled by setting the correspondent part of the control register to the hex 0x11 value. This counter is incremented whenever the Load-Store unit requests a data operation, and the data memory is stalled for whatever reason.

4.1.1 Fibonacci Sequence

```
uint64 a =  0, b = 1, c;
for(int i = 1; i < n; i++) {
    c = a+b; a = b; b = c;
}
printf("%d of fibonacci is %llu * ", n, a);
```

The Fibonacci sequence is a mathematical series in which each integer equals the sum of its two predecessors. The first numbers are usually fixed to 0, 1, and 1, even though variations, where the initial 0 is omitted, exist (but do not change the general behaviour). In the above code snippet, variable n is the one specifying what value of the sequence to calculate, and variable a is the one that will store the final result. In contrast, b and c are temporary memory locations used to swap and perform the operations needed to progress the mathematical sequence. It is easy to understand that the value of n will influence the execution: by only considering the HPC related to the number of issued instructions, it is possible to say that a

small n value will lead to a small HPC value, while a big n value will make the for cycle execute more times, increasing the HPC value as well.

$$F_0 = 0, F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...$$

In this case, when designing the function, one should be aware of the possible n values that fit the actual system: this choice permits the monitoring of the routine. It makes it possible to detect an attack when an incoming n is higher than a maximum threshold or lower than a minimum one. Some considerations could also be made about the ability to stop a possible attack when a function of this kind is utilized: since the calculations would be started with any n values (if no explicit checks are present in the code) the ability to promptly end the calculation after the detection of the attack could be complex because it would imply the ability to end the function externally. This ability should belong to the RTOS.

4.1.2 Moving Average Filter

The moving average is widely regarded as the most prevalent filter in digital signal processing (DSP) due to its simplicity and ease of use. Despite its straightforward nature, the moving average filter effectively reduces random noise while preserving a sharp step response [11]. As a result, it is considered the foremost filter for signals encoded in the time domain. If, when it comes to signals encoded in the frequency domain, the moving average filter is not as effective, variants such as the Gaussian, Blackman, and multiple-pass moving average, offer slightly improved performance in the frequency domain, albeit at the cost of increased computational time.

```
static void movingAverage(
            uint8 data[],
            uint8 filteredData[],
            uint8 dataSize,
            uint8 windowSize
) {
    for (uint8 i = windowSize - 1; i < dataSize; ++i) {
        uint16 sum = 0;
        for (uint8 j = 0; j < windowSize; ++j) {
            sum += data[i - j];
        }
        filteredData[i] = sum / windowSize;
    }
```

```
14 }
```

In the thesis project, the scenario depicts the NOx sensor sending an 8-byte signal via the CAN bus, meaning that a window size value from one to eight is expected. This will be the decisive factor differentiating a legitimate value from a malicious one, and it is what is simulated by the Arduino board. Unlike Fibonacci, the MAF is a function that, if input values are not coherent with the designed one, will not be executed thanks to the check of the exterior for loop. This is a wanted behaviour by the developer since it completely avoids the computation skipping a possible attack. Still, it could be interesting to monitor it anyway to understand if the attacker is present in the system and if it is trying to undermine the system or to implement a double line of defence in case good coding practices were not followed.

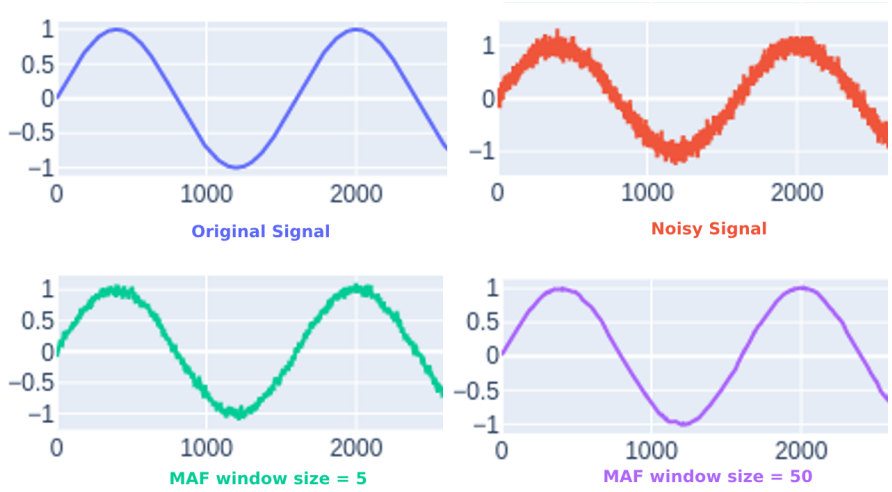


Figure 4.1: Moving Average Filter with different window sizes

4.2 One-Class Classifier and Model Creation

Deep Convolutional Neural Networks (DCNNs) have improved remarkably in object detection and recognition problems in the past decade. The availability of large multi-class annotated datasets has enabled deep networks to learn discriminative features that a classifier can utilize for recognition. The extreme case of recognition, namely One-Class Classification (OCC), the chosen one for the project implementation, is an implementation where only data from a single class (labelled "positive") is present during training [12]. During inference, the classifier encounters data from both the positive class and outside the positive class (also known as the "negative" class). The objective of OCC is to determine whether a query object belongs to

the class observed during training. In the absence of multiple-class data, both learning features and defining classification boundaries become more challenging in OCC compared to multi-class classification. Nonetheless, OCC has applications in abnormal event detection scenarios, like the one considered in this work. The acquisition of features that facilitate classification is an extensively studied issue in the realm of multi-class classification.

