# Integration Methods for Host Intrusion Detection into Embedded Mixed-Criticality Systems

vom
Fachbereich Elektrotechnik und Informationstechnik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades eines

**Doktor der Ingenieurwissenschaften (Dr.-Ing.)**

genehmigte Dissertation

von
Marine Kadar
geboren in Saint-Étienne, France

**D 386**

| | |
|---|---|
| Eingereicht am: | 23.02.2022 |
| Tag der mündlichen Prüfung: | 25.03.2022 |
| Dekan des Fachbereichs: | Prof. Dr. rer. nat Marco Rahm |
| | |
| Promotionskomission | |
| Vorsitzender: | Prof. Dr.-Ing. Wolfgang Kunz |
| Berichterstattende: | Prof. Dipl.-Ing. Dr. Gerhard Fohler |
| | Prof. Dr. Marcus Völp |

# Erklärung gem. § 6 Abs. 3 Promotionsordnung

Ich versichere, dass ich diese Dissertation selbst und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt und die aus den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Dissertation wurde weder als Ganzes noch in Teilen als Prüfungsarbeit für eine staatliche oder andere wissenschaftliche Prüfung eingereicht. Es wurde weder diese noch eine andere Abhandlung bei einem anderen Fachbereich oder einer anderen Universität als Dissertation eingereicht.

12.04.2022

(Kaiserslautern, Datum)

(Marine Kadar)

# Abstract

With the growing support for features such as hardware virtualization tied to the boost of hardware capacity, embedded systems are now able to regroup many software components on a same hardware platform to save costs. This evolution has raised system complexity, motivating the introduction of Mixed-Criticality Systems (MCS) to consolidate applications from different criticality levels on a hardware target: in critical environments such as an aircraft or a factory floor, high-critical functions are now regrouped with other non-critical functions. A key requirement of such system is to guarantee that the execution of a critical function cannot be compromised by other functions, especially by ones with a lower-criticality level. In this context, runtime intrusion detection contributes to secure system execution to avoid an intentional misbehavior in critical applications.

Host Intrusion Detection Systems (HIDS) has been an active field of research for computer security for more than two decades. The goal of HIDS is to detect traces of malicious activity in the execution of a monitored software at runtime. While this topic has been extensively investigated for general-purpose computers, its application in the specific context of embedded MCS is comparatively more recent.

We extend the domain of HIDS research towards HIDS deployment into industrial embedded MCS. For this, we provide a review of state-of-the-art HIDS solutions and evaluate the main problems towards a deployment into an industrial embedded MCS.

We present several HIDS approaches based on solutions for general-purpose computers, which we apply to protect the execution of an application running into an embedded MCS. We introduce two main HIDS methods to protect the execution of a given user-level application. Because of possible criticality constraints of the monitored application, such as industrial certification aspects, our solutions support transparent monitoring; i.e. they do not require application instrumentation. On one hand, we propose a machine-learning (ML) based framework to monitor low-level system events transparently. On the other hand, we introduce a hardware-assisted control-flow monitoring framework to deploy control-flow integrity monitoring without instrumentation of the monitored application.

We provide a methodology to integrate and evaluate HIDS mechanisms into an embedded MCS. We evaluate and implement our monitoring solutions on a practical industrial platform, using generic hardware system and SYSGO's industrial real-time hypervisor.

*To my parents and grandparents.*

# Acknowledgment

During these last four years, I have professionnally and personnally learned a lot. I have been through a first industrial experience, doing PhD research; I moved to Germany and discovered the subtility of German food and bureaucracy; like everyone else, I also had to deal with the Covid-19 pandemie. I met many people on the way, who helped me – directly or indirectly – to achieve this work.

I am deeply grateful to Prof. Gerhard Fohler, my PhD advisor at TU Kaiserslautern, for his constant guidance and encouraging support all along this work. With him, I learned a lot about real-time systems research, and more generally about academic research. Working together helped me to sharpen my scientific perceptions with more simplicity and accuracy. I sincerely thank Philipp Gorski, my PhD advisor at SYSGO during the last two years, for his many helpful feedback, his positive and motivating support despite the long distance between Mainz and Rostock. He provided me with invaluable technical guidance and programming tips. His software code is a magical source of inspiration for my everyday programming work. I also particularly thank Sergey Tverdyshev, my former PhD advisor at SYSGO during the two first years, as he trusted in me for this research opportunity. He provided me with precious guidance and work methods, which guided this whole work. He also introduced me to SYSGO and industrial research practises.

I am thankful to Prof. Marcus Völp for accepting to be on my PhD Defense Committee and to Prof. Wolfgang Kunz for accepting to chair the Committee.

I want to thank the co-authors of my publications for their collaboration: Jan Ruh, Patrick Denzler, and Cosmin Avasalcai. I am grateful to the anonymous reviewers of my publications, who helped me to improve my work and to structure my knowledge on intrusion detection for embedded mixed-criticality systems.

During my PhD time at SYSGO, I had the opportunity to grow personnally and professionally in a friendly work environment. I am grateful to Don Kuzhiellil for the constructive discussions and collaborations during this work. I want to thank Bertrand Marquis, for his great support in the beginning of the PhD. Many thanks to Caspar Gries, my tutor when I joined SYSGO, who has been a fantastic fellow office-mate for two years. More largely, I am grateful to the overall R&T team – past and current – members: Holger Blasum, Mario Brotz, Zeeshan Ansar, Enkhtuvshin Janchivnyambuu, Axel von Blomberg, Stephan Wagner, Ingo Speer, and Torsten Voegler. Many thanks to Etienne Butery for his trust and consideration in my work, as well as for his constant

support. He gave me the opportunity to make connections and to improve visibility of my work outside SYSGO, leading to enriching discussions with industrial experts on embedded mixed-criticality systems, machine-learning, and security. Many thanks to my former colleagues Andrea Bastoni and Alex Zuepke for their valuable feedback on this work. Special thanks to the international lunch crew: Konstantin Tarandevich, Alex Dehmel, Agostino Mascitti. Your everyday good mood and humor were of great help to relax, even in stressful times. Many thanks to the pillars of SYSGO office, who nurture an enjoyable and positive work environment: Nicole Hirschmann, Kerstin Henss, Pia Marchlewitz, Britta Ehlenberger, Kai Sablotny, and Cristina Da Costa. Thank you for your kindness and support with administrative issues. More broadly, I wish to thank my fellow colleagues, Benjamin, Daniel, Jasmin, Jasmine, Frank, Armin, Tibor, Oliver, Frank, Wolfgang Andreas, David, Patrick, Henrik, Rudolf, Franz, Matthias, José, Fanny, Guillaume, Yannick, Jean-Jacques, Antonios, and Jérôme for their friendly support.

I warmly thank my colleagues at the Real-Time Systems Chair of TU Kaiserslautern. Many thanks to Stephanie Jung and Markus Mueller for their support with the administrative and technical issues at the university. Special thanks to the former and current PhD students of the Chair: Kristin Krüger, Gautam Gala, Florian Heilmann, Carlos Rodriguez, Luiz Maia, Ibrahim Alkoudsi, Gabriele Monaco, Alexandre Venito, Ankit Agrawal, Rodrigo Cohelo. Thank you for the many interesting discussions and for your friendly support, especially during the mentally challenging long lockdown periods.

I had the opportunity to supervise master-level students for internship and theses, which contributed to this research. Many thanks to Jordan Sautreau and Andrés Simancas Mateus for the fun and fruitful collaborations.

Last but not least, I am deeply grateful to my family, especially my parents and grand-parents who kept pushing me forward. Thank you for being there, always. My special thanks go to my partner, Christian, who accompanied me in this journey, from Grenoble to Mainz. Thank you for supporting me all the way, even in difficult times. I am so lucky and happy to have you by my side.

Marine Kadar
Mainz, 02.02.2022

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 764785, FORA—Fog Computing for Robotics and Industrial Automation [129].

# Publications

I have authored or co-authored the following publications:

## Journal Paper

- M. Kadar, G. Fohler, P. Gorski, and D. Kuzhiyelil. A survey of host intrusion detection for embedded mixed-criticality systems. submitted, 2021.

## Peer-Reviewed Conference and Workshop Papers

- M. Kadar, G. Fohler, D. Kuzhiyelil, and P. Gorski. Safety-aware integration of hardware-assisted program tracing in mixed-criticality systems for security monitoring. In IEEE Real-Time and Embedded Technology and Applications Symposium, 2021.
- M. Kadar, S. Tverdyshev, and G. Fohler. Towards host intrusion detection for embedded industrial systems. In 50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S), 2020.
- D. Kuzhiyelil, P. Zieris, M. Kadar, S. Tverdyshev, and G. Fohler. Towards transparent control-flow integrity in safety-critical systems. In 23rd Information Security Conference, 2020.
  - My contributions consist of the implementation of forward-edge control-flow integrity checking, the integration of control-flow integrity checks on the test platform, the timing overhead analysis, as well as parts of the results analysis and literature review.
- P. Denzler, J. Ruh, M. Kadar, C. Avasalcai, and W. Kastner. Towards consolidating industrial use cases on a common fog computing platform. In 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2020.
  - My contribution corresponds to all security-related considerations of industrial use-cases to build a common fog computing platform.

- M. Kadar, S. Tverdyshev, and G. Fohler. System calls instrumentation for intrusion detection in embedded mixed-criticality systems. In 4th International Workshop on Security and Dependability of Critical Embedded Real-Time Systems, 2019.

# Contents

# List of Figures

# List of Tables

I

# Introduction

## I.1 Motivation

Today, the industry of embedded systems tends to consolidating many software functions on a same hardware platform to save costs like hardware physical space, computing power, and cabling infrastructure. In particular in the domain of safety-critical applications, embedded Mixed-Criticality System (MCS) integrate several software components corresponding to different criticality levels on a single hardware platform. The criticality of a component depends on the type of safety-related constraints affecting its execution; e.g. soft, firm, or hard Real-Time (RT) deadlines. The failure of a high-critical software component can lead to catastrophic consequences.

In this context, safety and security pursue the same objective: the embedded MCS system must ensure the correct behavior of software components, in the presence of inner or outer effects that may lead to unintended execution paths altering the execution of one or several components. Safety and security have common ground in certain properties, such as the availability and the integrity of a system function. Goals of safety properties include protecting the systems from random faults caused by the environment: i.e. safety properties are not supposed to defend the system against attacks raised by a smart adversarial individual, who constantly adapts the attack to bypass the protection. Embedded MCS are subject to security threats because of low-critical applications. Low-critical software represent indeed attractive entry points for an attack, considering two main aspects:

- A low-critical software generally supports user interaction (i.e. through user software interface, via physical hardware access, or remotely leveraging networking capability).

- A low-critical software is more likely to contain software bugs compared to high-critical software components, which are developed, tested, and verified following rigorous certification processes.

Hence for a given software component, the probability of a successful attack decreases when the component's criticality level increases. Additionally, ScheduLeak threat [33] has recently demonstrated how an adversary controlling a low-critical task can disturb the execution of a higher-critical task via a side-channel attack on the real-time schedule. Thus, security is a major and realistic issue for systems with mixed-criticality requirements: **embedded MCS must support security mechanisms, aside from safety mechanisms, to prevent, detect, and mitigate the threats able to compromise system safety-critical functions.**

Host Intrusion Detection System (HIDS) represent a well-known domain of research for runtime threat detection in the execution of a given program. The state-of-the-art literature proposes diverse HIDS solutions, which vary with the analysis method (e.g. specification based methods or Machine Learning (ML) assisted) and in function of the type of monitored system events from hardware and software layers. The great majority of these approaches apply to General-Purpose (GP) computers. However, the integration of a solution into an embedded MCS requires additional effort for industrial deployment:

- The integration of a HIDS into an embedded MCS must cope with the hardware limitation of the embedded platform, while meeting the safety-related requirements of critical software components.

- As the development of embedded MCS follows a rigorous certification process for industrial deployment, the impact of integrating a HIDS into the system on the certification process must be evaluated beforehand.

**Hence, we need a set of basic properties and criteria to evaluate the suitability of a HIDS framework for deployment into industrial embedded MCS.**

The concept of MCS was mentioned for the first time after the introduction of partitioning in avionics system architecture with the concept of Integrated Modular Avionics (IMA): e.g. In 1999, Dutertre and Staviridou [52] have defined a model of safe integration of several independent software components having different criticality levels on a single hardware platform. Yet, despite the early focus on the non-interference requirement for integrating software into such systems [52, 69], the security focus – more specifically the topic of runtime intrusion detection – has come into light much later. For approximately a decade, several research works have introduced HIDS techniques for systems with criticality constraints; e.g. Control-Flow Integrity (CFI) monitoring [156, 127, 1]. Though, none of the works we reviewed addresses the problem of industrial deployment of intrusion detection in embedded MCS, i.e. especially involving certification standards to develop high-critical safety functions. The productization process imposes constraints on the HIDS approach to limit the monitoring impact on system properties at runtime (e.g. the response time of the monitored software) on one hand, the additional effort for certification when integrating HIDS mechanisms on the other hand. For example, an approach requiring instrumentation to monitor a high-critical program induces a significant impact on system certification: the certification process on the monitored program must be reiterated for every update of instrumentation based mechanisms. In addition, there is to our knowledge no prior research discussing how to integrate state-of-the-art HIDS mechanisms into an embedded MCS, even though there is a wide diversity of solutions, notably in function of types of traced system events and trace analysis methods, as well as in terms of evaluation approaches and criteria.

Developing HIDS for embedded MCS represents a major challenge with the diversity of these systems, in particular in terms of portability. Because of the wide variety of hardware and Operating System (OS) for embedded MCS, there is today no industry-standard platform. For example, ARM, PowerPC, and Intel are three common processor architectures. While we intend to leverage low-level system events for intrusion detection to limit monitoring impact on the monitored program, the tracing of these system events heavily depends on characteristics of the deployment platform: this potentially poses limits for HIDS portability. The diversity of platforms also induces the absence of widely available test benches to evaluate a given HIDS; the evaluation of a solution depends on the platform as well as on the monitored application use-case.

In addition, the porting of a HIDS framework built for GP computers to an industrial embedded MCS is not straight-forward. While the characteristics of the deployment platform are usually serious constraints for an embedded systems, system performances represent comparatively secondary considerations for GP computers (e.g. dematerialized execution of the HIDS in a Cloud infrastructure). A mixed-criticality execution environment brings safety critical constraints, like hard real-time deadlines, to be considered for HIDS deployment. Especially in the context of industrial deployment, the lower the intrusiveness level of HIDS mechanisms for monitoring, the lower the impact of the HIDS on system certification effort.

## I.2 Main Contributions

The work at hand contributes and extend the state-of-the-art HIDS research towards HIDS deployment in industrial embedded MCS. For this, we introduce HIDS solutions, which are able to run in environments with embedded and safety-critical constraints. A key objective is to identify traceable system events at runtime and analysis methods, which are relevant for intrusion detection in embedded MCS. We base our work on monitoring methods used for GP computers, applying them in the context of secure embedded MCS. We evaluate the impact of security solutions in a practical industrial environment, using a standard hardware platform and RT hypervisor.

The main contributions are as follows:

- **a comprehensive literature research for HIDS deployment into industrial embedded MCS** – In the survey [78], we identify suitable system events and trace analysis approaches for intrusion detection in embedded MCS. As a complementary focus, we describe security challenges for Industry 4.0 scenarios in [48].

- **a methodology to develop HIDS for embedded MCS** – We introduce a generic system architecture to integrate HIDS into an embedded MCS [81]. We also propose several intrusion detection approaches, as well as a set of evaluation methods, criteria, and metrics.

- **anomaly-based HIDS using heuristics for industrial embedded MCS** – We develop several machine-learning based HIDS approaches, leveraging a safety-aware System Call (SC) instrumentation based framework in [80] and transparent Hardware Performance Counters (HPC) monitoring. In the scope of this thesis, we investigated how to use online machine learning (ML) to detect intrusions: this led to a M.Sc. thesis publication [109].

- **a hardware-assisted CFI monitoring prototype for embedded MCS** – We introduced a safety-aware framework for CFI monitoring [92]. In [79], we extend this framework to configure the ratio between performance overhead and security; we also propose a transparent online service to monitor Control-Flow (CF) related traces.
- **a multi-mode HIDS approach for embedded MCS** – We discuss how to combine different intrusion detection approaches to help the configuration of adaptive HIDS; the main benefit being to adapt the trade-off between accurate intrusion detection, security coverage, performance overhead, and system intrusiveness for monitoring.

## I.3 Thesis Outline

This thesis is organized as follows.

**Chapter II** We introduce basic concepts related to HIDS and embedded MCS. After describing the security threat problem for these specific deployment platforms, we define a set of criteria to compare state-of-the-art HIDS framework later discussed. The goal is to define the ability of a given solution to protect a monitored application in an industrial embedded MCS, more particularly in the context of certification.

**Chapter III** We split our literature review in two main parts: while we provide an overview of HIDS in GP computers, we also discuss the state-of-the-art solutions applying to embedded MCS.