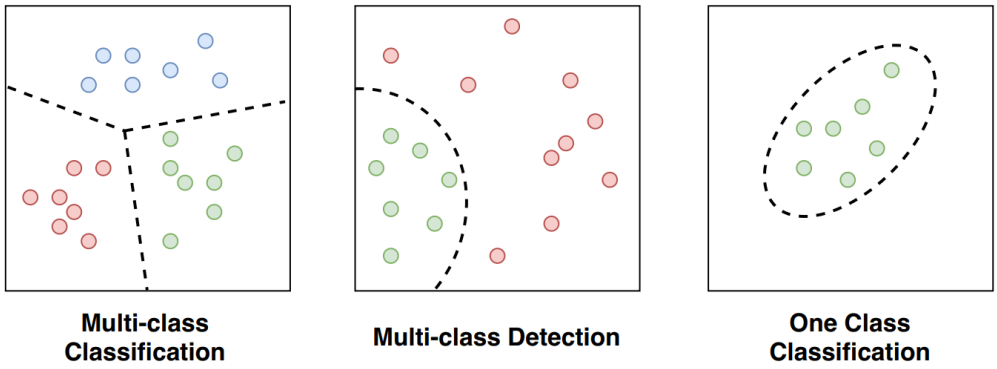


Figure 4.2: Comparisons between multi-class classification/detection and one class classification [12]

To ensure accurate classification, choosing an appropriate feature that enables a classifier to delineate regions where objects from each class consistently manifest is imperative. Analogous to multi-class classification, the task of One-Class Classification would also benefit from features that distinctly position data of the positive class apart from other objects in the environment. Nevertheless, the process of learning or selecting such a feature becomes more challenging in OCC, as there is an absence of non-positive data during the training phase: this is a key concept for the implemented solution and even more important when considering systems in the automotive, aerospace, or similar fields where some events are designed to happen but their frequency is way lower when compared to constantly running tasks. To make a concrete example, the OCC should be trained with HPC values registered over the airbag functioning, otherwise, their usage could be interpreted as a novelty to the regular system workflow. For these reasons, the OCC is trained using a specific set of five HPC where the three of them that are user-settable are chosen based on the kind of computation performed.

For the implementation of the OCC, the Python programming language was utilized. This choice has been taken considering the ease of development of the wanted solution, thanks to pre-existing libraries such as *scikit-learn* that come with all the tools in order to create the classifier and tune its parameters. After installing the library in the development environment, the first step is the one to perform some pre-computation on the HPC data directly coming from the Aurix

board: this procedure is needed to remove any formatting that is not needed and extract only the values of the HPC. After the pre-computation phase is performed, the classifier needs to be defined with its parameters and the training phase has to be started for the model to build its discriminating factors to classify future unknown data.

```
TRAIN_PERCENTAGE = 0.9
while True:
    threshold = int(TRAIN_PERCENTAGE * np_train_lines.shape[0])
    if threshold <= 0: break
    X_train = np_train_lines[:threshold]
    X_test = np_train_lines[threshold:]
    X_outliers = np_test_lines

    # fit the model
    clf = svm.OneClassSVM(kernel="rbf", gamma="scale", nu=0.01)
    clf.fit(X_train)

    y_pred_test = clf.predict(X_test)
    y_pred_outliers = clf.predict(X_outliers)
    n_error_test = y_pred_test[y_pred_test == -1].size
    n_error_outliers = y_pred_outliers[y_pred_outliers == 1].size

    print(f"Trained on {X_train.shape[0]} data - \
    Errors novel regular: {n_error_test}/{X_test.shape[0]} - \
    Errors novel abnormal: {n_error_outliers}/{X_outliers.shape[0]}")
    TRAIN_PERCENTAGE -= 0.05
```

A preliminary action that has to be performed is the definition of the amount of data that will be used to train the classifier. Initially, In the implementation, an amount of data corresponding to 90% (value of the *TRAIN_PERCENTAGE* variable) of all the HPC data is taken to perform the training phase and the remaining 10% will be used as test data. The value of *TRAIN_PERCENTAGE* will then be lowered at every iteration of the while loop in which all the code is contained: this is done to stress the classifier with the aim of understanding what is the minimum number of train data capable of training a model with good precision in detecting novelties while maintaining a low false positives rate. At this point, the classifier is instantiated by the *svm.OneClassSVM()* class method that accepts the following parameters:

- *kernel="rbf"*: Kernel functions have the objective of taking data as input and transforming it into the required form. In this case, it is specified the Gaussian radial basis function (RBF), one of the most utilized and versatile [13]. The function is defined by the following mathematical description of two equations:

$$K(x, x') = exp(-\gamma \left\| x - x' \right\|^2)$$

$$\gamma = \frac{1}{2\sigma^2}$$

The Euclidean word derives from the fact that the mathematical formula uses the Euclidean distance definition, which is key for the behavior of the classifier: the objective of this kind of kernel is to try to describe a way to group the positive data passed during training in order to define a model that circumscribes the data of a legitimate execution. When new or unknown data arrives, the Euclidean distance from the existing points of the model is useful to understand how similar this new data point is to the known ones. The great value of the radial basis function is that it performs well when a lot of dimensions are available, like in the case of this project, where we have a set of five HPC, each one of them potentially containing distinctive information about the monitored software.

- gamma and nu parameters: This mathematical function has different parameters that need to be specified in the code as well. Each of them has a specific meaning that directly impacts the ability of the model to classify data correctly and their values are selected by an extremely practical process that iterates with different values and selects the one that produces the best results. Nonetheless, each of these values has specific correlations on the model, in particular:
  - *gamma*: In a formal context, it can be stated that the gamma parameter serves to determine the extent of influence exerted by a single training example, with lower values indicating a wider range and higher values indicating a narrower range. The gamma parameters can be interpreted as the reciprocal of the radius of influence of the samples chosen by the model as support vectors: in the code, its value was set to *scale* which is a standard value defined by the *scikit-learn* library as *1 / (n_features * X.var())*.
  - *nu*: a value in the range from 0 (excluded) to 1. As the official documentation reports, it is an upper bound on the fraction of training errors and a lower bound of the fraction of support vectors.

After defining the classifier and its parameters, the *fit()* method is called over the created object, passing as an argument the list of train data previously defined. Once the model is created, it is possible to call the *predict()* method, which accepts as a parameter the list of data to be classified and returns a list where objects classified as positive will have a value of 1, while objects classified as negative will have a value of -1. By counting both values it is possible to yield the number of mistakes made by the classifier under the form of Fale Positives Rate (FPR) and False Negatives Rate (FNR).

4.3 Final Results

The experiment yielded some interesting results that are analyzed in this section. Presented results are expressed in the form of False Positive Rate (FPR) and False Negative Rate (FNR) to clearly highlight the precision of the model in classifying new data. The classifier was tested on 65 data points for both the legitimate values and the malicious ones.
For both experiments, the performed tests highlighted the behavior of the classifier to correctly detect 100% of the HPC values representing an attack. This is the only similarity of the two functions because the False Positive Rate highly depends on the kind of computations and the considered scenario.

4.3.1 Fibonacci Results

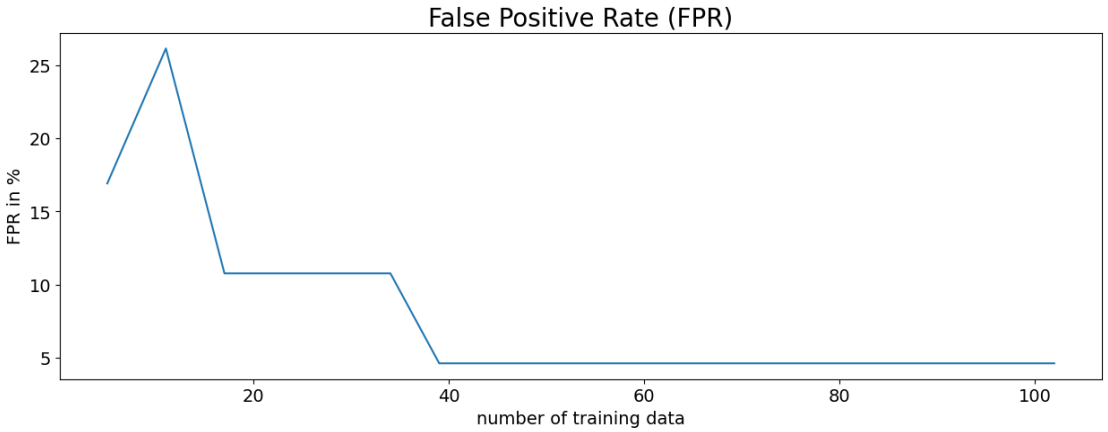


Figure 4.3: FPR graph for Fibonacci scenario

Starting from the Fibonacci example, it can be noticed a particularly high False Positive Rate. The explanation behind this is that, given the nature of the Fibonacci function, the routine will always be executed, meaning that a higher input value will result in a more expensive computation from the time and instructions point of view. Meanwhile, a lower input value will lead to a shorter execution, lowering the values of HPC.
When executing the example the following assumptions were made:

- If the input value to the Fibonacci function is in the range [0;50], the execution is considered legitimate (HPC values monitored in this case are the train data for the OCC)

- If the input value to the Fibonacci function is in the range]50;100], the execution is considered malicious

That being said, an important note has to be made when considering the extremes that separate a legitimate execution from a malicious one. If the ability to detect attacks remains optimal, the FPR is generally higher than the MAF example because, for the values where the nominal and malicious scenarios are similar, it will result in a more difficult classification. In the case of the example, input value 50 (legitimate) will have HPC results similar to value 51 (malicious). The graph shows that the best possible FPR for the test is 4.62% circa and it is achievable by training the model with at least 39 data points.

```
Trained on 102 data - Errors with good data: 3/65 (FPR: 4.62%)
Trained on 96 data - Errors with good data: 3/65 (FPR: 4.62%)
Trained on 91 data - Errors with good data: 3/65 (FPR: 4.62%)
Trained on 85 data - Errors with good data: 3/65 (FPR: 4.62%)
Trained on 79 data - Errors with good data: 3/65 (FPR: 4.62%)
Trained on 74 data - Errors with good data: 3/65 (FPR: 4.62%)
Trained on 68 data - Errors with good data: 3/65 (FPR: 4.62%)
Trained on 62 data - Errors with good data: 3/65 (FPR: 4.62%)
Trained on 56 data - Errors with good data: 3/65 (FPR: 4.62%)
Trained on 51 data - Errors with good data: 3/65 (FPR: 4.62%)
Trained on 45 data - Errors with good data: 3/65 (FPR: 4.62%)
Trained on 39 data - Errors with good data: 3/65 (FPR: 4.62%)
Trained on 34 data - Errors with good data: 7/65 (FPR: 10.77%)
Trained on 28 data - Errors with good data: 7/65 (FPR: 10.77%)
Trained on 22 data - Errors with good data: 7/65 (FPR: 10.77%)
Trained on 17 data - Errors with good data: 7/65 (FPR: 10.77%)
Trained on 11 data - Errors with good data: 17/65 (FPR: 26.15%)
Trained on 5 data - Errors with good data: 11/65 (FPR: 16.92%)
```

4.3.2 Moving Average Filter Results

Considering the Moving Average Filter example, the first look at the HPC values immediately shows how the function behaves when an attack is performed. In nominal scenarios, the window size parameter can have a value between 1 and 8 (both included) that translates to different values of HPC when measuring the function: if the window size is a low value, the execution of the function will be quicker and the number of instructions will be lower, together with the number pipeline stalls that, as a direct consequence of the low number of instructions issued, will be lower too. Instead, if the window size value is higher, all the HPC values will be higher too because of the longer computational effort. If an attack is performed in this kind of scenario, the MAF function is not executed at all. Even