**Chapter IV** We describe an approach for integrating HIDS into embedded MCS. We also define a generic system architecture and identify a set of metrics to evaluate the HIDS solution with respect to security, system schedulability, and performance overhead. Based on this generic HIDS framework, we develop several detection approaches in the following chapters.

**Chapter V** We present ML based methods to detect intrusions in the execution of a monitored application. The goal is to limit intrusiveness in monitoring, by tracing low-level system events with system calls at OS level and HPC at hardware level. While the tracing of these events do not require instrumentation of the monitored program, the implementation depends on the system platform (OS and hardware layers). Based on this system call and HPC tracing infrastructure, we develop an open-source ML framework to perform detection locally on the deployment platform.

**Chapter VI** We introduce a configurable and predictable CFI monitoring framework, a specification based approach, which leverages hardware processor tracing to avoid instrumentation of the monitored program. In this chapter, we also introduce a CF events monitoring service using statistics to be combined with sporadic CFI checking, to address the trade-off issue between performance overhead and coverage of the monitored execution.

**Chapter VII** We present the approach to combine two monitoring solutions with an analytical adaptive HIDS. We define a set of requirements and evaluation considerations to guide future implementations. Our goal is to combine several detection approaches to improve the detection of intrusions in the monitored application; i.e. possibly decreasing the rate of detection errors, while reducing the heavy performance overhead induced by some intrusion detection methods like our CFI monitoring approach.

**Chapter VIII** We summarize and conclude this thesis by comparing our contributions to the state of the art. We discuss main limitations of our implementations and paths for future HIDS development in the context of industrial embedded MCS. Finally, we examine remaining open questions to address by future research.

II

# Background

Our goal is to deploy Host Intrusion Detection Systems (HIDS) into industrial embedded Mixed-Criticality Systems (MCS) to protect these systems against threats. It is essential to clarify several basic concepts beforehand, in particular to understand the inherent specificities and constraints of industrial embedded MCS. We explain the main concepts involved in HIDS deployment for embedded MCS in Section II.1. In Section II.2, we describe our work environment, related specific system constraints, and the scope of threats to consider, to finally define the problem which we propose to address through this work. After this contextualization, we specify the thesis goal in Section II.3.

## II.1 Basic Concepts

This section clarifies the essential concepts involved in the domains of MCS and intrusion detection based security. It compares the notions of safety and security (Section II.1.1), defines the specifities of MCS platforms (Section II.1.2, and introduces HIDS (Section II.1.3).

### II.1.1 Safety and Security

Even though both, safety and security, address the problem of system failure, these two approaches fundamentally differ in their context of deployment related to the nature of the adversary they consider.

#### From Fault to System Failure

We define a computer system as a set of subcomponents executing programmed tasks. Laprie clarifies the impairments to dependability [93] (Figure II.1). A **fault** is the result of a misbehavior of a system component; i.e. a software or hardware element. It can be provoked by a random event – physical phenomena such as short-circuits or electromagnetic perturbations – or a human action; i.e. an inadvertent or deliberate violation of operating or maintenance procedures [93]. An **error** is an activated fault that compromises the state of the system. A **failure** is an error, which makes the system behavior deviate from its expected normal execution and propagates to the system

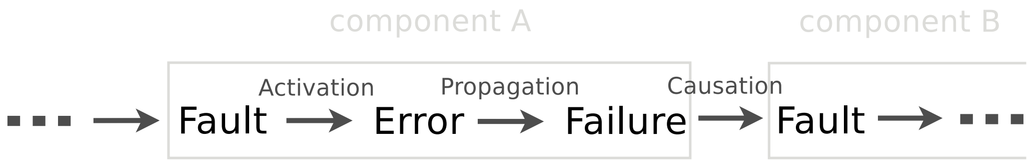


**Figure II.1:** *Error propagation chain of dependability, introduced by Avizienis et al. [21]*

interface. A fault, an error, or a failure possibly causes further faults propagation to other system components.

### Difference Between Safety and Security

**Safety** is a property applying to the whole system or a subset of system components. Its objective is to assure that no failure in the set of components can cause catastrophic consequences in the deployment environment; e.g. endangering, life of individuals, material damages, economical issues, etc. More specifically in this study, we consider **Functional safety**, a safety property which intends a system to operate predictably and deterministically, under normal execution as well as in case of failure. In this context, we perceive the physical environment as an adversary, which can induce system random faults. For example in computer systems, a bit flip in RAM is a common hardware fault, which arbitrarily sets the bit to an incorrect value at random times.

While the safety objective is to protect the system against the environment, a goal of **security** (which Avizienis et al. define as a set of attributes including confidentiality, integrity and availability [21]) is to protect the system against intrusions. We define an **intrusion** as an intentional malicious fault initiated by a hostile individual. Depending on the context, the intrusion can lead to further errors and failures in the system. While in terms of safety the adversary does not behave intelligently (generating random faults), from the security's perspective the adversary is smart: the individual continuously adapts its action to hijack existing protections. Hence, securing a system is a continuous iterative process to counter evolving adapting threats. In that sense, the security goal differs from the safety goal, which aims at minimizing the risk probability of random faults.

## II.1.2 Embedded Mixed-Criticality System

In the last decade, MCS have come into light in the context of Real-Time Systems (RTS) and have spread out in the industry of embedded systems. A MCS includes a set of functions associated with various significance levels in case of failure, introducing several criticality domains of execution. Embedded MCS combine the limitation of embedded systems in terms of hardware resources with MCS specific constraints; in particular the independence and freedom from interference between the distinct criticality domains.

### Embedded Mixed-Criticality System Definition

Following the definition from Heath [68], we define an embedded system as a «microprocessor based system that is built to control a function or a range of functions ». Embedded systems are generally designed to fit a specific use-case: in opposition to personal computers or cloud servers, they have limited hardware resources (i.e. connectivity, memory, I/O peripherals) and computing power. However, with the increase of hardware capacity in the last decades, virtualization and multi-core CPU support have spread among embedded platforms to save costs such as data storage, computing power, and power consumption. Consequently, the complexity of embedded systems has globally increased, as they aim at bringing different workloads on the same hardware platform. In that respect, MCS address the problem of consolidating several system functions on a same hardware platform

In their literature review [27], Burns and Davis provide a thorough state of the art of MCS research. They designate **criticality** as the level (or domain) of assurance against failure that is needed for a system component. In that respect, they define a MCS as a system including components from two or more (up to 5) distinct criticality levels. To meet the multiple safety and security requirements of the different criticality domains, MCS must support proper separation of the domains. **Hence, we define an embedded MCS as an embedded system including components from two or more distinct criticality levels.**

### Mixed-Criticality System Modeling

Basically, MCS modeling follows the general approach of RTS modeling; one difference being the consideration of tasks' criticality in the task and system models. In RTS research, a simple representation of a periodic system task $\tau_i$ is: $\tau_i = (C_i, D_i, T_i)$, with $C_i$ the worst-case execution time of the task (WCET), $D_i$ its deadline, and $T_i$ its period. For MCS modeling, we introduce a new parameter $L_i$ for the task $\tau_i$, which represents the criticality of the task; i.e. $\tau_i = (C_i, D_i, T_i, L_i)$.

To support MCS, the system design must integrate the new criticality parameter of tasks. The criticality parameter may also indirectly influence other task parameters (WCET, period, and deadline): e.g. the higher the criticality level, the more conservative the WCET is computed. Additionally, context switch between distinct criticality levels takes intuitively longer than switching between tasks of the same level as discussed by Davis et al. [46]. In a MCS composed of tasks of different criticality levels, the authors introduce processes to define distinct criticality domains, which correspond to a set of resources – cache and memory address space – and tasks of the same criticality level. The switch between two tasks of different processes (1) takes longer to execute than the switch between two tasks of the same process (2). The two tasks require stronger spatial isolation in Case (1) compared to Case (2); i.e. the switch between tasks must update the cache and memory address space of the tasks. With this approach, Davis et al. integrate the variation of context switch costs into schedulability analysis for several scheduling schemes.

### Multiple Criticality Domains Separation Problem

A key requirement of MCS is to achieve strong separation of the multiple criticality domains in the system: i.e. assuring that an application executing in a given criticality domain cannot affect the execution of other applications in domains with higher criticality levels. Because high-critical tasks share system resources with low-critical tasks, system interferences could cause damages like catastrophic failure of a high-critical safety function. The system should guarantee basic safety and liveness properties of safety-critical systems, such as mutual exclusion on shared resources, the absence of deadlock and the freedom from starvation [9, 122]. Such safety and liveness properties contribute both to system safety and security, preventing the system to reach – via intentional or hazardous transitions – a compromised state or failure.

In addition to these basic properties, a MCS specifically requires **sufficient independence** and **freedom from interference**: it must assure strong spatial and timing separation between components of distinct criticality levels. Blanquart et al. [25] define freedom from interference as the absence of cascading failure from low to higher criticality levels; i.e. a system component with a low criticality level cannot influence another component from a higher criticality domain. They define sufficient independence as the absence of common cause failures and cascading failures between components that could lead to the violation of the initial safety requirement. As for basic safety and liveness properties, these two additional properties enforce system safety and security, through a strong separation of the criticality domains at design level. In addition, runtime control mechanisms, such as as memory protection and admission control, can enhance the domains separation [117].

## II.1.3 Runtime Security via Host Intrusion Detection System

Information Technology (IT) security involves various defense bricks deployed at multiple enterprise levels (e.g. network traffic monitoring, enterprise processes, computing systems). In particular, Host Intrusion Detection is an approach to runtime threat detection in a computer system. Several criteria contribute to evaluate the quality of a HIDS solution in terms of security: the detection rapidity through the maximum time window of opportunity for an attack metric and the detection accuracy using false-positive and false-negatives related metrics.

### Host Intrusion Detection System Definition

A Host Intrusion Detection System (HIDS) is a runtime security monitoring service to detect threats occurring at runtime in a system. It identifies threats in a system's execution as deviations from the expected behavior of this system. After the detection of an intrusion, the HIDS possibly generates log traces or raises an alarm, so that further protection measures can be activated. For example in complex organization infrastructures, System Information and Event Management (SIEM) systems can use HIDS of multiple single devices for security monitoring. For example, Splunk [144] is a popular SIEM on the IT security market [83] SIEM systems watch the security of an infrastructure by analyzing traces from multiple sources, such as HIDS alerts, network, system, and application logs. They provide an interface for the human response team to interpret the observations and correlations to take actions for solving upcoming security threats; for instance, Cinque et al. [35] discuss the challenges for applying an open-source SIEM (OSSEC [120]) to a real-life critical infrastructure, using an industrial Air-Traffic Control platform. HIDS are to be distinguished from network intrusion detection systems (NIDS), which exclusively monitor and focus on network communications. While NIDS generally operate inside an internal network, a HIDS indeed remains at system level, inside a single device.

### Overview of Host Intrusion Detection Approaches

There is a wide diversity of HIDS, depending on traced system events and analysis techniques. Section III.2 provides a focus on frameworks suitable for our specific context of work – industrial embedded mixed-criticality systems. Host intrusion detection can apply at different levels: to protect the execution of a single application, a software component, or the full operating system. As later developed in Section II.2.2, we focus on solutions monitoring a user-level application. We split the intrusion detection runtime process into two main runtime phases (Figure II.2):

- **trace collection**: this stage corresponds to observing the monitored software through its execution footprint on the system (i.e. hardware and software system events). Table III.3 provides an overview of trace types which can be collected in common computer systems.

- **trace analysis**: this stage infers the execution of a threat in the monitored software through the identification of malicious traces. The HIDS performs the identification using a policy that defines benign and malicious execution traces. Example of policies are a statistical model of execution, set of authorized actions, list of malicious trace patterns, etc.

We categorize three main data analysis approaches for intrusion detection [91, 76]. On one hand, **signature based** analysis uses a database of signatures of known attacks to detect these same attacks when they occur at system runtime. Such technique generally comes with a good detection performance for the predefined attack scope. However, they offer poor protection against zero-days attacks, which are attacks unknown at the time of building the analysis model. On the other hand, **specification based** and **anomaly-detection based** HIDS techniques define an intrusion as an anomaly which leads to an intentional malicious fault in the system; n.b. an anomaly does not systematically correspond to an intrusion (an abnormal system behavior that has no impact on security). Such HIDS solutions set up a profile (also called model) of normal software behavior; e.g. early intrusion detection expert system (IDES) [47] prototypes leverage statistical profiles of system events. They monitor the execution by observing system traces of software behavior and detect an intrusion when the runtime behavior deviates from its corresponding profile. Hence, both approaches are suitable to detect zero-day attacks.

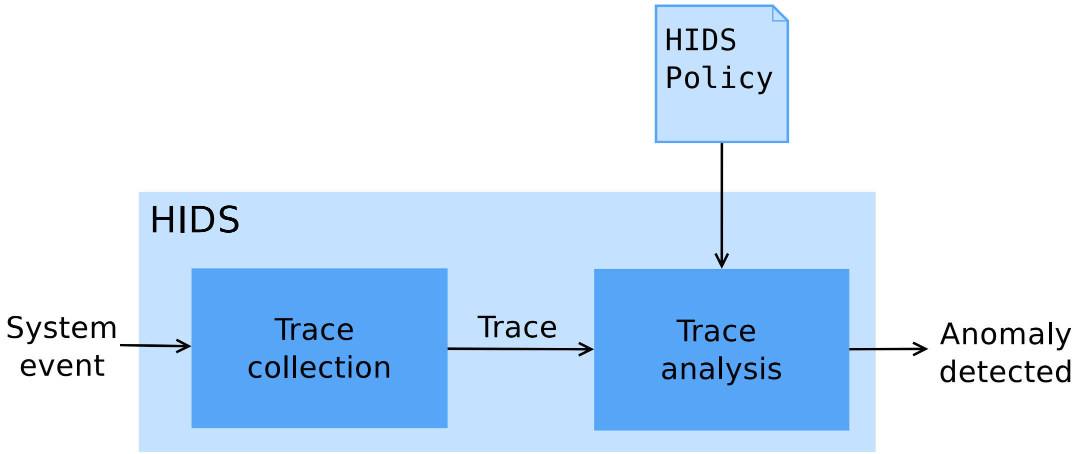


**Figure II.2:** *Host intrusion detection system overview*

Specification based solutions define a set of rules representing benign execution; i.e. a set of authorized states and transitions. These approaches use knowledge of the program to monitor (e.g. sets of legitimate memory access, authorized operations, system interactions, etc.) to generate the model of benign execution. Hence, this analysis does not generate any false-positive detection. Ideally a perfect representation of benign execution infers a null rate of false-negatives (all actual threats are accurately detected). In practice, depending on the precision of the model describing normal execution, some attacks can bypass the detection. In contrast with specification based methods, anomaly-detection based framework derive the model of benign execution from the observation of the program running under normal and anomalous conditions, leveraging system events such as system calls and logs. Such analysis can use machine-learning or statistical based methods to generate the execution model. Security detection results (false-positive and false-negative rates) depend on the model representation.

According to our later defined problem definition (Section II.2.2), we consider HIDS which are able to detect zero-day attacks. Hence, we focus in this work specifically on specification and anomaly based detection; signature based detectors represent additional complementary security bricks to improve the overall system security.

### Host Intrusion Detection System Evaluation

For many HIDS, Machine Learning based solutions among others, the profile of normal software execution is determined offline prior to deployment, using a predefined set of normal execution traces. We can then test the profile at runtime, for normal and anomalous executions. The evaluation of a classification model in terms of detection performance can involve several metrics. For example, Table II.1 shows the confusion matrix, a basic metric corresponding to a table layout to visualize correct and incorrect predictions made by the model. The confusion matrix leverages the following indicators: false-positives (FP), false-negatives (FN), true-positives (TP) and true-negatives (TN). The objective of a good detection solution is to minimize $r_{FP}$ the False-Positive Rate (FPR) and $r_{FN}$ the False-Negative Rate (FNR) – corresponding respectively to the total

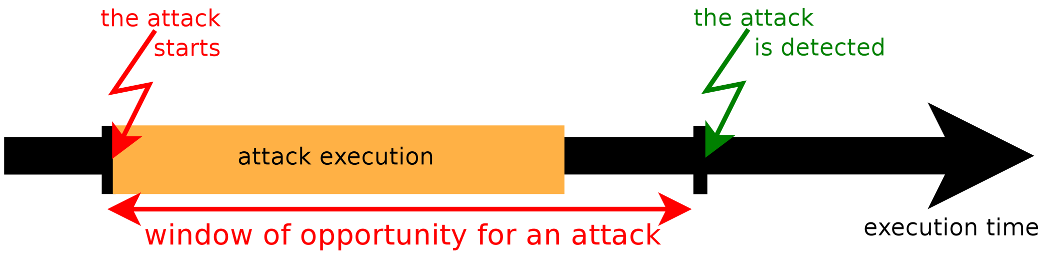


**Figure II.3:** *Window of opportunity for an attack*

of actual normal samples that the solution detects as anomalous and to the total of actual anomalous samples that the solution detects as normal. In this work, we introduce the detection precision $P$, recall $R$, and F1-score $S$ metrics as follows:

$$\begin{aligned} r_{FP} &= \tfrac{FP}{TP+FP} \\ r_{FN} &= \tfrac{FN}{TP+FN} \\ P &= \tfrac{TP}{TP+FP} = 1 - r_{FP} \\ R &= \tfrac{TP}{TP+FN} = 1 - r_{FN} \\ S &= 2 * \tfrac{P*R}{P+R} \end{aligned} \tag{II.1}$$

The rapidity of the detection represents another key aspect of runtime intrusion detection: the shorter the detection latency, the less time an adversary has to perform an attack before being detected. Hence, we introduce an additional evaluation metric, the **window of opportunity for an attack**, as the time between the moment the anomaly starts and until the HIDS has raised an alarm (Figure II.3).

**Table II.1:** *Confusion matrix*

| | | **Predicted** | |
|---|---|---|---|
| | | **Positive** | **Negative** |
| **Actual** | **Positive** | TP | FN |
| | **Negative** | FP | TN |

## II.2 Security Problem for Embedded Mixed-Criticality Systems

In this section, we leverage the concepts previously introduced in Section II.1 to highlight the specificities for deploying industrial embedded MCS (Section II.2.1) and characterize the type of system intrusions threatening such systems (Section II.2.2).

### II.2.1 Industrial Embedded Mixed-Criticality Systems Requirements for Deployment

Our work targets runtime security deployment into industrial embedded MCS systems, possibly involving high-critical safety constraints. On the one hand, such environment

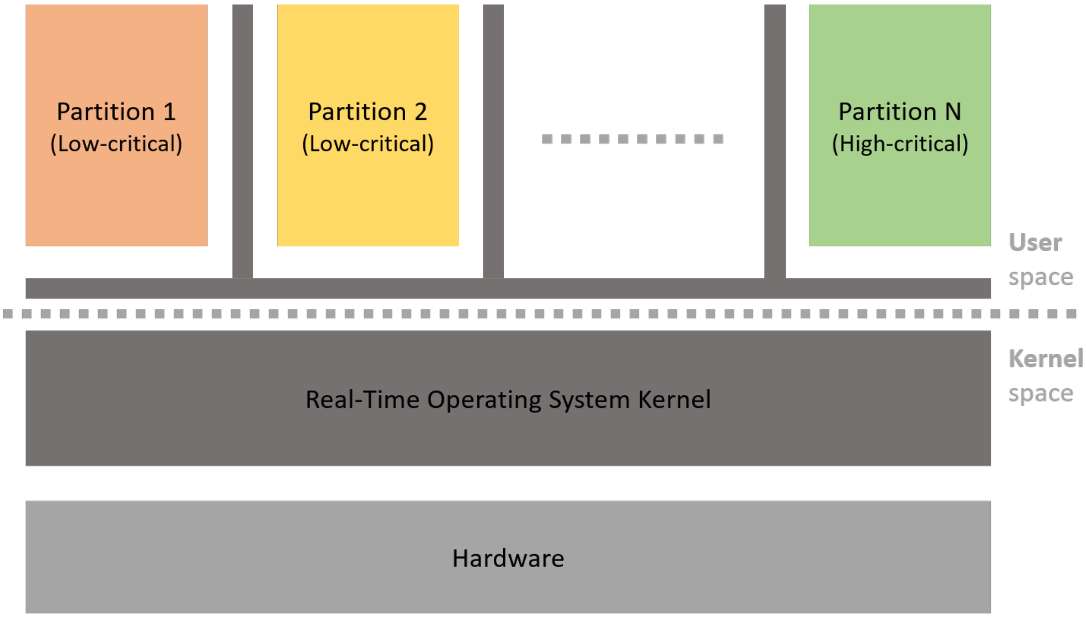


**Figure II.4:** *A generic embedded mixed-criticality system architecture*

can induce catastrophic consequences in case of system failure; taking the role of the system integrator, we must assure that all criticality requirements are met. On the other hand, this work environment brings additional constraints: e.g. reduced hardware resources of embedded systems compared to generic computers and limited access to application software.

### Embedded Mixed-Criticality Systems

Figure II.4 shows the architecture of a generic embedded MCS. As for general-purpose computers, we can split this architecture in 3 main system levels: system hardware level with physical resources, operating system kernel level for privileged software operations, and user-level for remaining software activity. Compared to GP computers, embedded systems usually come with limited hardware resources because of the cost constraints: reduced device weight, power consumption, memory capacity, computation power. This problem is even more relevant for embedded systems which integrate safety-critical functions. For example with avionics, the weight of devices integrated into an aircraft is a determinant factor to reduce; e.g. by limiting hardware space and computation power [41]. Embedded mixed-criticality systems leverage virtualization to cope with the limitation of resources for improving resource usage via sharing. For instance in aircraft systems, the integrated modular avionics (IMA) architecture has replaced the traditional federated architectures [160].

The system must support software separation in user space to avoid illegitimate interference between user-level components (e.g. cache based side-channels threats). Thus, we introduce the concept of partitioning, defining a partition as a domain of execution in user space to execute a given software component; this execution domain assures that the software component executes without illegitimate interferences with other partitions. Specifically, partitioning consists of explicitly defining the set of system resources (e.g. memory, I/O peripherals, CPU cores, etc.) each partition can access, and when (scheduling policy). As an example, Figure II.4 shows a generic MCS composed of user-level partitions, each containing an application corresponding to a certain level of criticality.

Hence, to assure a proper system operation complying with execution constraints such as real-time deadlines, the system must guarantee the freedom from interference and sufficient independence between system tasks. A partitioned architecture can assure the separation of user-level software. The Multiple Independent Layers of Security (MILS) architecture [10, 152, 55] is one direct application. It implements both hardware and software separation mechanisms to isolate the defined criticality domains, as well as secure communication channels to control the information flow transiting between the different domains. It embeds the notion of Separation Kernel (SK) [137], an execution domain composed of all system software running privilege operations.

### Certification and Compliance for Industrial Deployment of Host Intrusion Detection Systems

An industrial embedded mixed-criticality system typically includes software components from different criticality levels. A recent survey by Akesson et al. [8] about industrial practice in real-time systems development and deployment analyzes the inputs from 120 practitioners, mainly from the Automotive and Avionics domains. It shows that many industrial systems integrate distinct types of time constraints (i.e. hard, firm, and soft). Therefore, considering that the level of task time constraints and criticality are correlated, modern industrial systems include various criticality levels.

Common system architecture standards such as AUTomotive Open System ARchitecture (AUTOSAR) [19] for the Automotive and Avionics Application Standard Software Interface (ARINC 653) [12] for the Avionics provide mixed-criticality support. Criticality is determined through diverse industrial standards depending on the deployment context: e.g. Automotive Safety Integrity Levels (ASIL) risk classification scheme of ISO-26262 standard [75] for functional safety of automotive systems, Design Assurance Level (DAL) given by DO-178C standard [135] to certify aircraft systems. The MCS must satisfy constraints requirements and objectives imposed by the targeted standard, assigning a dedicated assurance level for each of its system components. Four main factors determine the criticality of a component:

- the consequence of a failure of the component
- the probability for a failure of the component to occur
- the exposure of the component to failure
- the mitigation means at hand to resolve a potential failure

Nowadays, very few platforms are entirely developed by a single company from the hardware circuit to the high-level software functionalities: modern OS support standard hardware platforms (i.e. standard CPU architecture like ARM, Intel, PowerPC), while various companies provide specific functionalities as hardware or software components of the shelf (COTS): e.g. connectivity, custom hardware driver, legacy functionality, etc. Hence, many actors participate to the product life-cycle. We simplify the main roles involved in the system design as follows:

- The **hardware provider** supplies the hardware platform. A recent survey [8] highlights the hardware complexity of industrial critical systems: i.e. multi-core components, memory hierarchy including several cache levels, connectivity support.
- Various **application providers** provide user-level software, from the non-critical infotainment application to the high-critical actuator control, typically compliant to standardized architecture and application interfaces such as AUTOSAR for automotive applications and ARINC 653 for avionics applications.
- The **software platform provider** develops software to allocate hardware resources to user-level software (e.g. providing OS, runtime environment, and device drivers).
- The **system integrator** is our main focus in this paper. The system integrator gathers the hardware and software blocks provided by the previously described agents. This person is responsible for complying with the system requirements such as time deadlines. The survey [8] shows that timing predictability is one of the major concerns, together with system safety, functional correctness, as well as reliability and availability (which are two dependability properties).

The RTOS must support the key properties of systems with real-time constraints: freedom from interferences and sufficient independence (Section II.1.2). Hence, a strict separation of time and hardware resources is necessary in this context. Partitioning is a way to assure the separation of user-level software in isolated partitions, by controlling the access to shared resources and inter-partition communications. For example, SYSGO's industrial hypervisor PikeOS [128] implements time and hardware resource partitioning so that it is able to integrate multiple partitions certified at different criticality levels on one hardware platform.

Besides the certification requirements, the HIDS needs to comply with specific industry standards for the application interface and architecture. The automotive industry has been forefront in defining a specification for automotive IDS. AUTOSAR Release R20-11 [18] defines the onboard architecture of a distributed HIDS, security events for selected base software (BSW) modules, interfaces for reporting security events, and communication between different modules in the distributed IDS. The standard allows any ECU to act as security sensors and develop custom analysis methods, as long as they comply with the standard.

#### Industrial Embedded Mixed-Criticality System Constraints

Performing host intrusion detection at runtime in an industrial embedded MCS poses several constraints. At runtime, the monitoring solution must comply with the limited hardware resources, while satisfying the real-time constraints of critical tasks. The system should indeed assure strict separation of time and hardware resources. The monitor software especially, must be isolated from the monitored software to prevent attackers to access it and influence the detection.

As we adopt the role of the system integrator, transparency is a key aspect to consider for designing HIDS. Ideally, the solution must be transparent for the monitored software; HIDS mechanisms would observe the software from its interface to limit the intrusiveness. Industrial safety-critical programs are carefully developed following certification standards before being provided to the system integrator; notably for automotive certification, ISO-26262 introduces a process to develop a safety-element out of context [164]. Thus, the system integrator most probably cannot modify the programs without reiterating the certification process. Furthermore, these third-part software can also be legacy programs: in such case, the system integrator receives a black box (e.g. binary file); she has no knowledge on the internal implementation of the software components and has most probably no modification ability. Hence, the HIDS intrusiveness is a key criteria to evaluate the suitability of a solution for deployment in embedded MCS. Therefore in this context, we suggest to avoid monitoring approaches requiring instrumentation of the monitored software. Additionally, before deploying a HIDS framework into an embedded MCS, we must consider the related efforts required for full or partial re-certification of the system.

In addition to its impact on the overall system certification, the HIDS is also subject to certification. For example while monitoring a certified safety-critical program, the detection of a threat could be used to further apply reactive measures modifying the execution of the Monitored application (Monitoree). Hence depending on the configuration, we assign an assurance level to HIDS mechanisms, which must satisfy the constraints, requirements, and objectives imposed by the targeted standard: HIDS design and development must follow the standard's processes (e.g. providing specification and test documentation of HIDS functionality).

## II.2.2 Threat Environment

We define the asset of this work as the integrity of the execution of a user-level program to protect. Depending on the system's exposure to threats (e.g. user access, networking capability), different adversary profiles and attack scenarios are likely to attempt at compromising the asset. We consider two main scenarios of intrusion initiated from a user-level application: the malicious misuse of a benign program, which contains a security vulnerability exploited by the adversary at runtime, and the execution of a program dissimulating malware activity provided by a malicious supplier.

### Adversary Profile

The profile of the adversary varies depending on the system environment. For business applications (e.g. avionics or automotive domains), adversaries can be malicious software providers; i.e. organizations potentially with high financial resources, equipment, and skills. On the consumer market, a further adversary could be the customer (i.e. a system user) who has direct access to the physical system. Such adversary is distinct from the system owner: e.g. a car user faking wrong system configuration (modifying mileage, hiding warning signals, ...), opposed to the car vendor whose goal is to guarantee the integrity of the configuration. In this context, the malicious user likely comes with low or basic knowledge on the system, for example applying malware from the darknet. With connectivity support, a system becomes more accessible from the outside world, i.e. getting exposed to many agents through the network, including adversaries with sophisticated equipment and advanced skills.

Attacks can result from various motivations. Hamad et al. [63] describe possible motivations for the automotive domain. An adversary could target illegal profit by selling the attack further to malicious agents or deploying ransomware threats; it could also be a commercial competitor with the ambition to earn market shares. Alternative motives include vandalism (e.g. produced by a former employee seeking revenge) and research for test purpose. In this second case, the goal is to outpace potential attackers, by identifying new threats and taking countermeasures before attackers can actually exploit them.

A successful attack results in diverse effects in function of the context. A direct attack stops after it has affected a direct target: it does not propagate further in the system. Conversely, a multi-stage attack propagates after disturbing the first target to other system components. For instance, ScheduLeak [33] threat can be the first stage of a multi-stage attack in MCS. The attacker task first gets the exact time information of sensitive tasks execution. In a second stage, it performs further intrusion to compromise the sensitive tasks: e.g. by overriding system control signals (or applying cache side-channel attacks) just before or just after the victim task has executed.

The access to the victim device conditions the practicability of vulnerability exploits. With a direct access to the physical device, an adversary can achieves an intrusion plugging in malicious portable devices like USB sticks [126]. Side-channel attacks like electromagnetic emission based observation [130] require both physical access to the device and software interaction to leak confidential data. Other threats do not require physical manipulation of the system, occurring from software user interface through the network: i.e. software bug exploits.

### Threat Model

As explained in Section II.1.1, an attack is an attempt to compromise the asset valued by the system owner. In this work, the asset corresponds to **the integrity of the execution of the user-level program to protect**; we consider intrusions occurring inside a user-level software, that alter runtime behavior of the victim program. Typical threats are memory corruption exploits like return oriented programming (ROP) attacks, processor vulnerability exploits such as Spectre [87], denial of service, etc.

Because of system connectivity, the network is a potential entry point for an intrusion. An intrusion can also result from a physical attack such as malicious memory bit flips, or software memory corruption. In such case, the threat infers a malicious deviation of the runtime execution of the victim program to be protected.

Passive threats such as side-channels attacks, which do not interact with system software are out of scope for this work. Network based intrusions (distributed denial of service, spoofing, etc.) represent a specific area of research. Therefore, we do not consider them directly; we instead focus on more generic attacks targeting system execution.

For this work, we make the following assumptions:

- In MCS, because we assume the OS to be certified at highest integrity levels following rigorous development and validation standards, we consider the hardware and OS layers as free from malware: i.e. no hardware IP nor OS routine are running with malicious intentions during system execution.
- Hardware and OS layers may have unknown flaws to be exploited by an adversary, like cache interference leveraged in side-channel based attacks to leak confidential data. Because of system certification however, we deem this risk as highly unlikely. Additionally, we assume that mechanisms are in place to perform software and firmware updates when new flaws are discovered in these system layers. Thus, hardware and OS layers are trusted: they indeed must be certified to the highest assurance level which needs to be supported for the set of applications.
- Since user-level programs are provided by multiple external agents, with fluctuating certification levels, we do not trust them. In particular, low and no critical programs represent attractive targets for an adversary, since they probably contain more hidden flaws – compared to high-critical tasks – to be exploited for achieving system intrusions; they indeed follow less strict development process and traceability than critical certified software.

Thus, we model the adversary as a **non-trusted user-level program**. We identify two main attack scenarios:

- **Program misuse**: the user-level program has not been implemented to execute malicious activity. Though, it may contain flaws to be exploited at runtime by the attacker to perform malicious actions. For example, the adversary could send malicious inputs from system interface (i.e. inter-partition communication or network access) for the program to misbehave, provoking disastrous consequences; e.g. denial of service attack resulting in missed hard deadline, ROP threat exploiting a buffer overflow to perform privilege escalation.
- **Compromised program binary execution**: the user-level program binary provided to system integration is malicious: the application provider has designed it to execute – and possibly hide – malware activity at runtime. For example, such program could be implemented so that upon specific interaction with the adversary (e.g. via known program inputs) it triggers malware execution: hiding malicious activity, leaking information, modifying system configuration, etc.