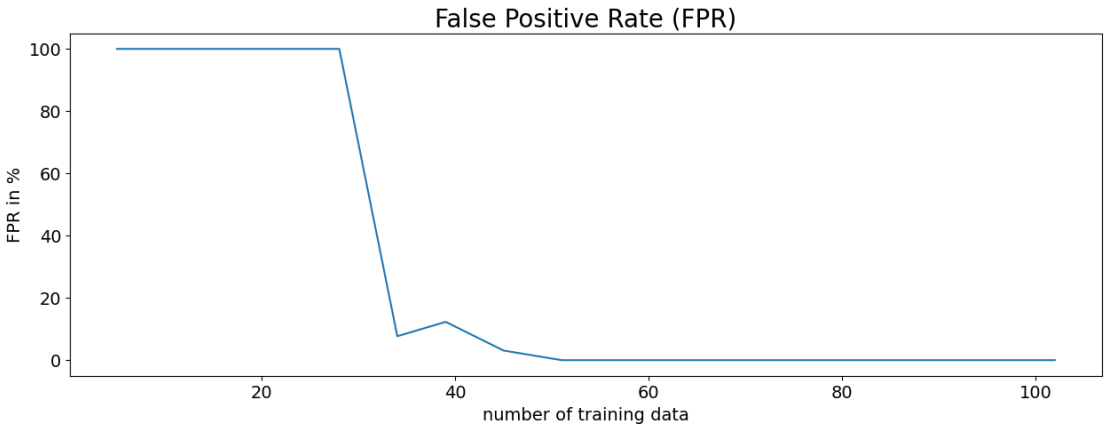


Figure 4.4: FPR graph for MAF scenario

if this check is not explicitly performed as an if statement inside the code, proper implementation of the filter itself should be enough to avoid the execution when the window size is out of range. In this case, the computation is skipped, and HPC reflects this event by having lower values. That being said, it is possible to find similarities between HPC value reflecting a legitimate execution with a small window size and a malicious execution since these values will be similar, resulting in a more complex model optimization.
The results in Image 4.4 clearly show how the trained classifier, in the run tests, is always capable of correctly detecting an ongoing attack inside the system. As a drawback, there is a small percentage of False Positives that depends on the amount of used training data.

```
Trained on 102 data - Errors with good data: 0/65 (FPR: 0.00%)
Trained on 96 data - Errors with good data: 0/65 (FPR: 0.00%)
Trained on 91 data - Errors with good data: 0/65 (FPR: 0.00%)
Trained on 85 data - Errors with good data: 0/65 (FPR: 0.00%)
Trained on 79 data - Errors with good data: 0/65 (FPR: 0.00%)
Trained on 74 data - Errors with good data: 0/65 (FPR: 0.00%)
Trained on 68 data - Errors with good data: 0/65 (FPR: 0.00%)
Trained on 62 data - Errors with good data: 0/65 (FPR: 0.00%)
Trained on 56 data - Errors with good data: 0/65 (FPR: 0.00%)
Trained on 51 data - Errors with good data: 0/65 (FPR: 0.00%)
Trained on 45 data - Errors with good data: 2/65 (FPR: 3.08%)
Trained on 39 data - Errors with good data: 8/65 (FPR: 12.31%)
Trained on 34 data - Errors with good data: 5/65 (FPR: 7.69%)
Trained on 28 data - Errors with good data: 65/65 (FPR: 100.00%)
Trained on 22 data - Errors with good data: 65/65 (FPR: 100.00%)
Trained on 17 data - Errors with good data: 65/65 (FPR: 100.00%)
```

```
17 Trained on 11 data - Errors with good data: 65/65 (FPR: 100.00%)
18 Trained on 5 data - Errors with good data: 65/65 (FPR: 100.00%)
```

As can be seen in the above output, the classifier can be considered efficient if trained with 51 data or more, while lowering the training data will result in awful detection capabilities, nullifying the performance gains.

Chapter 5

Conclusions

5.1 Future Work

This thesis project aimed at finding useful implementations of a classification algorithm in a Real-Time context exploiting HPC as a way to extract training data. In the starting implementation described in this document, only some of the possible use cases were considered, resulting in many possible future improvements. In fact, two of the main constraining decisions that were taken during the design phase focused on the architecture and the RTOS: as hinted in section 2.3, different architectures will have different sets of HPC, which may lead to different ways to monitor software execution, opening new paths for the detection of attacks, while RTOS is one of the discriminating factors when choosing the architecture itself since specific support has to be available in order to run it on a specific platform. In this work, an RTOS was emulated in order to create a working proof of work for the desired goal, focusing all the attention on the development of the IDS and not on the porting of existing software for the TriCore Infineon architecture. Despite this, some efforts were made in order to understand how a future implementation of the project could be developed using a real RTOS and how existing libraries like iLLD could be integrated into the programming environments, a goal that could be useful from the point of view of the ease of programming and effectiveness of the final solution since the library is shared by the manufacturer and used and tested throughout the years.

5.1.1 Using a Real RTOS

Section 3.2 explains how the RTOS in the final implementation is emulated thanks to a simple bare-metal scheduler and the definition of two fictitious tasks that are run following a fixed schedule. In a real-life scenario, one that we can find, for example, in the automotive industry where hundreds of ECUs are deployed in a

single commercial vehicle, each board will run a real RTOS, carefully chosen and configured for the specific application. To extend the work proposed in this report, deploying an actual RTOS on the Aurix board will be an important improvement to better understand possible situations that may arise in a more real-life scenario.

When the design phase of the project began, the initial idea was to use a commonly utilized RTOS in the automotive industry called Erika Enterprise: this entirely made-in-Italy operating system is compliant with the OSEK/VDX standard and has support for a multitude of architectures like TriCore, ARM, x86, AVR, and others. With the assistance of this operating system, the inherent capabilities of the multicore architecture can be effectively harnessed. It is fully compatible and configurable for executing Real-Time parallel code on such architectures. The RTOS Reference Manual [14] describes Erika Enterprise as providing a real-time scheduler and resource managers, enabling the complete utilization of the potential of modern microcontrollers and multicore platforms. Moreover, it ensures predictable real-time performance while retaining the programming model of traditional single-processor architectures. On top of all the features and directions of the OSEK/VDX standard, Erika also has some valuable features such as stack sharing between different tasks to optimize RAM usage or hook functions that can be called before and after each context switch between tasks. Its schedule algorithm is a fixed priority design, and in the OIL file, each task is assigned to a priority level.

Figure 5.1: Erika Enterprise logo

To be able to program and configure Erika Enterprise the RT-Druid Development Environment has to be used: this IDE is a custom version of the Eclipse IDE with the integration of a plugin that makes it possible to retrieve the latest Erika version from a local repository easily and to perform all the compilation work based on the specification defined in the OIL file, adapting the solution to the wanted architecture. Not only: in the OIL file, it is possible to define which compiler has to be used. When compiling for the TriCore architecture, two possible solutions could be adopted:

- Hightec compiler: this compiler has a free version called Hightech Free Toolchain that can be used without buying a license.
- TASKING compiler: this compiler is used internally by Aurix Development Studio and is only available as paid software.

Specifying the compiler in the OIL file is necessary because Erika has a build system based on many connected Makefiles where a reference to the executable of the compiler is present. As a note, using the TASKING compiler embedded in Aurix Development Studio is impossible because the specific version is only usable when directly called by ADS and not by some external commands, like those in a Makefile. That being said, it could be helpful to understand how to integrate the build process of Erika directly in ADS in order to build and compile Erika projects easily and debug applications without the need to use some external debugger.

5.1.2 iLLD usage under ERIKA OS

For the bare-metal implementation, the iLLD library has been extensively utilized to set the CAN communications, start, stop, and reset performance counters, and use useful debugging features like printing text on a simulated filesystem. The library, developed by Infineon, offers a simple way to handle all the different features that the architecture and the development board have to offer, and it is present in a lot of the example projects on Aurix Development Studio. The complex code base has been tested and documented by the company, and its use is highly recommended. Still, the compatibility with the Erika build process is not straightforward and some work needs to be done to be able to use the provided functions inside tasks running on Erika RTOS. In this section, the process of compiling and including the library within the Erika environment is described, together with how some of the functionalities can be included and the difference in handling specific scenarios, such as interrupts.

Including iLLD in Erika

As explained before, everything regarding the operating system configuration must be done via the OIL file since Erika is compliant with the OSEK/VDX standard.

```
1  CPU mySystem {
2    OS myOs {
3      EE_OPT = "OSEE_DEBUG";
4      EE_OPT = "OS_EE_VERBOSE";
5      EE_OPT = "OS_EE_APPL_BUILD_DEBUG";
6      EE_OPT = "OS_EE_BUILD_DEBUG";
7
8      // iLLD include paths
9      CFLAGS = "-IE:\\dev\\eclipse-workspace\\TricorePerfCounter\\
       Libraries\\iLLD\\TC29B\\Tricore\\Psi5s\\Psi5s";
10     CFLAGS = "-IE:\\dev\\eclipse-workspace\\TricorePerfCounter\\
       Libraries\\iLLD\\TC29B\\Tricore\\Eth\\Phy_Pef7071";
```

```
11      CFLAGS = "-IE:\\dev\\eclipse-workspace\\TricorePerfCounter\\
        Libraries\\iLLD\\TC29B\\Tricore\\Fce";
12      ...
```

In the code listing a small snippet of an OIL file is shown. In this file, some flags are shown that are useful to specify some build configurations, like the verbose option, the debug one, etc... But what's important here is the definition of the *CFLAGS* constants, where the paths of all the folders of the iLLD library have to be specified in order to correctly include all the header files contained in them. Definitions of these constants have to be made in the *OS* section. After doing this procedure, the *.c* files that correspond to the used header files present in the previous paths need to be specified as well since their inclusion is not immediate.

```
1  APPDATA tricore_full_2 {
2       APP_SRC = "code.c";
3       APP_SRC = "MULTICAN.c";
4
5       // _Impl
6       APP_SRC = "Libraries/iLLD/TC29B/Tricore/_Impl/IfxAsclin_cfg.c";
7       APP_SRC = "Libraries/iLLD/TC29B/Tricore/_Impl/IfxCpu_cfg.c";
8       APP_SRC = "Libraries/iLLD/TC29B/Tricore/_Impl/IfxScu_cfg.c";
9       APP_SRC = "Libraries/iLLD/TC29B/Tricore/_Impl/IfxSrc_cfg.c";
10      APP_SRC = "Libraries/iLLD/TC29B/Tricore/_Impl/IfxStm_cfg.c";
11      APP_SRC = "Libraries/iLLD/TC29B/Tricore/_Impl/IfxPort_cfg.c";
12
13      // In use
14      APP_SRC = "Libraries/iLLD/TC29B/Tricore/Asclin/Std/IfxAsclin.c";
15      APP_SRC = "Libraries/iLLD/TC29B/Tricore/Scu/Std/IfxScuWdt.c";
16      APP_SRC = "Libraries/iLLD/TC29B/Tricore/Port/Std/IfxPort.c";
17      APP_SRC = "Libraries/iLLD/TC29B/Tricore/Scu/Std/IfxScuCcu.c";
18      APP_SRC = "Libraries/iLLD/TC29B/Tricore/Cpu/Std/IfxCpu.c";
19 };
```

As can be seen by the above example, the *APPDATA* section requires the developer to specify the source code files that the application uses: the *code.c* file is the main file, while the others are present in the iLLD library, containing the function definitions of the routines directly called in the source code. This process requires a bit of error handling: when building the project, errors will highlight that some of the called functions are not defined, a clear hint of the need of the correspondent *.c* files to be added in the above section.

Interrupts definition

Interrupts are an essential feature for the project since they are needed to simulate a more close-to-reality scenario in which the board exchanges data via the CAN

bus with other ECUs or sensors. While their definition is straightforward in the bare-metal implementation, using Erika is not as easy. The RTOS has a different way of defining the interrupts, which involves the OIL file, as usual, being the operating system compliant with the OSEK/VDX standard. The following code represents an example of the definition of an interrupt.

```
ISR asclin0TxISR {
    CATEGORY = 2;
    SOURCE = "ASCLIN0TX";
    PRIORITY = 10;
};
```

The official Erika Wiki [15] reports the following instructions: *"Due to the special implementation of the Interrupt Vector in AURIX architecture, each entry of the vector table is simply identified by its priority value. The SRN register of the source is configured by ERIKA3 during the StartOS. Valid values for the SOURCE field of an ISR follow the same naming convention used by Infineon inside its own iLLDs (Infineon Low-Level Drivers)"*. The presented code defines an *ISR* code block where the developer can specify the name, which needs to be the same as the function that will be called once the interrupt is fired. On top of that, the proprieties of the interrupt service routine have to be declared and assigned:

- *CATEGORY* is the kind of interrupt (this attribute is specified in the OSEK/VDX standard)
  - If assigned to 1, the function cannot call all of the OS primitives
  - If assigned to 2, the function can call all of the OS primitives
- *SOURCE*: the interrupt source ID or source number. This value is usually a string, and it is architecture-dependent.
- *PRIORITY*: the interrupt priority

These parameters are the mandatory ones, but some other optional ones are available, such as the possibility of specifying what core should handle the interrupt (*CPU_ID*) or defining the ISR as a TRAP (if done, the *CATEGORY* and *PRIORITY* values are ignored).

When trying to implement the solution with Erika's support, the interrupt definition was one of the activities that could not be solved, mainly due to a lack of documentation. The only ISR that could be correctly defined and executed is the one presented in the previous code snippet: in that case, the UART peripheral is used to print characters on the serial monitor when the Aurix board is connected to a host PC via a USB cable. When trying to define the ISR for the CAN bus

(both for a more standard implementation like the raise of an interrupt at every frame received, as well as for the presented solution with the queue of messages raising interrupt at overflow), it wasn't possible to find the list of names defined in the iLLD library for all the peripherals able to generate interrupts. Moreover, using the *IFX_INTERRUPT()* defined in iLLD is useless because it is not a static definition.