### II.2.3 Problem Definition

Our security asset corresponds to the runtime integrity of the execution of a user-level program running in an industrial embedded MCS. We intend to protect this program

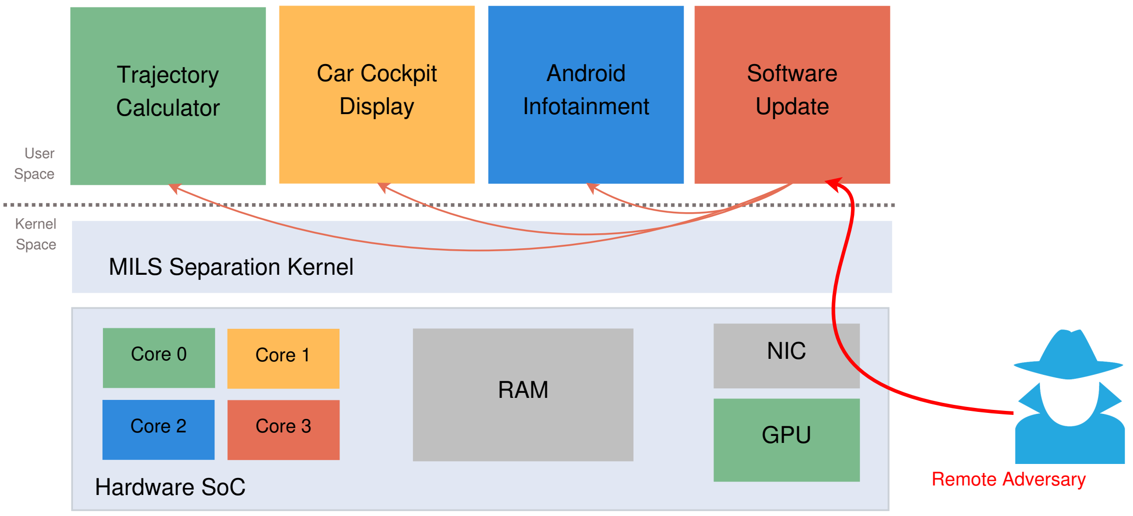


**Figure II.5:** *An example of system under threat*

leveraging host intrusion detection based monitoring solution. On the one hand the security module must be unreachable by the adversary. On the other hand, it must be transparent to comply with the inherent industrial constraints of the system: limited access to the program source code and restricted modification means of the program binary.

### System Model

For this study, we consider an industrial embedded MCS. Since we take the role of OS provider and system integrator, we assume that the OS is trusted; it is an industrial – potentially certified – RTOS which supports a separation kernel to isolate applications in space and time. We expect the RTOS to assure strong separation of user-level software components and isolation of the separation kernel from other untrusted components.

Figure II.5 provides an example of automotive system architecture. The hardware is a generic embedded platform with limited resources and networking ability. In the example, the system runs real-time critical user-level applications such as the car cockpit display and GPS trajectory calculation software, as well as a security critical application for software update from the network. It also includes a non-critical Android guest OS for in-vehicle infotainment.

### System Asset

We consider a system including several untrusted user-level applications provided by external agents. We define the asset as the **integrity of the execution of a user-level program in the system at runtime**. We call monitoree the user-level program to protect. This program can be a complex workload such as guest OS, as well as real-time critical software. It is a black box for the monitoring application: as system integrator, we potentially have no access to the source code and no mean to modify it. Alternatively, as described in the threat model, the monitoree could be a malicious binary; a malicious operator replacing the binary before deployment or by compromising a software update operation.

### Monitoring Model

The monitoring functionality is a trusted component separated from the monitoree using the SK mechanisms, so that it cannot be accessed by an adversary. Because of the inherent constraints of MCS and black-box monitoree, the monitoring component observes the system through the OS and hardware interfaces used by the monitoree: e.g. hardware interface using hardware performance counters, OS services with system call tracing.

## II.3 Thesis Goal

For this thesis, our goal is to address the problem of deploying security monitoring into industrial embedded MCS, protecting the execution of a user-level application (which we call monitoree). We define a threat as a deviation of the monitoree's execution from its expected behavior. We consider two main attack scenarios: the misuse of the monitoree, for example via wrong interface stimulation or memory corruption exploit, and the execution of a malicious binary dissimulating malware activity.

This work focuses on HIDS for runtime detection of intrusions in the execution of a monitored user-level application. We intend to provide a "security for safety" solution. Because of the environment constraints, the HIDS must induce limited intrusiveness to comply with certification processes for industrial deployments; i.e. no modification of the monitored program and potentially limited access to information in the monitoree binary.

III

# State of the Art on Host Intrusion Detection

We are not aware of any extensive literature review, which compares state-of-the-art solutions applying to embedded MCS. Even though, for the great majority of solutions, the system certification aspects are out of scope, they are essential from an industrial perspective to deploy the solution into production. Table III.1 compares this work with other surveys covering HIDS research, with the following criteria:

- **system:** the types of systems considered for HIDS deployment covered by our literature review.
  - **MCS** (also called cyber-physical systems in literature): systems supporting safety-critical tasks and real-time constraints; our MCS scope also covers real-time systems.
  - **embedded systems:** systems with limited resources such as hardware functionality, computing power, memory, etc.
  - **GP computers:** any other computer systems.
- **methods:** the types of HIDS methods covered by our literature review, with a focus on specification and ML based approaches.
- **data:** the types of system events involved in HIDS monitoring, which are covered by our literature review: we especially focus the comparison on hardware events (such as hardware counters and processor tracing based data) and system call data types.
- **certifiability:** our literature review evaluates the HIDS with regard to system certification towards industrial deployment.

As described on Table III.1, several surveys provide an overview on state-of-the-art host intrusion detection solutions: e.g. the survey by De Clercq and Verbauwhede about hardware based CFI [36], surveys about ML based intrusion detection [32, 76, 20]. To our knowledge, our literature review is the first to address the deployment of HIDS into industrial embedded MCS.

In this chapter, we provide a holistic overview on the state-of-the-art approaches for hardware, software, and hybrid based HIDS solutions towards deployment into industrial

**Table III.1:** *Comparison of surveys on host intrusion detection systems in function of explicitely covered research areas. (×, ●) symbols respectively correspond to unsupported and supported fields.*

| **Survey** | | **Systems** | | | **Methods** | | **Data** | | |
|---|---|---|---|---|---|---|---|---|---|
| | | GP computers | MCS | Embedded systems | Specification based | ML based | OS API (system calls) | Hardware events | Certifiability |
| Othman et al. | [121] | ● | × | × | × | ● | ● | × | × |
| Bridges et al. | [26] | ● | × | × | × | ● | ● | × | × |
| Sandhu et al. | [139] | ● | × | × | × | ● | ● | × | × |
| Rudd et al. | [136] | ● | × | × | ● | ● | ● | ● | × |
| Liu et al. | [103] | ● | × | ● | × | ● | ● | × | × |
| Zuech et al. | [167] | ● | × | × | × | ● | × | × | × |
| Khraisat et al. | [84] | ● | × | × | ● | ● | ● | × | × |
| Liao et al. | [98] | ● | × | ● | ● | ● | ● | ● | × |
| Axelsson et al. | [22] | ● | × | × | ● | × | ● | × | × |
| De Clercq and Verbauwhede | [36] | ● | × | ● | ● | × | × | ● | × |
| Sayeed et al. | [140] | ● | × | × | ● | × | × | ● | × |
| Elrawy et al. | [54] | ● | × | ● | × | ● | × | × | × |
| Han et al. | [64] | × | ● | × | × | ● | × | × | × |
| Mitchell and Chen | [113] | ● | ● | ● | ● | ● | × | × | × |
| This literature review | [78] | ● | ● | ● | ● | ● | ● | ● | ● |

embedded MCS. Our motivation is to bridge the gap between research and industry, helping future HIDS development to address the intrusion detection problem in an industrial context involving system certification. We describe relevant HIDS methods for GP computers and discuss their applicability to protect embedded MCS in an industrial context. We also compare HIDS solutions specifically designed for systems with real-time constraints. After defining key concepts to guide our literature review, we introduce a set of criteria and properties to compare and evaluate the cost for integrating the HIDS solutions into industrial embedded MCS.

This chapter is organized as follows. Using the basic concepts described in Section II.1, we describe our evaluation scheme to compare HIDS solutions in Section III.1. We split our literature review in two main parts: while Section III.2 provides an overview of HIDS in GP computers, Section III.3 focuses on state-of-the-art solutions applying to embedded MCS. Finally, we conclude the chapter in Section III.4, describing the contributions of this thesis.

# III.1 Evaluation Scheme to Guide our Literature Review on Host Intrusion Detection

We propose a classification to evaluate and compare the HIDS solutions discussed in our literature review (Section III.1.1); we define a set of metrics to assist the review. Section III.1.2 outlines the structure of our study and provides an overview of the reviewed solutions.

## III.1.1 Host Intrusion Detection System Classification

**Table III.2:** *Evaluation criteria base for our literature review*

| Criteria | Description | Metrics | Possible Values |
|---|---|---|---|
| Transparency | ability of the HIDS to trace the monitored application with limited intrusiveness | 1. application instrumentation<br>2. OS kernel modification | ✕, –<br>✕, – |
| Portability | ability of the HIDS to be deployed on different system platforms | 3. custom hardware<br>4. other hardware dependencies | ✕, –<br>✕, – |
| Security | ability of the HIDS to detect intrusions rapidly and accurately | 5. no false-positives<br>6. type of security evaluation<br>7. intrusion detection latency evaluation | ✕, ●<br>✕, ❍, ◗, ●<br>✕, ● |
| Certifiability | ability of the HIDS to run into a certified environment | 8. effort level for integrating the solution into a certified system<br>9. effort level to certify the HIDS | ✕, ❍, ◗, ●<br>✕, ❍, ◗, ● |
| Performance | ability of the HIDS to provide an acceptable trade-off between security impact in terms of system overhead and schedulability | 10. evaluation of runtime overhead caused by the HIDS | ✕, ● |

On the grounds of the deployment requirements described previously in this section, our goal is to determine whether a given HIDS solution:

- is suitable to be deployed in an embedded mixed-criticality system: upper-bounding the interference caused by the HIDS on the monitoree and coping with typical embedded system constraints (i.e. limited hardware resources).
- requires limited effort for integration into a certified systems: it can be certified to detect intrusions in a certified user-level application.
- can easily be ported on different embedded platforms: i.e. with limited performance impact and hardware dependencies.

Therefore, we define in Table III.2 five main criteria to base our literature review: transparency, portability, security, certifiability, and performance. The table provides a description of the criteria and corresponding metrics for our evaluation. For transparency and portability criteria, the metrics (1 to 4) correspond to the presence (✕) and absence (−) of constraining requirements for the HIDS framework. For security, certifiability, and performance evaluation criteria, each metric (5 to 10) defines whether a feature is supported (●) or not (✕). More specifically for metrics (6, 8, 9), we define further levels of support: none (✕), basic (○), incomplete (◗), and full (●). We provide a definition of the possible values below.

We compare the reviewed solutions in function of implementation constraints for deployment in embedded mixed-criticality systems through transparency and portability criteria.

- **transparency:** this depends mainly on the type of data to trace. For example, system call instrumentation requires OS kernel modification, while CF tracing may induce application instrumentation.
- **portability:** like for transparency criteria, the portability depends on the tracing approach; i.e. hardware implementation (e.g. with CF tracing to limit tracing intrusiveness) or hardware-assisted solution (e.g. processor tracing based monitoring). These dependencies directly affect the portability of a HIDS framework for deployment on other hardware platforms.

However, because of the great diversity of discussed frameworks (CFI monitoring, machine-learning assisted detection, etc.), implementation, and evaluation approaches (simple proof of concept, analytical security evaluation, test on diverse realistic complex threats, WCET analysis, etc.), we cannot directly compare the evaluation results. For example, standard false-positive and false-negative rates metrics used to evaluate a machine-learning based solution are generally irrelevant in the context of CFI monitoring; such metrics cannot be directly compared for different test configurations. Similarly, the detection latency can be evaluated through average measurements on a given test set or computing a worst-case analytical value. We instead propose a set of features to be supported by HIDS in our deployment environment. To evaluate the relevance of a framework for our deployment target, we follow 3 axes:

- **HIDS security evaluation:** we consider the presence of false-positive and detection latency evaluation as key metrics regarding security; in addition, we define 4 levels of security evaluation.
  - ✕: no security evaluation
  - ❍: basic evaluation with simple proof of concept
  - ◗: limited security metric measurements
  - ●: exhaustive security checks, through theoretical analysis or in a real-life complex environment
- **Performance impact for running the HIDS:** the impact for running an HIDS framework must be properly evaluated (e.g. time overhead, hardware cost) towards industrial deployment, especially in the context of embedded systems.
- **Certifiability of the solution:** we distinguish two main aspects for integrating a HIDS framework into a certified system. The corresponding metric values are inferred from other binary metrics: instrumentation of the monitoree program ($PI$), OS kernel modification ($KM$), solution designed for RTS ($RT$), and presence of false-positive in the detection ($FP$).
  - **HIDS integration impact on system certification:** we consider solutions requiring program instrumentation difficult to certify, since the certification process needs to apply to each targeted user-level application; as developed in Section II.2.3, in our position of system integrator, we may have no access to the application source code nor ability to modify the binary. Especially works which do not cover deployment in safety-critical systems, do not evaluate the impact of the HIDS on the time properties of the monitored application. Required kernel modifications induce in comparison a single effort in system certification, since they are independent from the monitored user-level program. Hence, we provide 4 effort levels for system certification:
    - **impractical** (✕): $PI$ and $\overline{RT}$
    - **heavy** (❍): $\overline{PI}$ and $\overline{RT}$ and $KM$
    - **moderate** (◗): $PI$ and $RT$
    - **low** (●): $\overline{PI}$ and ($RT$ or $\overline{KM}$)
  - **HIDS module certifiability:** towards active monitoring – i.e. in opposition to passive monitoring through system observation and logs stocking – the HIDS detection information must be reliable enough to impact further critical decisions such as stopping the monitored application after the detection of an intrusion. Notably in such context, a suitable detection solution should come with a guarantee to generate no false-positives.

    Solutions raising false-positives, and more specifically ones leveraging machine learning, cannot be certified at a current stateFrameworks requiring program instrumentation are complicated to certify, for the same reason developed in HIDS integration impact on system certification. In addition, with HIDS designed for generic computers, research works do not analyze the timing impact of the HIDS activity on the monitoree program. Thus, we evaluate the effort to certify the HIDS with 4 levels:

- **not certifiable** (✗): $FP$
- **heavy** (❍): $\overline{FP}$ and $\overline{RT}$ and $PI$
- **moderate** (◗): $\overline{FP}$ and $((\overline{RT}$ and $\overline{PI})$ or $(RT$ and $PI))$
- **low** (●): $\overline{FP}$ and $RT$ and $\overline{PI}$

### III.1.2 Literature Review Focus

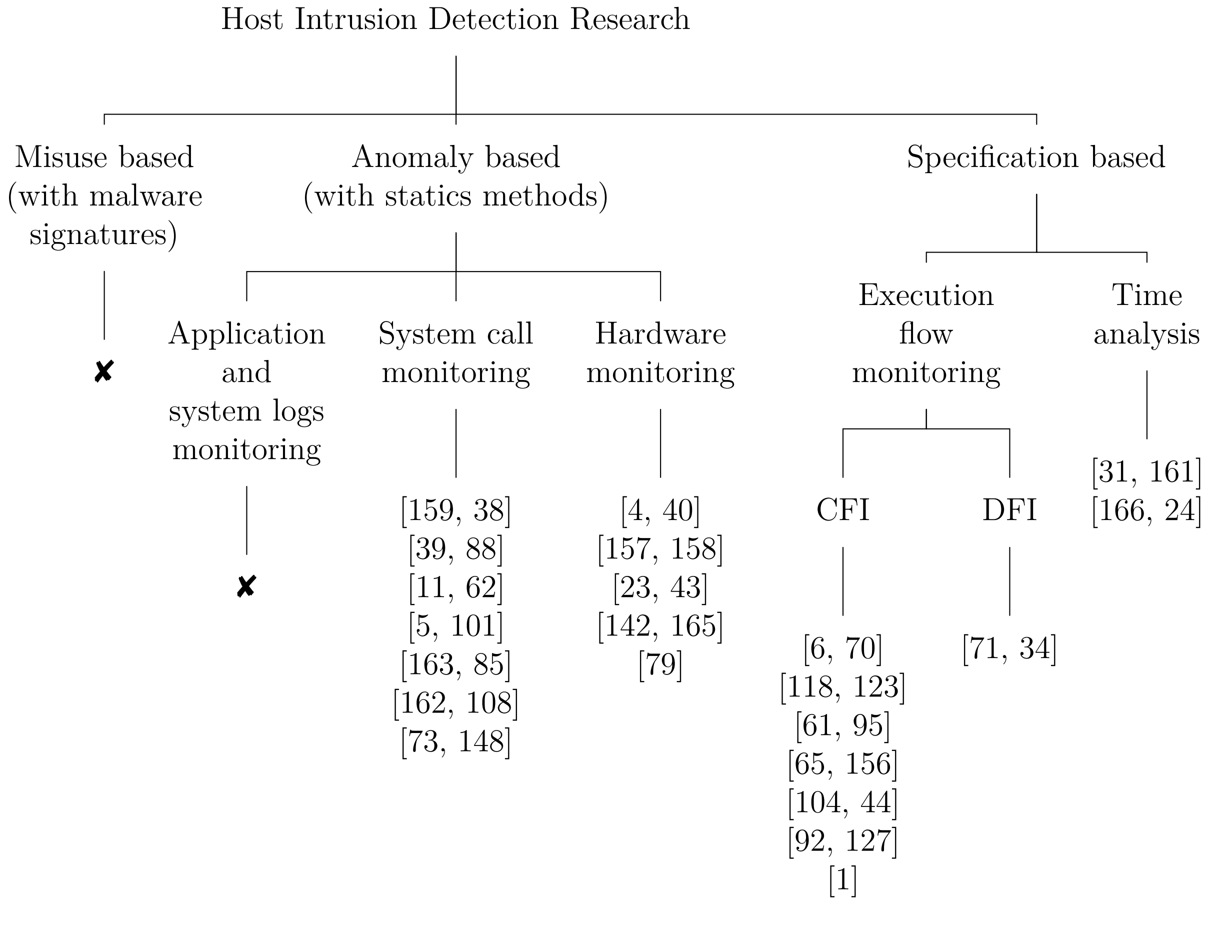


**Figure III.1:** *Overview of reviewed host intrusion detection approaches*

We review a non-exhaustive list of frameworks (notably for the broad topic of system call based monitoring): as we are not aware of comparative studies, we intend to provide with this study a state of the art on HIDS research for deployment in an industrial perspective involving system safety certification. More specifically, the aim is to evaluate how the diverse approaches can address the problem of HIDS deployment in an embedded MCS, referring to the criteria described in Table III.2.