RTOS integration of HPC monitoring

As an OSEK/VDX-compliant operating system, Erika Enterprise allows the developer to define and call Hook routines. The Hook routines explained in the standard document are system-specific functions to allow user-defined actions within the OS's internal processing. The operating system calls them in a particular context, depending on the implementation of the OS. They have higher priorities than all the running tasks and cannot be interrupted by interrupts defined with category 2. The action taken by these functions is defined by the developer using the OIL file and can be used at different moments during the execution:

- System startup: the hook routine called *StartupHook* is called just after the operating system startup and before the scheduler is run.
- System shutdown: the hook routine called *ShutdownHook* is called when a task explicitly requests the OS shutdown or before a severe horror occurs.
- Context-switch: in this case, two hook routines are defined, called *PostTaskHook* and *PreTaskHook*, executed respectively before and after a context switch is started.

The third option allows the developer to perform some exciting computation practical also for debugging purposes, and the standard specification explicitly states that they *"may be used for debugging or time measurement (including context switch time)"* [2].

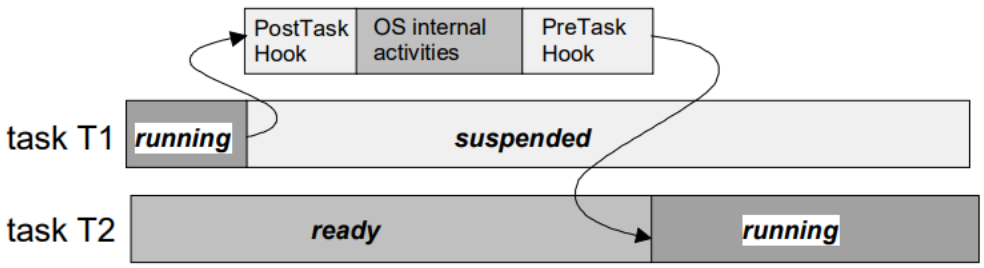


Figure 5.2: PostTaskHook and PreTaskHook [2]

With this concept in mind, it is natural to think about a possible implementation of HPC handling directly inserted in Hook routines executed at context switch: before the task is run, the performance counters could be reset and re-started, and when a task is ended, the values of the registers will be saved or transmitted to the classifier, and then reset to be started again.

In Erika, the specification has to be done in the OIL file, as usual, by specifying the following code block inside the *OS* section of the file:

```
1 PRETASKHOOK = TRUE {
2       HOOKNAME = "pre_hook";
3 };
4 POSTTASKHOOK   = TRUE {
5       HOOKNAME = "post_hook";
6 };
```

First, the option specific to the kind of hook needs to be set to the *TRUE* value, and then the function specification to run when the hook rises is defined with the *HOOKNAME* parameter. The hook name must be a valid C function without input or output parameters, like *void hook(void)*.

This solution could be optimal for tracking architectural events happening during a single task execution and for providing ad-hoc classification.

5.1.3 Real-time Intrusion Detection

The whole project is based on developing an IDS, usually a passive system that aims to detect possible ongoing attacks without triggering any security mechanism. This is not always true, though, and in the automotive context, it could be helpful to implement such a feature (paying particular attention to the False Positive/False Negative rates of the classifier) to guarantee security and safety for the people that may use a vehicle. To do this job, an entity separate from the RTOS should be utilized since the classification, and the reaction could take a non-negligible amount of time, possibly leading to dangerous changes to the RTOS throughput. Evidence S.r.l. recently proposed a single-board dual-OS system that aims to combine the flexibility of the Linux general-purpose operating system, which is capable of performing any complex calculations, with the reliability of the automotive-grade ERIKA Enterprise operating system that is capable of executing commands triggered by Linux. The two operating systems run on dedicated cores and share memory with limited support for memory protection for efficiency purposes. However, this system lacks proper isolation between safety-critical and non-safety-critical components, which is necessary for safety certification. A recent solution [16] presented (on the ARM architecture) an improved implementation of a double-OS system running ERIKA Enterprise and a full-featured Linux OS on a

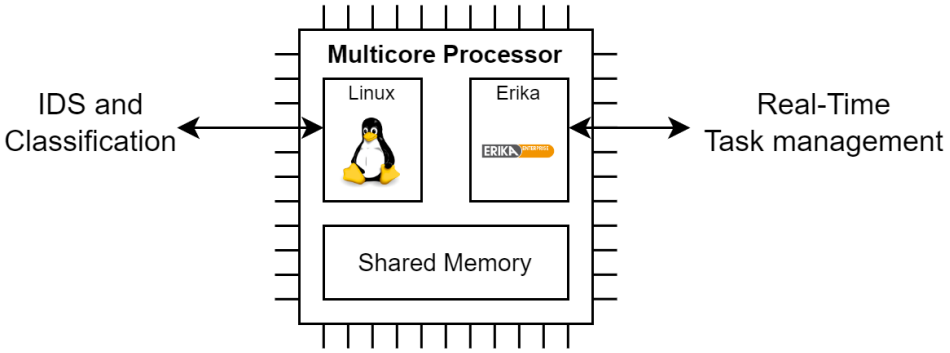


Figure 5.3: Multicore design of Linux and Erika on different cores

dual-core platform. Xen, a type 1 hypervisor, runs the two operating systems in isolated domains. Each domain runs on a dedicated core statically assigned by the hypervisor.

A hypervisor is an essential software component that enables the implementation of virtualization. Its primary function is establishing a virtualization layer, effectively separating the tangible hardware elements such as processors, RAM, and other physical resources from the virtual machines and their respective operating systems. Hypervisors simulate available resources, thereby enabling guest machines to utilize them. Regardless of the operating system being initiated on a virtual machine, it will perceive that genuine physical hardware is at its disposal. There exist two kinds of hypervisors:

- Type 1: also called bare-metal or native hypervisors. They are simple operating systems only able to virtualize other OS, configurable to set the resources that every guest OS will have.

- Type 2: also known as hosted hypervisors. They are applications that run on top of existing operating systems to virtualize other OS.

In the project scenario, Linux would serve as the control domain and could execute any components of the Xen tool stack. It would also grant the real-time operating system access to any I/O-memory range required for control tasks. The described system would also provide a simple and secure communication mechanism between the two operating systems. This mechanism would be based on Xen's inter-domain event notification primitives and the real-time operating system's explicit sharing of a dedicated set of memory pages.

5.2 Final Statements

The results that came from the project's development are an essential indicator that an Intrusion Detection System can be a crucial factor in the future development of ECU software inside vehicles. Not only does the system aim to protect the computation being executed on the board, but the close relationship between RTOS and real-time computation and the safety of people on board vehicles is significant. The most important data this work produced is the number of False Positives generated by a solution implemented in the described method: while classification seems optimal for situations where actual attacks are executed, the detection often mistakes legitimate actions as malicious, situations that could lead to potential harm. Further study can be done about how to train the model to lower this kind of event while always paying attention to the size of the model itself, which has to be small and fast so that it can be run on computationally constrained systems such as embedded ones.
Moreover, succeeding in executing a real RTOS will lead to essential data gathering and more significant situations related to production-ready devices. With multiple tasks, maybe even different from each other, different HPCs will be involved to depict a significant image of the running system, which will inevitably complicate correctly detecting attacks. Besides achieving high precision in detecting attacks, when dealing with an RTOS, the timings of the tasks running on the system must be precise and not modified by running the routines handling the HPC or, potentially, even the IDS itself. Despite these notions that could be useful to improve the project (some effort was already made, as described in section 5.1), valuable and insightful results have been recorded, highlighting the differences in HPC values when monitoring different functions that follow different behaviours. The central concept proved with the project's development is that the IDS has to be trained over the specific function monitored by HPC. Sadly, a more comprehensive approach where the IDS can detect attacks for general computation seems unfeasible.

Appendix A

Interrupt Service Routine

```
#define MOVEAVG      1
void canIsrOverflowHandler(void) {
    enablePerfCountersWithMultiCount(
            0x000,   //  IP_DISPATCH_STALL
            0x000,   //  LS_DISPATCH_STALL
            0x011    //  DMEM_STALL
    );
    IfxMultican_Status readStatus;
    uint8 currentCanMessage;
    static volatile uint32 numOfReceivedMessages = 0;

    for(currentCanMessage = 0; currentCanMessage < RX_FIFO_SIZE; currentCanMessage++) {
        /* Read the received CAN message and store the status of the operation */
        readStatus = IfxMultican_Can_MsgObj_readMessage(&g_multican.canDstMsgObj, &g_multican.rxMsg[currentCanMessage]);

        /* If no new data has been received, report an error */
        if(readStatus != IfxMultican_Status_newData) {
            g_status = CanCommunicationStatus_Error_noNewDataReceived;
            printf("No new data\n");
        }

        /* If a new data has been received but one message was lost, report an error */
        if(readStatus == IfxMultican_Status_newDataButOneLost) {
            g_status = CanCommunicationStatus_Error_newDataButOneLost;
            printf("New data but one lost\n");
```

```
26          }
27
28          /* If there was no error, increment the counter to indicate
    the number of successfully received CAN messages */
29          if (g_status == CanCommunicationStatus_Success) {
30              // ID: 0x777 - Random value between 0-7, toggling leds
31              if(g_multican.rxMsg[currentCanMessage].id == 0x777) {
32                  uint8 led_number = g_multican.rxMsg[currentCanMessage
    ].data[1] >> 24;
33                  IfxPort_togglePin(&MODULE_P33, PIN0 + led_number);
34              }
35
36              // ID: 0x700 - Window size value
37              if(g_multican.rxMsg[currentCanMessage].id == 0x700) {
38                  local_windowSize = g_multican.rxMsg[currentCanMessage
    ].data[0] & 0xffUL;
39              }
40
41              //ID 0x722 - Values of the signal
42              if(g_multican.rxMsg[currentCanMessage].id == 0x722) {
43                  printf("[%lu] * ", numOfReceivedMessages);
44#if !MOVEAVG
45                  uint8 n = g_multican.rxMsg[currentCanMessage].data[1]
     >> 24;
46                  uint64 a =  0, b = 1, c;
47                  for(int i = 1; i < n; i++) {
48                    c = a+b; a = b; b = c;
49                  }
50                  printf("%d of fibonacci is %llu * ", n, a);
51#else
52                  if(local_windowSize <= 8 && local_windowSize >= 1) {
53                      uint8 resultData[8];
54                      uint8 data[8];
55                      // Reconstructing data from the uint32 values in
    the CAN node struct
56                      for(uint8 x = 0; x < 2; x++) {
57                          for(uint8 y = 0; y < 4; y++) {
58                              uint8 d = (g_multican.rxMsg[
    currentCanMessage].data[x] & (0xff << (y * 8))) >> (y * 8);
59                              data[y + (x * 4)] = d;
60                          }
61                      }
62                      movingAverage(data, resultData, 8,
    local_windowSize);
63                  }
64                  local_windowSize = 100;
65#endif
66              }
67              numOfReceivedMessages++;
```

```
68              }
69          }
70
71          g_perfCounts = IfxCpu_stopCounters();
72          printPerfCounters(g_perfCounts, 0);
73      }
```

Appendix B

One Class Classifier Python Implementation

```
import sys
import numpy as np
from sklearn import svm

TRAIN_PERCENTAGE = 0.9

def clean_data(data, debug_print = False):
    for id, d in enumerate(data):
        d = d[d.find('*') + 1:]
        d = d.replace("(0)", "").replace("\n", "").replace("CCNT:", "
    ").replace("ICNT:", "").replace("M1CNT:", "").replace("M2CNT:", ""
    ).replace("M3CNT:", "").replace(" ", "").replace(",0", "")
        data[id] = [int(x) for x in d.split(",")]
        if debug_print: print(data[id])

# Opening file specified in the cmd argument
filename_train = sys.argv[1]
filename_test = sys.argv[2]

HPCfile_train = open(filename_train, "r")
train_lines = HPCfile_train.readlines()

HPCfile_test = open(filename_test, "r")
test_lines = HPCfile_test.readlines()

# Pre processing, sanitize data
clean_data(train_lines)
clean_data(test_lines)
```

```
np_train_lines = np.array(train_lines)
np_test_lines = np.array(test_lines)

while True:
    threshold = int(TRAIN_PERCENTAGE * np_train_lines.shape[0])
    if threshold <= 0: break
    X_train = np_train_lines[:threshold]
    X_test = np_train_lines[threshold:]
    X_outliers = np_test_lines

    # fit the model
    clf = svm.OneClassSVM(nu=0.01, kernel="rbf", gamma="scale")
    clf.fit(X_train)

    y_pred_train = clf.predict(X_train)
    y_pred_test = clf.predict(X_test)
    y_pred_outliers = clf.predict(X_outliers)
    n_error_train = y_pred_train[y_pred_train == -1].size
    n_error_test = y_pred_test[y_pred_test == -1].size
    n_error_outliers = y_pred_outliers[y_pred_outliers == 1].size

    print(f"Trained on {X_train.shape[0]} data - Error train: {
    n_error_train}/{X_train.shape[0]} - \
    Errors novel regular: {n_error_test}/{X_test.shape[0]} - \
    Errors novel abnormal: {n_error_outliers}/{X_outliers.shape[0]}")
    TRAIN_PERCENTAGE -= 0.05
```