Figure III.1 provides an overview of the HIDS approaches investigated in this literature review. Our two main focuses correspond to AI assisted anomaly-based and specification based analysis approaches (definitions in Section III.2). In the scope of our problem definition (Section II.2.3), we do not consider misuse based solutions: our goal is to detect any (i.e. possibly unknown) intrusion in the monitoree's execution.

Following the context of deployment described earlier in this section, our target of anomaly-based HIDS leverages low-level events tracing to limit intrusiveness towards industrial system certification. Consequently HIDS frameworks leveraging other trace types are out of scope for this literature review. Notably, we do not discuss solutions tracing physical signals such as temperature and power consumption [110], since they highly depend on the use-case and user-level application. We also do not cover HIDS based on application and system logs, as these high level data depend on the monitored application or require general-purpose computer infrastructures; i.e. with file-system, network stack support, etc. For example, some detection solutions exist for Linux [120] or Windows OS [115], which classify system logs leveraging specific OS counters referring to processes, processor, memory, thread, IP network stack activities. Our main focus is on basic OS level (system calls) and hardware based monitoring.

Specification based methods split between time analysis (i.e. watchdog systems, WCET static analysis based monitoring), and execution flow monitoring. Because they come with no false-positives, they are more suitable to be deployed into MCS than statistical based methods. Because of the deployment constraints we mainly focus our review on HIDS solutions, which use hardware assistance to limit the intrusiveness compared to software based solutions.

Table III.4 provides an overview of the different HIDS approaches in regard to the criteria defined in the previous section. In the panel of solutions with hardware dependencies, we only introduce ones using event tracing features widespread among hardware platforms and architectures (i.e. hardware performance counters and processor tracing hardware). On one hand, we consider all anomaly-based detection methods unsuitable for HIDS certification because of the uncertainty of the detection results (i.e. no guarantee against false-positives). In particular, none of the hardware based HIDS solutions which we surveyed comply to the AUTOSAR IDS specification; it is not a surprise as the first version of IDS specification [18] was added to AUTOSAR only in 2020. On the other hand, according to the table, such frameworks seem to induce workable effort for integration into certified systems. In reason of their transparency properties, system call and HPC tracing based solutions are likely to come with limited certification impact. Hence, we deduce that such anomaly based HIDS are good candidates for passive monitoring; i.e. detection data recording from the analysis of system execution traces. However, we also note that many of the reviewed approaches were tested offline [163, 159, 5, 11, 73, 148, 39, 101, 88, 108, 58]: i.e. they did not evaluate the runtime overhead for deploying the solution into embedded systems. This is especially the case of machine-learning based frameworks.For these solutions, the trade-off between security impact in terms of runtime performance overhead and system schedulability is still to be determined.

Few specification-based approaches for general-purpose computers are likely to be applied realistically into embedded MCS: several reviewed solutions [157, 158, 70, 44, 118, 6] indeed require program instrumentation, even though their implementation is based on hardware tracing. In addition, we see that the security accuracy evaluation seems comparatively easier for generic computers, as they can leverage more complex test cases; e.g. datasets of real-life threats for machine-learning based solutions. Though, while we consider that detection latency is a key metric to evaluate the security of a framework in an embedded MCS, very few of generic computers based solutions provide an evaluation; notably for works introducing offline HIDS, which only provide detection accuracy metrics.

We structure our literature review as follows:

- Section III.2 gives an overview of HIDS for general-purpose computer systems (e.g. PC, server). While they remained an active research topic over the years, HIDS solutions are now productized in the industry, mainly for IT applications.

- Section III.3 focuses on HIDS solutions for embedded MCS. Few works apply to these systems because of their specific constraints. On one hand, we can characterize embedded systems with limited computing capacity and storage resources. On the other hand, MCS may be constrained by safety requirements: a system failure can lead to disastrous consequences. For industrial deployment, MCS may imply a certification process: this limits the action for monitoring. Additionally, a system integrator has no necessarily access to monitored software source code and information (i.e. program instrumentation is not possible).

## III.2 Host Intrusion Detection in General-Purpose Computer Systems

This section covers HIDS solutions designed for general-purpose computers such as PC and servers. These systems usually support a user-interface OS, like Linux OS.

The objective of the HIDS is to distinguish anomalous and benign executions of a process to monitor. Table III.3 summarizes the set of collectable data on common systems. We develop two main detection approaches:

- Section III.2.1 and Section III.2.2 respectively develop CFI and DFI monitoring, two specification based intrusion detection solutions which analyze CF and Data-Flow (DF) transitions in the execution of the monitored program.

- Section III.2.3 introduces statistical based anomaly detection approaches to correlate an anomalous sequence of events with a threat execution.

**Table III.4:** *Comparison of intrusion detection solutions discussed in this study. Symbols (−, ×) for transparency and portability columns respectively stand for not applicable and required fields. For security, certifyability, and performance criteria, we define the following symbols (×, ❍, ◗, ●) respectively meaning no, basic, partial, and full support of the given field.*

| **Solution** | | | **Transparency** | | **Portability** | | **Security** | | | **Certifiability** | | **Performance** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | application instrumentation | kernel modification | custom hardware | technology dependencies | in-depth security evaluation | no false-positives | detection latency evaluation | certified systems integration | suitable for HIDS certification | runtime overhead evaluation |
| General-purpose computer systems | Yoon et al. | [163] | − | × | − | − | ❍ | × | × | ❍ | × | × |
| | Warrender et al. | [159] | − | × | − | − | ❍ | × | × | ❍ | × | × |
| | Abed et al. | [5] | − | × | − | − | ❍ | × | × | ❍ | × | × |
| | Anandapriya et al. | [11] | − | × | − | − | ● | × | × | ❍ | × | × |
| | Hu et al. | [73] | − | × | − | − | ● | × | × | ❍ | × | × |
| | Subba et al. | [148] | − | × | − | − | ● | × | × | ❍ | × | × |
| | Creech et al. | [39] | − | × | − | − | ● | × | × | ❍ | × | × |
| | Liu et al. | [101] | − | × | − | − | ❍ | × | × | ❍ | × | × |
| | Koucham et al. | [88] | − | × | − | − | ● | × | × | ❍ | × | × |
| | Maske et al. | [108] | − | × | − | − | ❍ | × | × | ❍ | × | × |
| | Fiser and Sanchez | [58] | − | − | − | × | ❍ | × | × | ● | × | × |
| | Basu et al. | [23] | − | − | − | × | ● | × | × | ● | × | × |
| | Abbas et al. | [4] | − | − | − | × | ◗ | × | × | ● | × | × |
| | DCFI-Checker | [142] | − | × | − | × | ❍ | × | × | ❍ | × | ● |
| | NumChecker | [157] | × | × | − | × | ● | × | ● | × | × | ● |
| | Wang et al. | [158] | × | − | − | × | ◗ | × | × | × | × | ● |
| | Krishnamurthy et al. | [89] | − | − | − | × | ● | × | × | ● | × | ● |
| | Cronin and Yang | [40] | − | − | − | × | ● | × | × | ● | × | ● |
| | PT-CFI | [61] | − | × | − | × | ◗ | ● | ● | ❍ | ◗ | ● |
| | FlowGuard | [104] | − | × | − | × | ● | ● | ● | ❍ | ◗ | ● |
| | $\mu$CFI | [70] | × | × | − | × | ● | ● | × | × | ❍ | ● |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Yongje et al. | [95] | – | – | – | × | ❍ | ● | × | ● | ◗ | ● |
| | HAFIX | [44] | × | – | × | – | ● | ● | × | × | ❍ | ● |
| | CaRE | [118] | × | – | – | × | ● | ● | × | × | ❍ | ● |
| | C-FLAT | [6] | × | – | – | × | ● | ● | × | × | ❍ | ● |
| Mixed-criticality systems | Hao et al. | [65] | × | × | – | – | ◗ | ● | ● | ◗ | ◗ | × |
| | RECFISH | [156] | × | × | – | – | ◗ | ● | ● | ◗ | ◗ | ● |
| | TrackOS | [127] | – | × | – | – | ❍ | ● | × | ● | ● | × |
| | OCFMM | [1] | – | – | × | – | ❍ | ● | × | ● | ● | × |
| | Bellec et al. | [24] | – | – | × | – | ● | ● | ● | ● | ● | ● |
| | Zimmer et al. | [166] | × | × | – | – | ◗ | ● | ● | ◗ | ◗ | ● |
| | Wolf et al. | [161] | × | – | × | – | ◗ | ● | × | ◗ | ◗ | ● |
| | Carreon et al. | [31] | – | – | × | – | ● | ● | ● | ● | ● | × |
| | Kuzhiyelil et al. | [92] | – | – | – | × | ◗ | ● | ● | ● | ● | ● |
| | Kadar et al. | [79] | – | – | – | × | ◗ | × | ● | ● | × | ● |

**Table III.3:** *Collectable traces in general-purpose computers to observe system execution*

| System level | Trace type |
|---|---|
| **Application** | - application binaries<br>- library calls<br>- user configuration (network connections, user filesystem, etc.)<br>- system log files |
| **OS kernel** | - system calls<br>- inter-process calls<br>- network packets<br>- system configuration (hardware resources, process activity, etc.) |
| **Hardware** | - System on Chip (SoC) performance counters<br>- processor hardware performance counters<br>- other processor-level control-flow related traces (e.g. using ARM CoreSight tracing) |

## III.2.1 Control-Flow Integrity Monitoring

### Control-Flow Hijacking Based Attacks

A memory corruption like a buffer overflow is a common vulnerability in modern applications and systems (e.g. Linux OS, Internet browser). Return-Oriented Programming (ROP) based attacks [131, 124, 134] exploit this type of flaws to hijack the control-flow (i.e. sequence of instructions) executed by the victim program, by executing a malicious combination of short instruction sequences called gadgets. In some cases, an adversary

**Table III.5:** *Common control-flow transitions*

| **Type** | **Instruction** | **Example on ARM64 architecture** |
|---|---|---|
| **Forward-Edge** | Branch to direct address | B *addr* (branch to address *addr*) |
| | Branch to indirect address | BR $R_N$ (branch to register $R_N$) |
| | Function call to direct address | BL *addr* (branch and link to address *addr*) |
| | Function call to direct address | BLR $R_N$ (branch and link to register $R_N$) |
| | Generate exception | SVC (supervisor call) |
| **Backward-Edge** | Return from function | RET (return to address stored in the stack) |
| | Return from exception | ERET (exception return) |

can eventually gain control over the whole system and cause disastrous damages: stealing confidential data, altering the execution of system services, etc. Therefore, CFI monitoring aims to protect a program against CF hijacking attempts.

### Control-Flow Integrity Monitoring Approach

We decompose CFI monitoring into 2 steps. First, the method generates before deployment a CFI policy to represent benign execution of the monitored program. This policy for example corresponds to a list of all accepted CF transitions for the program. We can generate the CFI policy with static or compile-time analysis; e.g. using LLVM compiler [150]. Then, a monitoring service continuously observes the program CF at runtime; it detects a security threat whenever an observation breaks the predefined CFI policy. We can split this second step into two main operations: CF trace collection and CFI checking. For trace collection, at any time or strategic points in the execution, the monitoring service should be able to access the CF transitions executed by the monitored program. Table III.5 lists common instruction types that can modify the CF (ARMv8-A CPU architecture Manual [13]). During CFI checking, the monitoring service compares collected runtime observations to the CFI policy.

### Control-Flow Integrity Monitoring Implementations

Table III.6 provides an overview of state-of-the-art CFI monitoring frameworks. We identify two main approaches for CFI monitoring: **Forward-Edge CFI** (FE-CFI) monitoring [6, 118, 70, 104, 123, 92] covers branch and call instructions, while **Backward-Edge CFI** (BE-CFI) monitoring [6, 118, 44, 61, 70, 95, 92] covers return instructions.

FE-CFI monitoring approach introduces Control-Flow Graphs (CFG) to analyze the

**Table III.6:** *Comparison of hardware-assisted and hardware-based CFI monitoring approaches*

| Solution | | FE-CFI | BE-CFI | Transparency |
|---|---|---|---|---|
| PT-CFI | [61] | – | ● | ● |
| FlowGuard | [104] | ● | ● | ● |
| $\mu$CFI | [70] | ● | – | – |
| Yongje et al. | [95] | – | ● | ● |
| HAFIX | [44] | – | ● | – |
| CaRE | [118] | ● | ● | – |
| C-FLAT | [6] | ● | ● | – |
| Kuzhiyelil et al. | [92] | ● | ● | ● |

execution flow of the monitored program at runtime. A CFG lists all valid FE-CF transitions (Table III.5), so that the monitoring service can detect any unauthorized jump attempt. We can define CFG at different granularity levels, statically with program offline analysis or dynamically at runtime. A static CFG can be generated at compilation step or with binary analysis of the monitored program. Statically generated CFG are often incomplete to cover complex monitored programs; for example when CF transitions are determined dynamically by dependencies on program inputs. Such protection makes the attack difficult but remains bypassable by a motivated and knowledgeable attacker. For example, RAP framework [123] implements a light-weight type based CFG. With this approach, for every indirect function call (e.g. via function pointer), the policy allows CF transfers to functions that have the same signature than the reference defined during static analysis. Alternatively, Kuzhiyelil et al. [92] also implement a type based CFG for FE-CFI monitoring, leveraging ARM CoreSight hardware processor tracing feature to transparently – i.e. without program instrumentation – generate CF traces to analyze. However, a recent research proves that such CFI policy can be bypassed by motivated attacker [112]. Intuitively in large programs, many functions have same signatures; this lead to potential collisions and over-approximation. The authors exploit this security breach to successfully hijack a vulnerable program protected by PaX's RAP monitor. Dynamic-CFG based FE-CF monitoring aims to improve the precision of the CFG to track incoherent CF transitions in function of the execution context. With $\mu$CFI, Hu et al. [70] enforce a precise FE-CFI policy based on Intel-PT processor tracing technology and compile-time analysis. During compile-time, they recursively identify constraining data (i.e., non-control data used in control-flow instructions) and insert routines at locations where these data are written, in order to dynamically trace their values at runtime. With this method, $\mu$CFI is able to enforce exactly one possible target for every FE transition. However, achieving this precision requires an intrusive modification of the monitored program. Liu et al. [104] also leverage Intel-PT to monitor critical code areas. Their framework FlowGuard provides FE-CFI and BE-CFI coverage, performing a dynamic analysis of the most recent execution traces stored at runtime. The authors speed up CFI checking by learning legitimate CFG transitions from previous observations.

BE-CFI goal is to protect return instructions against stack corruption: when a program returns from a function, the next instruction address to jump to is stored on the top of the stack. BE-CFI monitoring implementations maintain a second read-only call stack, called shadow stack. The shadow stack maintenance can be implemented in hardware [118, 61, 92] or in software [65]. The survey [36] considers hardware implementations more secure than software ones: because of the inherent physical isolation in memory, user access is generally not possible or more difficult. For example, Yongje et al. [95] utilize the ARM CoreSight hardware extension on a FPGA-based SoC to transparently enforce a precise BE-CFI policy. The authors build a dedicated FPGA soft core processor that is capable of extracting control-flow information from CoreSight traces and using that information to maintain and enforce a shadow stack. Similarly, Kuzhiyelil et al. [92] use ARM CoreSight hardware to monitor CF backward-edges of a program without code instrumentation. However, their solution differ since it implements the shadow stack on the CPU instead of requiring FPGA support. Their solution differ as the shadow stack maintenance runs on the CPU: it does not require FPGA support. CFI monitoring runs on a dedicated set of CPU cores and user-level application to assure the separation with the monitored program execution. Hence, both solutions are able to fully separate the BE-CFI enforcement from the monitored program at runtime. Another hardware-based solution for embedded systems, presented by Davi et al. [45, 44] as HAFIX, implements a hardware function call stack to enforce an imprecise backward-edge CFI policy The policy deployed by HAFIX forces function returns to target any call-preceded instruction residing within any function currently recorded on the function call stack. To implement the function call stack and CFI enforcement, Davi et al. develop new processor instructions on the fully synthesizable Intel Siskiyou Peak and SPARC LEON3 microprocessors. Finally, PT-CFI solution [61] extracts BE-CF traces in hardware with Intel-PT, to analyze them dynamically in a separate monitoring process.