Appendix C

Bare-metal RTOS Emulation

```
#include "IfxStm.h"
#include "Bsp.h"

#include "Ifx_Types.h"
#include "IfxCpu.h"
#include "IfxScuWdt.h"
#include "MULTICAN_RX_FIFO.h"
#include "CPU_Perf_Counters.h"

#include <stdio.h>

IfxCpu_syncEvent g_cpuSyncEvent = 0;

// This function with these parameters take maximum 3ms, while considering interrupts
void TASK(void) {
    uint32 i;
    for(i = 0; i < 100000; i++) { }
    for(i = 0; i < 100000; i++) { }
}

int core0_main(void) {
    IfxCpu_enableInterrupts();

    /* !!WATCHDOG0 AND SAFETY WATCHDOG ARE DISABLED HERE!!
     * Enable the watchdogs and service them periodically if it is required
     */
    IfxScuWdt_disableCpuWatchdog(IfxScuWdt_getCpuWatchdogPassword());
    IfxScuWdt_disableSafetyWatchdog(IfxScuWdt_getSafetyWatchdogPassword());
```

```
29
30      /* Wait for CPU sync event */
31      IfxCpu_emitEvent(&g_cpuSyncEvent);
32      IfxCpu_waitEvent(&g_cpuSyncEvent, 1);
33
34      /* Application code: initialization of MULTICAN and transmission
        of the CAN messages */
35      initMultican();
36      initLed();
37
38      // Amount of ticks needed to count a 25ms wait time
39      Ifx_TickTime ticksFor25ms = IfxStm_getTicksFromMilliseconds(&
        MODULE_STM0, 25);
40
41      uint8 i = 0;
42      uint64 t1, t2, tdelta;
43      while(1) {
44          t1 = IfxStm_get(&MODULE_STM0);
45          TASK();      // corresponds to the task executing every 25ms
46          i++;
47          if(i == 4) {
48              TASK();      // corresponds to the task executing every
        100ms
49              i = 0;
50          }
51          t2 = IfxStm_get(&MODULE_STM0);
52          // ticks passed since beginning of execution
53          tdelta = t2 - t1;
54          waitTime(ticksFor25ms - tdelta);
55      }
56      return (1);
57  }
```

Bibliography

[1] *CAN Specification*. Bosch. 1991 (cit. on pp. 3, 5).

[2] *Operating System*. OSEK/VDX. 2005 (cit. on pp. 10, 11, 51).

[3] *OIL: OSEK Implementation Language*. OSEK/VDX. 2001 (cit. on p. 13).

[4] David A. Patterson John L. Hennessy. *Computer Architecture - A Quantitive Approach. Sixth Edition*. Cambridge, MA: Morgan Kaufmann, 2019 (cit. on p. 14).

[5] «Spectre and Meltdown processor flaws threaten billions of computers and mobile devices». eng. In: *Computer fraud & security* 2018.1 (2018), pp. 1, 3–1, 3. ISSN: 1361-3723 (cit. on p. 16).

[6] Congmiao Li and Jean-Luc Gaudiot. «Detecting Spectre Attacks Using Hardware Performance Counters». eng. In: *IEEE transactions on computers* 71.6 (2022), pp. 1320–1331. ISSN: 0018-9340 (cit. on p. 16).

[7] Rebecca Bace and Peter Mell. *NIST Special Publication on Intrusion Detection Systems*. eng. 2001 (cit. on p. 16).

[8] Bifta Sama Bari, Kumar Yelamarthi, and Sheikh Ghafoor. «Intrusion Detection in Vehicle Controller Area Network (CAN) Bus Using Machine Learning: A Comparative Performance Study». eng. In: *Sensors (Basel, Switzerland)* 23.7 (2023), pp. 3610–. ISSN: 1424-8220 (cit. on p. 17).

[9] *TC27x D-Step - 32-Bit Single-Chip Microcontroller*. Infineon. 2014 (cit. on p. 32).

[10] Mehdi Fallahpour and David Megías. «Audio watermarking based on Fibonacci numbers». eng. In: *IEEE/ACM transactions on audio, speech, and language processing* 23.8 (2015), pp. 1273–1282. ISSN: 2329-9290 (cit. on p. 35).

[11] S. W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. eng. San Diego, CA, USA: California Technical Publishing, 1999 (cit. on p. 37).

[12] Pramuditha Perera, Poojan Oza, and Vishal M Patel. «One-Class Classification: A Survey». eng. In: (2021) (cit. on pp. 38, 39).

[13] Sadeep Jayasumana, Richard Hartley, Mathieu Salzmann, Hongdong Li, and Mehrtash Harandi. «Kernel Methods on Riemannian Manifolds with Gaussian RBF Kernels». eng. In: *IEEE transactions on pattern analysis and machine intelligence* 37.12 (2015), pp. 2464–2477. ISSN: 0162-8828 (cit. on p. 40).

[14] *ERIKA Enterprise Manual, Real-Time made easy.* Version 1.4.5. Evidence S.r.l. 2012 (cit. on p. 47).

[15] Evidence S.r.l. *Infineon Tricore AURIX - Interrupt Handling.* URL: `https://www.erika-enterprise.com/wiki/index.php/Infineon_Tricore_AURIX#Interrupt_Handling` (cit. on p. 50).

[16] Arianna Avanzini, Paolo Valente, Dario Faggioli, and Paolo Gai. «Integrating Linux and the real-time ERIKA OS through the Xen hypervisor». eng. In: *10th IEEE International Symposium on Industrial Embedded Systems (SIES).* IEEE, 2015, pp. 1–7. ISBN: 1467377112 (cit. on p. 52).