Several solutions support both BE-CFI and FE-CFI monitoring. Nyman et al. [118] introduce CaRE, a binary rewriting solution that deploys imprecise FE-CFI and BE-CFI utilizing the ARM TrustZone hardware extension. In particular, CaRE secures indirect function calls by restricting target addresses to a list of valid function entry points, generated during a static pre-processing phase. To protect function returns, CaRE maintains a shadow stack securely isolated in the ARM TrustZone. C-FLAT [6], another solution utilizing the ARM TrustZone, defines a remote attestation protocol, in which a (bare-metal) embedded system proves its correct execution to a remote verifier. A cumulative hash chain is computed over the target addresses of indirect branches taken by the embedded system. The hash chain is calculated and stored securely within the ARM TrustZone. The final hash value is transmitted to the verifier and compared to a list of valid hash values gathered during a dynamic pre-processing phase. When performing continuous attestation, C-FLAT achieves complete conformance to the legitimate CFG. However, C-FLAT requires a dynamic pre-processing phase, which has to identify every benign path in the embedded software—a non-trivial task.

CFI is traditionally implemented in software. By the use of dedicated hardware components for CFI monitoring, the objective is to reduce the performance overhead. The survey [36] details the state of the art of hardware architectures that integrate CFI solutions in the literature. The authors show that practical and fine-grained CFI remains unsolved. Most of the initiatives in the survey are still research projects: from 21 studied CFI implementations, only two are based on commercial off-the-shelf hardware.

#### Performance Overhead

CFI monitoring aims at countering ROP-based attacks: as a specification based solution, it is potentially able to detect any intrusion that modifies the CF, with no false alarms. Though, the main issue is the performance overhead. In [42] for example, the authors measure approximately 10% of performance overhead for traditional implementations of shadow call stacks.

The performance of the CFI monitoring solution varies in function of the monitoring granularity; e.g. full or partial CFG coverage, filtering of monitored CF transitions, etc. In Table III.5 for example, jumps to direct addresses and supervisor call instructions do not represent vulnerable entry points for an attack, since we assess that an adversary has no mean to modify read-only memory of the monitored program at runtime. On one hand, a fine-grained CFI based implementation is more efficient for security, even though it brings a significant performance overhead. Thus, such precise monitoring is more suitable for simple programs. On the other hand, coarse-grained CFI monitoring offers a trade-off between performance and security.

To reduce the performance overhead, a first approach is to apply a partial CFG coverage. Another strategy consists of privileging specific vulnerable regions in the program source code to run CFI monitoring. For example, FlowGuard [104] and PT-CFI [61] are light-weight solutions, since they perform CFI checks at predefined endpoints (such as system calls) a fixed sequence of the most recently recorded traces.

### III.2.2 Data-Flow Monitoring

#### Data-Flow Hijacking Based Attacks

CFI monitoring does not cover non-control data, which Chen et al. define as «data that are loaded to processor program counter at some point in program execution » [34]. More specifically, CFI monitoring involves program instructions and data which are directly used in a CF transition executed by the monitored program. Consequently, the scope of CFI monitoring excludes other (non-control) execution context related data. For example, a CFI monitoring solution can check the target address of a conditional branch, but it does not consider the correctness of the condition value.

A set of program vulnerability exploits can bypass fine-grain CFI monitoring security solutions: because CFI monitoring does not protect non-control data, an attacker can modify them to influence the context of execution and finally hijack CFI defense. In 2005, Chen et al. alerted against attacks targeting the data surface [34]. More recently, Hu et al. [71] have illustrated the potential of such attacks called Data-Oriented Programming; they prove their feasibility by performing privilege escalation and data leakage on examples of real-life X86 programs.

### Data-Flow Integrity Monitoring Approach

The goal of Data-Flow Integrity (DFI) monitoring is to detect data-flow hijacking attempts by controlling the integrity of non-control data of the monitored program at runtime. DFI monitoring follows a similar approach to CFI monitoring. During the compilation of the monitored program, metadata are generated to point out critical non-control data, and locations of legitimate memory writes. At runtime, the framework transmits memory writes to such critical data to the monitor, which checks the legitimacy of the operation with the reference policy.

Because of the amount of non-control data modifications (i.e. all write instructions in the monitored program), the performance overhead can be significant. A first solution to limit the impact on performance is to filter relevant non-control data to monitor, e.g. by defining criticality levels: Work by Chen et al. [34] defines critical non-control data as the data involved in configuration, decision-making, user input, and user identity operations.

### Combination with Control-Flow Integrity

On one hand, a framework combining CFI and DFI protection forms a complete runtime detection of threats to protect the CF of a monitored program against sophisticated attacks. Because it is a specification-based method, it does not bring false detection. On the other hand, we can expect an important performance overhead to monitor the data flow. Thus, the trade-off between security coverage and performance overhead must be determined for the monitored software. The monitoring impact depends greatly on the properties of the application (like the amount of computational operations and memory accesses), since in principle, a CFI or a DFI policy is generated for a single program. A limitation could also come from the necessary access to the monitored program source-code – to build the CFG at compile-time or to insert instrumentation in the binary for runtime monitoring.

## III.2.3 Machine-Learning and Statistical Based Anomaly Detection

### Overview

We divide the machine-learning based anomaly detection process in two main phases. The trace collection phase provides a sequence of system signal observations representing the monitored execution to the second trace analysis phase, which applies machine-learning techniques to distinguish malicious from normal execution on collected traces.

The selection of relevant system signals plays a decisive role to distinguish normal and malicious execution with machine-learning. It varies in function of the execution context (i.e. system hardware and software architectures) and can target several system levels: hardware activity through hardware performance counters (HPC), OS-level operations with system calls (SC), or system log files. In this work, we focus on HPC and SC based solutions.

Jose et al. [76] describe three types of anomaly-based data analysis techniques.

- Statistical-based analyses apply statistical tools to produce a model of normal execution represented by a set of observed variables, e.g. leveraging metrics such as mean, variance, distributions thresholds. For example, Yoon et al. [163] propose a light-weight system call based anomaly detection based on system call frequency distributions analysis.

- Machine-learning based analyses follow a similar approach to statistical based methods: they build a machine-learning model of normal execution, through a training involving heavy data load.

- Knowledge-based analyses use knowledge about a set of known-attacks to detect any attack or system vulnerability. Although it is supposed to reduce the false-positive rate, we do not develop this topic further, because in the context of industrial critical system, we do not have access to many attack types.

### System Call Based Host Intrusion Detection

A system call is a specific assembly instruction which generates an exception trapped in the OS kernel, so that a user-level program can call OS kernel services (like memory page allocation, hardware I/O access). For example on ARMv8-A architecture, it corresponds to the SuperVisor Call (SVC) [13]. The system calls set depends on the OS software Application Programming Interface (API).

System call is a well-known data type to detect anomalies in a program's execution. System call based HIDS has been indeed an active research topic since the late nineties [159]. System call data are part of several popular datasets from the last decades to support research on HIDS: with ADFA-LD [38, 72], KDD99 [154], DARPA [153]. Intuitively, this type of signals discloses information of the monitored user-level software interface with lower more privileged system layers. Generally, an attack on a user-level program aims at compromising shared resources or more privileged system layers. Hence, it potentially involves system call instructions to access critical resources and operations. Additionally, we can deploy system call tracing at kernel level; i.e. such solution is non-intrusive for the monitored software, as we can observe it without instrumentation.

For the machine-learning analysis, a system call is an abstraction that can be represented by different sets of traces, as described by Wunderlich et al. [162]. Although the great majority of solutions exclusively leverage the type of system call [5, 11, 62, 73, 148, 38, 39], some initiatives [101, 88] propose to enhance the analysis with contextual data (i.e. system call arguments, return value, occurrences, handler execution time, etc.). The survey by Wunderlich et al. [162] discusses different approaches to pre-process system call data to then feed neural networks, which take numeric values as inputs. One-hot encoding is a basic pre-processing method, which transforms a system call data to a numeric vector with a dimension matching the count of different data in the system call API. In this representation, every system call corresponds to a single vector dimension; e.g. for PikeOS API, every system call corresponds to a vector of 122 dimensions. More advanced pre-processing techniques based on classification for Natural Language Processing (NLP) like word2vec [111] can be applied. Such approaches suppose that some system calls are not totally independent. For example, file management related system calls (open, close, read, write, etc) presume an implicit time relation: the file is open prior to other operations, then read or write, and finally closed. The final dimension of the vector depends on the set of system calls. In NLP, for an alphabet of millions of words, a few hundred dimensions are generally sufficient to analyze a text. For instance with word2vec algorithm [111], the input vocabulary corresponds to one million words, while the output vector's dimension remains below 600. Wunderlich et al. [162] compare different pre-processing methods for system call type inputs, using ADFA-LD datasets and the same detection algorithm, a Long Short Term Memory (LSTM) classifier. Their results show in this context the best detection accuracy with one-hot encoding.

system call based HIDS generally come with good detection performance: generally, the detection accuracy is above 90% and the false-positive rate (FPR) varies. Maske et al. [108] compare two methods on ADFA-LD dataset, with less than 4% FPR and more than 80% accuracy. Subba et al. [148] analyze their solution with two others from literature, in the majority of tested cases, the detection rate and the accuracy respectively are above 80% and 90%. Creech et al. [39] implement 100% detection rate for 0.6% FPR, while Anandapriya et al. [11] apply an Extreme Learning machine model to detect intrusions in ADFA-LD datasets, achieving 86% detection accuracy and 2.2% FPR. Despite the quantity of solutions and performance numbers, security remains difficult to evaluate. Even though open datasets allow to compare different frameworks with the same data inputs, they lack representativeness of program execution: while DARPA and KDD99 datasets are over 20 years old. The most recent open dataset ADFA-LD [38] is from 2013. Besides, such datasets are highly system dependent. For example, ADFA-LD corresponds to traces of processes executing on Ubuntu-11 server OS, which potentially differ from traces observed on embedded or RTOS. Pendleton et al. [125] propose a solution to generate more suitable system call datasets and highlight general limitation of existing datasets: system call sequences are non deterministic and depend on the monitored process' complexity.

Research articles usually do not mention the time and memory overheads for their system call based HIDS, probably because such technology target deployment in server systems, in contrast with embedded systems having limited resources. Additionally, the performance overhead depends greatly on the monitored application; i.e. how frequently it executes system calls (the worst-case being for a program consisting of a sequence of system calls only, while the best-case is a program executing no system call). A program executing few system call comes with low performance overhead. However, the system call based protection of the program execution could be more easily bypassable by a knowledgeable adversary, especially if malicious execution does not generate further system calls during runtime. The time window of opportunity for an attack also increases in such configuration: the adversary has more time to make its attack successful before being detected.

#### Hardware Performance Counters Based Host Intrusion Detection

Hardware Performance Counters (HPC) are CPU core-wise configurable counters. These counters are available in modern widespread CPU architectures such as ARMv8-A and

**Table III.7:** *Common hardware performance counter events*

| Type | | Examples |
|---|---|---|
| Memory | virtual memory related events<br>cache operations<br>memory operations | TLB flush, TTBR write<br>flush, invalidate<br>load, store instructions |
| CPU | pipeline<br>speculative execution<br>program execution | stall instructions<br>(mis)predicted branches<br>exceptions, executed instructions |
| Time | | bus cycles, CPU cycles |

**Table III.8:** *Hardware performance counters traced in literature*

| Solution | | HPC Events | | | |
|---|---|---|---|---|---|
| | | Instructions | Branch | Branch Predictor | Cache |
| Basu et al. | [23] | all<br>integer inst. | all<br>taken br. | - | - |
| Krishnamurthy et al. | [89] | all | all | - | - |
| Fauzi et al. | [4] | call inst. | - | br. miss | I cache miss<br>D cache miss |
| Cronin and Yang | [40] | all | all | br. miss | cache access<br>cache miss<br>LLC access |

Intel-64. Most of the hardware platforms supporting HPC trace the same set of common events (see Table III.7). For an ARM Cortex-A53 processor [132], more than 50 events can be configured: cache accesses, branch predictor performance, executed instructions, etc.

Recent research has emphasized the applicability of Hardware Performance Counters (HPC) tracing to detect malicious exploits: e.g. work by Fiser and Sanchez [58] shows how to detect Spectre [87] intrusion by observing cache activity at runtime. Table III.8 shows a set of common HPC events used in recent literature. Basu et al. [23] demonstrate the relevance of tracing CPU performance counters for detecting anomalies in a program CF. The authors compute the probability for two CF executions to correspond to the same HPC traces: they confirm the probability of matching to be very low, validating their approach by using a selection of four common hardware events.

Abbas et al. [4] implement an offline anomaly detector, which traces four events to profile a benchmark of applications. The initial benchmark defines the baseline, while patched version and combination of these applications represent anomalies. The detector nearly identifies all deviations. While this approach applies offline, after the monitored program execution, Shi et al. [142] focus on online anomaly detection to identify deviation in an application's CF at runtime. They develop DCFI-checker, a CFI monitor, which counts executed branches during each control transfer in the kernel for detecting intrusions.

NumChecker [157] is a security framework based on system call profiling using performance counters for detecting kernel rootkits in guest virtual machines (VMs). The method consists of creating offline profiles for a set of system calls in a safe environment. At runtime, a test application performs regularly system calls inside the guest VM. The monitor reads the counter values during the test and compares the results to the baseline. When read values exceed a certain threshold, an intrusion is detected. ConFirm [158] is a malicious firmware detector for embedded systems. Before starting the firmware, the tool inserts checkpoints in the executable. It reads CPU performance counters at each one of these checkpoints and analyzes the measurements: it either compares counters' traces to a reference signature, or it uses a predefined machine learning model to determine whether a potential threat is occurring. As both ConFirm and NumChecker frameworks modify the monitored software, they induce indeterminism in time execution.

A novel IDS implementation for multi-threaded processes tested on PLC software [89] considers each monitored process as a black-box. The method creates a baseline of benign execution from the observation of the monitored process execution in a trusted environment. The paper demonstrates very good precision of the anomaly detector; but it does not disclose its impact on system performance. Work by Cronin and Yang [40] discusses the performance impact for tracing HPC at system runtime. The authors propose an online system malware detection solution with low frequency sampling period. They reduced the memory bandwidth to 7KB/s, compared to the 3MB/s required in previous work, confirming the validity of HPC tracing deployment for security in embedded systems.

Even though many research works outlined the relevance of HPC for intrusion detection, some limitations have emerged from recent publications [165, 43]. Zhou et al. [165] prove that many solutions base their results on unrealistic setups and optimistic assumptions (e.g. the programs in the training set appear in the test set). They believe there is no causation between low-level architectural events and high-level software behaviors. Sanjeev et al. review the different HPC types and recent analyses methods. The authors define a set of recommendations to use HPC for anomaly detection. First, evaluation results of different frameworks cannot be compared, because of the high variability of observations across platforms and test setups. Second, to perform HPC-based application profiling in a multi-applications execution on the same CPU core, the solution should support routines to save and restore the counters values at context switch. Third, the trace analysis should consider non-determinism of HPC measurements; an adversary could attempt to influence the counters values to thwart the defense. Finally, a new framework should be documented for reproducibility and consistency in literature.

## III.3 Host Intrusion Detection in Embedded Mixed-Criticality Systems

### III.3.1 Applying Host Intrusion Detection in Embedded Mixed-Criticality Systems

#### Dedicated Solutions for Embedded Mixed-Criticality Systems

Industrial critical systems usually follow rigorous processes for the development, verification, and validation. For example, PikeOS real-time hypervisor complies with several industry standards reaching various assurance levels, involving the Automotive Safety Integrity Level B, Design Assurance Level A for the avionics, etc. Hence, critical certified software are unlikely to have software bugs, in contrast to non-certified software.As described in Section II.1.2, a MCS is a system composed of diverse software applications from different criticality levels. In MCS, low-critical tasks are more likely to have software bugs compared to high-critical software; additionally, low-critical tasks are comparatively more accessible, notably via connectivity features and user interface. Therefore, they are more vulnerable to attacks.

With the openness and large attack surface commonly associated with non-critical software such as infotainment, networking, and multimedia applications, and despite their secure design, MCS become increasingly exposed to threats. Even though theses systems support prevention mechanisms to preserve critical tasks from corruption (e.g. multi-mode execution including a fail-safe state), an infected low-critical task can alter the normal execution of a critical task: used by a high-critical task, a low-critical service becoming unavailable or compromised potentially reduces the user-experience and adds inconvenience. In worst-case, low-critical tasks can impact the execution time of high-critical tasks beyond their real-time constraints. For example, Chen et al. [33] have demonstrated the feasibility of side-channel attacks on industrial critical systems: they implemented an attack against a custom rover, that alters the system behavior by overwriting system control signals from a low-critical task. In work [106], the adversary indirectly manipulates low-critical tasks to increase the jitter of a more critical task to finally compromise temporal properties of the MCS. Thus, low-critical tasks represent attractive entry points to compromise MCS.

Hence, as developed in Section II.2.1, the system must guarantee freedom from interference and independence between system tasks, to assure a proper system operation complying with execution constraints such as real-time deadlines. For example, PikeOS hypervisor implements MILS architecture, leveraging resource and time partitioning strategies; it achieves Evaluation Assurance Level 5+ for security [50].

In real-time systems, timing is a central variable, both for the attack and protection. Iliya and Ivo Georgiev [60] discuss the vulnerability of real-time systems through a set of timing parameters, which they classify in function of their potential critical impact level. According to their classification, changing the main clock frequency corresponds to the highest criticality level at system level. Several attacks [33, 106] exploit timing scheduling properties to leak confidential information or compromise the execution of a real-time system. Therefore in reaction, scheduling design defenses aim at limiting the attack surface: Mohan et al.[114] propose security-aware fixed-priority real-time schedulers. Other approaches prevent timing deviation side channels via obfuscation based methods, like schedule randomization [90, 116]. Though, these design-based techniques cannot mitigate the CF hijacking attempts, which run in compliance with the time execution specification (e.g. WCET constraint).

Some basic security features are related to safety properties. For example, the resilience to system errors protects a system against random faults (e.g. software bug); it also contributes to increase the system robustness to handle errors resulting from an explicit malicious intention. However, safety properties are not sufficient to guarantee system security: an adversary, contrarily to a random fault, tries to dissimulate malicious activity, and possibly bypasses the safety controls. The spread of MCS is recent, due to the growth of hardware capacity; therefore, while MCS security has become a crucial issue, the literature counts few HIDS solutions designed for this type of critical systems. Section III.3.2 provides an overview of current HIDS solutions for MCS.

HIDS methods for generic computers introduced in Section III.2 cannot directly apply to MCS. Since these solutions run concurrently to the monitored software, they possibly modify temporal properties of systems like real-time constraints; therefore, we must evaluate their intrusiveness before deployment. Our objective is to define monitoring methods, which do not endanger the system constraints.

#### Methodology to Integrate Security Tasks in Embedded Mixed-Criticality Systems

Hasan et al. propose generic methods to integrate security tasks in a real-time systems, which address the trade-off between security monitoring and system impact. The authors first introduce an opportunistic solution [67], leveraging a server with low priority to execute security tasks. Their approach supposes a deferrable execution of the security tasks, so that security activity can execute when no task with higher priority is running. The authors generalize their opportunistic server approach with Contego [66], an adaptive framework where the server does not necessarily run with lowest priority. They introduce a multi-mode model. The passive mode corresponds to the opportunistic server method described above. In the second active mode, the server can execute with higher priority. We leverage this abstract approach in our practical research scope in Chapter VII, to propose a multi-mode HIDS; our solution involves non-intrusive heuristics based monitoring in low security mode and intrusive specification-based method in high security mode.

Hao et al. [65] address the problem of integrating instrumentation-based defenses in real-time systems, proposing two schedulability algorithms to control the time overhead induced by security checks in the monitored tasks. They define for every task a security level and a time overhead for instrumentation to perform security checks. Their schedulability algorithms decide whether a job can run security checks, in the limit of time to guarantee system schedulability, prioritizing tasks with higher security levels. The authors apply their approach on a auto-drive car prototype, implementing a software-based backward-edge CFI monitoring (shadow stack).

Compared to these two principal integration approaches based at task level, we provide in Chapter IV, a practical methodology for a safety-aware integration of a HIDS into an embedded MCS; we introduce a system architecture to integrate monitoring mechanisms, as well as a set of criteria and metrics to evaluate the HIDS in terms of security, system overhead, and impact on system schedulability.

## III.3.2 Host Intrusion Detection Solutions Designed for Embedded Mixed-Criticality Systems

Table III.9 provides an overview of research approaches discussed further in this section. Most of the solutions are specification-based: CFI monitoring, watchdogs, and software obfuscation. They follow diverse implementation strategies: at software level via program instrumentation and RTOS kernel modification or on custom hardware. While the evaluation process varies among the discussed frameworks, we consider three main elements for comparison:

- the performance impact for deployment of the method (e.g. in terms of time overhead and hardware cost).
- the security coverage offered by the solution (e.g. condition for threat detection, time to detect, etc.).
- the impact of the security approach deployment on the system schedulability; i.e. on real-time constraints of the system.

Some hardware-only based solutions [1, 24, 31] have no direct impact on the software execution at runtime. Few papers [65, 156] address the trade-off between security and system schedulability, while others [57, 166] only investigate the performance impact through the time overhead for monitoring estimation. Several methods [161, 24, 1] based on hardware implementations, evaluate the hardware cost; i.e. necessary additional computing and memory overheads for monitoring. The set of compared frameworks shows a significant heterogeneity in the security evaluations. Some works [127, 1, 161] provide a proof of concept, Walls et al. [156] discuss the scope of their solution analytically, and others [24, 31] propose to test the security on complex realistic use-cases. Though, many approaches use the attack detection latency as the principal evaluation metric [24, 166, 31, 65, 156].

### Specification-Based Solutions

CFI Monitoring directly impacts time execution of the monitored applications, because it requires instruction-level tracing. Such security solution seems unsuitable to secure system with real-time constraints. TrackOS [127] is a RTOS designed to integrate CFI monitoring. Before runtime, binary static analysis generates a call graph for each task (application), to control every function call and return instructions of monitored tasks at runtime. The CFI monitor occupies at runtime one privilege task; it can access all tasks's memory and checks their control stack with the call graphs generated in the previous step. To comply with real-time constraints, the user can freely control the

**Table III.9:** *Comparison of intrusion detection solutions designed for embedded mixed-criticality systems. Symbols (–, ×, ●) respectively stand for not applicable, missing, and applicable fields. In the security evaluation (○, ◗, ●) represent basic proof of concept, limited security metric measurements, and exhaustive security checks in a real-life complex environment levels.*

| **Solution** | | **Implementation** | **Program instrumentation** | **RTOS kernel modification** | **Protection type** | **Security/schedulability trade-off** | **Security evaluation** | **Performance evaluation** |
|---|---|---|---|---|---|---|---|---|
| Hao et al. | [65] | software | ● | ● | CFI | ● | ◗ | × |
| RECFISH | [156] | software | ● | ● | CFI | ● | ◗ | ● |
| TrackOS | [127] | software | – | ● | CFI | × | ○ | × |
| OCFMM | [1] | hardware | – | – | CFI | – | ○ | ● |
| Bellec et al. | [24] | hardware | – | – | WCET monitoring | – | ● | ● |
| Zimmer et al. | [166] | software | ● | ● | WCET monitoring | × | ◗ | × |
| Wolf et al. | [161] | hybrid | ● | – | watchdog | × | ◗ | ● |
| Fellmuth et al. | [57] | software | ● | – | obfuscation | × | × | ● |
| Carreon et al. | [31] | hardware | – | – | timing statistical analysis | – | ● | × |
| Kuzhiyelil et al. | [92] | hybrid | – | – | CFI | × | ○ | ● |
| Kadar et al. | [79] | hybrid | – | – | CF based monitoring | ● | ◗ | ● |

overhead of the monitor task with the RTOS scheduler. This solution allows a partial CF-coverage: it only considers function signatures, it does not control other branches such as conditional statements. RECFISH [156] is another CFI monitoring solution based on FreeRTOS open-source real-time system. Because it requires program instrumentation, this approach is less suitable for deployment in certified legacy systems. With OCFMM framework [1], Abad et al. propose an on-chip CFI monitoring solution, which is adapted to protect systems with strict real-time constraints. The authors define the framework time overhead in function of the monitored program source code. This overhead must be considered by the system designer to assure the schedulability of the whole system at runtime. Though, the authors provide a definition of the time overhead, they do not disclose techniques for practical measurement. They implement this technique as an extension of LEON3 processor.

As discussed in Section III.2.1, many frameworks involve processor-tracing hardware on common platforms for CFI monitoring [61, 59, 104, 70, 95, 96]. However to our knowledge, we are the first ones to design such a solution for deployment into critical systems with timing constraints [92]. In a second phase, we extend our initial monitoring framework to propose a configurable and deterministic monitoring solution, which addresses the trade-off for CF tracing between time overhead CF coverage [79]. We describe this safety-aware CF monitoring framework in Chapter VI.

Alternatively, Fellmuth et al. [57] deploys a WCET-aware artificial diversification mechanism to protect real-time systems against code-reuse based attacks. The artificial software diversity is an obfuscation approach consisting of running multiple randomly modified semantically equivalent variants of a program, making potential attacks more complicated and less reliable. To adapt the artifact software diversity for real-time systems, the authors base the diversification process on static timing analysis of the monitored real-time task to secure. Their solution require source code access and instrumentation at compile-time.

Survey [107] summarizes watchdog methods for error detection. Watchdog techniques investigated to protect systems against errors, randomly generated or caused by an attack on the monitored component. They involve a coprocessor to perform online sanity checks at runtime, by observing system-level behavior. The observation can focus on the control-flow execution, memory accesses, or timing properties. In the great majority of implementations, when the monitored hardware does not support transparent (i.e. non-intrusive) online trace collection, program instrumentation is necessary to provide the coprocessor with the data to check at runtime. For example, Work [161] proposes a watchdog based intrusion detection to monitor programs, based on the comparison of the execution time of protected program blocks with their WCET metric. While leveraging program instrumentation, the solution aims to assure the correct timing behavior of the monitored program.

### Temporal-Based Statistical Analysis Solutions

Bellec et al. implement a hardware framework [24] for embedded real-time systems, to detect CF hijacking threat such as ROP attacks. Their solution monitors the execution time of pre-defined code regions of the program to protect: when a monitored code region's execution exceeds its WCET, the framework detects a threat. However, the inherent pessimism of the WCET metric could be exploited by a knowledgeable adversary to hide additional malicious execution time.

Similarly, Zimmer et al. [166] leverage timing information to detect attacks such as buffer overflow attacks. Contrarily to the previous solution [24], this approach requires program instrumentation to insert checkpoints for execution time measurement controls. When integrating the solution to a real-time system, the predictability of the solution cannot be guaranteed: the monitored program now contains additional system calls at checkpoints for monitoring.

Carreon et al. [31] implement an anomaly-based malware detection solution using a window-based statistical analysis of timing measurements on individual system subcomponents. They leverage cumulative distribution functions to model the timing normal execution of protected subcomponents. After a first training phase to create the timing model, a hardware malware detector is deployed on the monitored critical system, to compare the model to runtime traces collected via the processor trace port.

In Chapter V, we study the practical use of ML to monitor system events like HPC and system calls, for intrusion detection in an embedded MCS. We present a safety-aware system call instrumentation solution [80], introduce a safe and secure system monitoring architecture [81], and discuss the impact of the ML engine configuration on system performance [109].

## III.4 Conclusion

This literature review provides a holistic overview on state-of-the-art approaches for deploying intrusion detection into industrial embedded MCS. We covered HIDS research in GP computers and discussed the applicability of existing solutions to run into an embedded MCS. We also compared intrusion detection frameworks designed to run in such constrained environment. In this evaluation, we defined a set of key criteria and properties to support further HIDS development in the context of industrial embedded MCS.

Considering this overview of HIDS research, we extend state-of-the-art HIDS research in this thesis through the following contributions:

- **This chapter – a literature review of HIDS for embedded MCS.** We submitted a survey to ACM Computing Surveys journal [78].
- **Chapter IV – a methodology for a safety-aware deployment of HIDS into embedded MCS.** This work includes a paper published in the 50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks [81] (2020). Additionally, in the joint paper submitted to the 25th IEEE International Conference on Emerging Technologies and Factory Automation [48] (2020), we contributed to the security aspects of industrial use-cases of a common fog computing platform.
- **Chapter V – a study of ML based HIDS, leveraging low-level system events (i.e. HPC and system calls).** We presented a system call instrumentation framework for safety-critical systems to the 4th International Workshop on Security and Dependability of Critical Embedded Real-Time Systems [80] (2019). We also supervised the M.Sc. Thesis of Andres S. Mateus [109], to build a HPC based HIDS using a ML framework.
- **Chapter VI – a safety-aware hardware-assisted CF monitoring framework.** In the joint paper submitted to the 23rd Information Security Conference [92], we presented a hardware-assisted CFI framework (2020). Our contributions consist of the implementation of forward-edge CFI checking, the integration of CFI checks on the test platform, the timing overhead analysis, as well as parts of

the results analysis and literature review. We then extended this framework in [79], for a practical and safety-aware deployment in an industrial embedded MCS.

- **Chapter VII – an analytic multi-mode HIDS composed of heuristics and specification based monitoring methods.**

IV

# A Methodology for Runtime Anomaly Detection in Embedded Mixed-Criticality Systems

Referring to the terminology of anomaly based intrusion detection introduced in Chapter II.1.3, we define for a program's execution an anomaly as an abnormal system behavior that can lead to an intentional malicious fault. Section IV.1 describes the basic idea to develop a safety-aware runtime HIDS for embedded MCS, which addresses the specific constraints of these systems.

A first key issue for developing such framework is to select suitable system events to trace and analyze for intrusion detection. Section IV.2.1 proposes a method to identify anomalies observing system traces, by selecting appropriate trace sources.

A second problem for anomaly based intrusion detection lies in the fact that an anomaly can correspond to an intrusion as well as a abnormal benign system behavior. Therefore, to be relevant, the solution must find an acceptable trade-off between correct detection of intrusions and false alerts. In this perspective, Section IV.3 introduces the two main approaches we develop in this thesis (Chapters V and VI). In Section IV.4, we then propose a method and an architecture to integrate the HIDS monitor into the local embedded MCS platform, which are compatible with safety criticality constraints of such system.

Finally, considering the system architecture and the type of proposed solutions, we propose an approach to evaluate the HIDS, in function of three criteria: detection efficiency, compliance with system safety requirements, and impact on system performance.

## IV.1 A General Approach for Safety-Aware Host Intrusion Detection in Embedded Mixed-Criticality Systems

In this section, we discuss three main problems to address for developing a safety-aware HIDS in embedded MCS:

- **selecting suitable system events** to trace for runtime intrusion detection.
- **selecting analysis methods** to distinguish anomalous from normal execution in system traces.
- **guaranteeing safety-related system constraints** when deploying runtime monitoring.

We propose several approaches addressing these challenges in Section IV.3.

### IV.1.1 Program Execution Tracing for Security

According to Section III.2 with Table III.3, many system events can be traced at different levels from hardware level to application level. Our literature review shows that several research works already highlighted the relevance of specific system events like system calls and HPC.

Formatting collected system traces represents a key stage in the intrusion detection process, to extract meaningful and interpretable information from system events. For example, an AI assisted solution analyzing human readable system log messages needs to convert traces to a machine-readable format. Alternatively, statistical based methods based on numeric continuous signals (e.g. HPC values) may require data normalization prior to computation.

Due to the inherent constraints of RTOS and system certification, embedded MCS offer limited data sources compared to general-purpose computers; e.g. no filesystem, restricted access to the monitored program's execution, difficult – in terms of certification – modification of the OS kernel, diversity of OS and hardware platforms, etc.

Hence, we propose to consider basic system events for intrusion detection; these must be accessible on standard hardware and software platforms. In particular, we focus our work on system calls, HPC, and common hardware tracing features.

### IV.1.2 Analysis Methods to Identify Intrusions

Our literature review (Chapter III) focuses on two HIDS analysis approaches: specification based and anomaly-detection based methods. On one hand, specification based solutions such as CFI monitoring (Section III.2.1) come with no false-positives, as they define rules describing normal execution of the monitored program – considering that no over-approximation rule induces wrong detections. In particular, specification based methods leveraging instruction-level analysis can support fine-grained detection. However, these solutions often bring challenging implementation constraints such as program instrumentation of the monitoree' source code.

On the other hand, anomaly-detection based methods based on statistical and AI analyses (Section III.2.3) can cover the execution at different program granularity levels, depending on the sampling rate for trace collection. These approaches can exploit various data sources, including hardware counters of the monitored processor. The relevance for threat detection of such HIDS relies on the exhaustiveness of the dataset used to build the monitoree profile of normal execution: i.e. we must collect enough data measurements to represent the monitoree running under all possible runtime configurations.

We propose to investigate how both specification based and anomaly-detection based solutions can be coupled to address the system safety-critical constraints, security efficiency, while limiting the impact on the system.

### IV.1.3 Safety-Aware Runtime Monitoring in Embedded Mixed-Criticality Systems

Due to the inherent constraints of our deployment environment (Section II.2.1), the HIDS solution must be transparent: the system architecture and scheduling policy must enforce strong freedom from interference and independence between user-level applications.

Hence, in our role of system integrator and RTOS provider, we must evaluate the certifiability of the HIDS solution. Notably, as system call data are well used and known in literature for intrusion detection in generic computers (Section III.2.3), we must consider the certification cost to integrate such HIDS framework into our embedded MCS; especially as system call types depend on the OS API, and tracing requires kernel-level modification of system call handlers.

## IV.2 System Events Correlation with Runtime Malicious Execution in Embedded Mixed-Criticality Systems

### IV.2.1 An Abstract Representation of Anomaly Detection

In our context of work, an anomaly is an abnormal system behavior that can lead to an intentional malicious fault. The goal of HIDS is to detect anomalies by observing the behavior of the monitored software execution. Since an anomaly does not necessarily correspond to an intrusion, the HIDS must be able to distinguish anomalous execution with no malicious faults from real threats in order to avoid detection errors.

The selection of trace sources to feed the HIDS directly influences the ability of the solution to correctly monitor the execution. The sources of observation vary among HIDS solutions and deployment environment related constraints, as discussed in literature review (Chapter III). A relevant set of trace sources must allow an acceptable trade-off between false-positives and false-negatives in the detection.

#### System Asset Formal Definition

As introduced in the problem definition (Section II.2.3), our goal is to protect a user-level application against intrusions altering its runtime behavior; we define our asset as the integrity of this user-level application's runtime execution (Section II.2.2). The integrity of the execution for a program depends on its control-flow and data-flow at runtime:

- **control-flow:** the sequence of instructions (exceptions, memory operations, function calls, etc.) executed by the program.
- **data-flow:** the data states which are read and written by the program (e.g. I/O, internal CPU registers, main memory).

Both data and CF depend on:

- **the static program binary:** executable code and static data.
- **dynamic system configuration:** hardware state, runtime input/output data, scheduling policy, etc.

Hence, our asset must cover both statically and dynamically determined aspects of runtime execution, so that the HIDS is able to detect intrusions, which correspond to our two threat scenarios (Section II.2.2): i.e. alternation of the static program binary and dynamical misuse of the expected program binary through its interface.

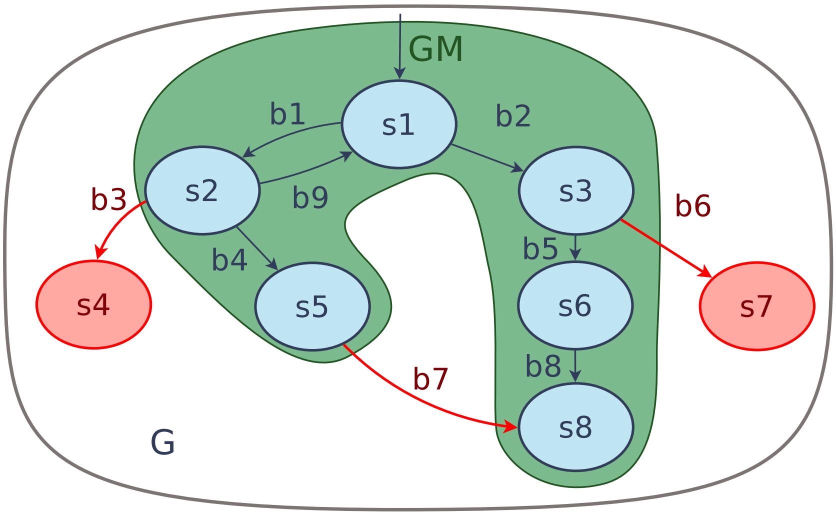


**Figure IV.1:** *A program execution graph*

We introduce an abstract representation of the execution of an application. Let $p$ be a program, we define its execution graph $G^{(p)} = (S^{(p)}, B^{(p)})$, where $G^{(p)}$ models the execution path of $p$ by a sequence of intermediate abstract states given in $S^{(p)}$. $B^{(p)}$ defines all possible transitions between two states in the graph. Figure IV.1 shows an example of graph, where threats are unauthorized transitions represented by bold red arrows. Let $G^{(p)}$ a graph with $N$ states and $M$ branches. We can define $S^{(p)}$ and $B^{(p)}$ as:

$$\begin{cases} S^{(p)} = \{s_i\}_{i=1..N} \\ B^{(p)} = \{b_k\}_{k=1..M}, \forall k \in [1, M].\exists i, j \in [1, N].b_k = (s_i, s_j) \end{cases} \tag{IV.1}$$

We introduce the notion of time with the sequence of state transitions executed by the program $p$ at runtime; we note $t \in \mathbb{N}^*$ a variable time and $T \in \mathbb{N}^*$ the time when the program $p$ returns (i.e. the last branch in the execution of $p$). We define $e^{(p)}(t)$ the execution path of the program $p$, from time $T_1 = 1$ (when the first branch is executed by $p$) until time $t$, as the sequence of branches executed by $p$ in $[1, t]$ (IV.2,IV.3). Hence, $\forall t \in \mathbb{N}^*.e^{(p)}(t) \subseteq B^{(p)}$.

$$p \text{ executing in } [1, T] \Leftrightarrow \forall t \in [1, T].e^{(p)}(t) = \{b_i\}_{i=1..t}, \forall i \in [1, t].b_i \in B^{(p)} \tag{IV.2}$$

$$p \text{ executing exclusively in } [1, T] \Rightarrow \forall t > T.e^{(p)}(t) = e^{(p)}(T) \tag{IV.3}$$

For a given program $p$, we introduce the golden model $GM^{(p)}$ to describe all trustworthy branches in $B^{(p)}$ (i.e. $GM^{(p)} \subseteq B^{(p)}$). The execution path integrity is guaranteed if and only if $\forall t \in \mathbb{N}^*.e^{(p)}(t) \subseteq GM^{(p)}$. We can then summarize our asset as the following invariant on the program execution at time $t \in \mathbb{N}^*$: $e^{(p)}(t) \subseteq GM^{(p)}$.

### Threat Formal Definition

We define a threat – or intrusion – as an anomaly in the program's execution, which compromises our security asset (i.e. the integrity of the execution for the program to protect). Our goal is to detect anomalies in the program's execution to prevent intrusions to occur.

Thus, for a given program $p$ executing exclusively in the interval $[1, T]$, a time $t \in \mathbb{N}^*$, and the execution path $e^{(p)}(t)$, an anomaly corresponds to a broken execution integrity invariant, as shown with (IV.4).

$$e^{(p)}(t) = \{b_i\}_{i=1..min(T,t)}, \exists k \in [1, min(T, t)].b_k \notin GM^{(p)} \tag{IV.4}$$

### Host Intrusion Detection Formal Goal

Our goal is to protect the system asset with runtime intrusion detection; the role of the HIDS is then to control that the program's execution path remains within the corresponding golden model. In this respect, the HIDS defines for a given program a model of execution, which must be as close as possible to the golden model of the program, to accurately identify unauthorized transitions between two execution states at runtime.

Let $p$ be a program corresponding to the execution graph $G^{(p)} = (S^{(p)}, B^{(p)})$, we introduce $M^{(p)}$ the model of the HIDS solution, so that $M^{(p)} \subseteq B^{(p)}$. To monitor the program's execution, the HIDS must be able to observe the program execution states at runtime. To this end, it can leverage one or several system signals – as discussed in Chapter III – to detect corruption attempts on our security asset. The HIDS can control the asset recursively at runtime following Definition (IV.5) and Induction (IV.6): each transition between two states is compared to the golden model, while the preceding path has already been checked.

$$\begin{cases} e^{(p)}(0) = \emptyset \\ e^{(p)}(t) = \{b_i\}_{i=1..t} = e^{(p)}(t-1) \cap \{b_t\}, \forall t \in [1, T] \end{cases} \tag{IV.5}$$

$$\forall t \in [1, T].e^{(p)}(t) \subseteq GM^{(p)} \Rightarrow e^{(p)}(t-1) \subseteq GM^{(p)} \wedge b_t \in GM^{(p)} \tag{IV.6}$$

### Program Execution Graph Coverage

Ideally for a given program to protect, the HIDS model of execution covers the exact golden model of this program: i.e. the HIDS identifies all transitions located inside the golden model as correct, all others are potential threats. Let $p$ be a program corresponding to the execution graph $G^{(p)} = (S^{(p)}, B^{(p)})$, $M^{(p)}$ the model of our HIDS solution: our target is to achieve $M^{(p)} = GM^{(p)}$.

However in practical implementations, a HIDS solution usually introduces errors of detection. In such case, the HIDS model was not correctly defined prior to deployment. On one hand, a HIDS solution possibly induces false-positives in the detection: i.e. the HIDS detects – with respect to its model of program execution – an anomaly which does not maliciously alter the program's execution path. In such configuration, the HIDS model does not fully include the program's golden model: i.e. $GM^{(p)} \nsubseteq M^{(p)}$. On the other hand, the HIDS may also miss the detection of actual intrusions: i.e. $M^{(p)} \nsubseteq GM^{(p)}$.

## IV.2.2 System Events Selection for Intrusion Detection

### From Abstract States to Observable Traces

We represent the abstract states of the monitoree's execution with relevant observable system signals at runtime, as developed in literature review (Chapter III). In our view of system integrator, the monitoree is a black-box: we have limited knowledge on the program (e.g. no access to the source code) and restricted instrumentation capability, especially in the context of certified critical applications.

Thus on one hand, we decide to privilege system events which we can monitor with low intrusiveness on the monitoree's interface and system software lower layers. In particular, we target hardware events located on the processor where the monitoree is executing: i.e. HPC and processor tracing features. On the other hand, we prioritize solutions which can run from user-space for certification reasons: a modification in the OS kernel would require to reiterate the certification process on the whole kernel, to assure that the add-ons do not disturb other system critical functionalities.

### Correlation between Intrusions and System Signals

The two threat scenarios (execution of a statically modified binary and dynamic misuse of the program), which we consider in Section II.2.2, imply an alteration of the CF of the application at runtime. In this context, the system signals related to the instructions executed by the monitoree represent a relevant source of information to observe the program's runtime CF: e.g. sequence of instructions, time per instruction, type of instructions, etc. For example, CFI checking solutions monitor CF transitions to protect a program's execution against code-reuse attack (Section III.2.1).

A wide variety of intrusions exploits hardware vulnerabilities related to hardware optimizations, like cache memory and processor extensions. HIDS can leverage such characteristics to identify real threats: for example side-channel attacks are likely to generate heavy flows of specific instructions (e.g. cache flush, branches, ...). Therefore, we propose to use HPC for intrusion detection. We introduce the following considerations to select appropriate HPC events for threat detection:

- the diversity of events across the scope of hardware platforms; even though HPC are available on the majority of common processor architectures (i.e. Intel, PowerPC, ARM), the set of available HPC event types differs among hardware platforms.

- the direct dependencies between some HPC events: e.g. the count of executed instructions includes the count of executed load instructions.

- the relevance of a given counter for threat detection can fluctuate among different monitored programs.

Hence, the selection of relevant HPC events for threat identification, which do not degrade the detection by adding noise, represents a key issue for building HPC based HIDS. According to literature (Chapter III), system calls represent an alternative valuable information source for intrusion detection. The trace configuration depends on the analysis approach and the RTOS API.

## IV.3 Two Safety-Aware Anomaly Detection Approaches for Embedded Mixed-Criticality Systems

We propose a generic approach for anomaly based intrusion detection, decomposing the monitoring solution into two phases:

- **trace collection:** can run at several system levels; from user-space or kernel-space, with hardware assistance.

- **trace analysis:** runs inside a dedicated partition in user-space.

We describe our HIDS architecture and its integration into an embedded mixed-criticality environment in Section IV.4. The rest of this section outlines the two approaches we develop in this thesis.

### IV.3.1 System Events Based Machine-Learning Assisted Anomaly Detection

We study anomaly based HIDS, which use ML trace analysis in Chapter V. For this, we leverage known system trace sources, following the solutions described in Section III.2. On one hand, the goal is to limit the intrusiveness for tracing the monitoree execution. On the other hand, we aim at evaluating the impact of such approaches on the system real-time performance and detection efficiency.

We develop two anomaly detection based approaches, which leverage ML methods to detect anomalous monitoree executions:

- **Offline ML assisted ADS based on system calls and HPC:** This is to our knowledge, the first framework to combine RTOS level and hardware level trace types to monitor the execution. We develop a safety-aware system call instrumentation framework and evaluate the corresponding performance overhead. We also evaluate how adding HPC values impacts the detection accuracy of the solution. Our contribution on safety-aware system call instrumentation led to a publication [80], while we introduced the concept of system call and HPC based monitoring in another paper [81].

- **Online ML assisted ADS based on HPC:** We provide a safety-aware HPC based ADS framework; we propose a method to integrate a ML framework for online detection and implement a proof of concept for evaluating the performance of the solution. This work was conduced in the context of a master thesis [109].

These anomaly based solutions are suitable for light-weight passive monitoring into an embedded MCS. However, because there is no guarantee for the detectors to be free from false-positives, the detection of an anomaly cannot be directly involved in chains of critical decisions in MCS. Specification solutions such as CFI monitoring, as described in the next section, are comparatively more suitable for deployment in critical environments. Though, we could combine both types of HIDS methods, to improve the security of the monitoree: for example, upon detection of an anomaly by the ADS, the system could enable further security checks processed by specification based methods.

### IV.3.2 Safety-Aware Hardware-Assisted Control-Flow Integrity Monitoring

We explore a specification-based runtime threat detection method with CFI monitoring. Our literature review has shown that few works [156, 127] already implement CFI monitoring for systems with real-time constraints (Section III.3). These solutions require software instrumentation, which limits their applicability to an industrial use-case involving system certification; i.e. instrumentation of the monitored program' source code or of kernel functions.

Recent research has introduced the use of hardware-assisted CF tracing for CFI monitoring into industrial real-time systems, since such implementation does not require instrumentation of the monitored program. However, CF tracing brings a high performance overhead for monitoring, which depends on the monitored program execution path (the more executed CF transitions, the higher the overhead). On one hand to integrate the security service in a MCS, we must consider the worst-case monitoring overhead. On the other hand, this assumption is highly pessimistic for a practical deployment.

Hence, we propose a first safety-aware method to integrate hardware-assisted CF based security monitoring with ARM CoreSight into a MCS, and metrics to evaluate the trade-off between performance impact and security monitoring coverage. Our security framework combines a predictable CF transition level monitoring with trace collection that can be used for CFI checking, together with an anomaly detection service to monitor the full program execution. We validate our approach on an industrial MCS platform with ARM CoreSight support, using a set of programs from TACLeBench benchmark.

We detail the solution further In Chapter VI we detail further the solution, which led to two publications: in the first we introduce the hardware-assisted CFI monitoring framework [92], while in the second we propose a practical safety-aware method to integrate the solution into a MCS [79].

### IV.3.3 Comparison of the Two Monitoring Approaches

Table IV.1 compares the ML based anomaly detection approach with the CFI monitoring technique for a deployment into embedded MCS. Our evaluation considers three main criteria: detection type, portability, and transparency.

**Table IV.1:** *Comparison of intrusion detection solutions introduced in this thesis. Symbols* ×*and* ●*respectively designate unsupported and supported fields. Symbol* ◗*refers to a field, which can be supported depending on the deployment configuration.*

| **Solution** | | **No FP** | **Portability** | **Transparency** | | |
|---|---|---|---|---|---|---|
| | | | | no runtime intrusiveness | no source code access | no kernel modification |
| ML | Chap. V | × | ● | ◗ | ● | ◗ |
| CFI | Chap. VI | ● | × | × | × | ● |

First, the detection scope differs for both approaches. On one hand, CFI monitoring is a specification based method; thus, every detected intrusion corresponds to an actual security threat (i.e. no false-positives). On the other hand, machine-learning assisted solutions can induce false-positives.

Second, as we leverage hardware assistance to limit intrusiveness in the execution of the monitoree, the portability of the solution particularly depends on the necessary hardware requirements. Our hardware-assisted CFI monitoring is based on ARM CoreSight technology to transparently trace specific instructions executed by the monitoree. Even though similar technologies are available on other processor architectures (e.g. Intel Processor Tracing [74] for Intel architecture), we expect that the porting of our solution to another processor architecture is not straightforward. Alternatively, we monitor low-level system events – i.e. HPC and system calls – for ML assisted monitoring. On one hand, our system call monitoring framework cannot directly be deployed into another OS (e.g. Linux), as we run our experiments using SYSGO's PikeOS hypervisor. On the other hand, we can comparatively more easily adapt our mechanisms to monitor HPC in systems based on different processor architectures:

- We consider generic HPC events, which are traced on various processor architectures like ARM, PowerPC, and Intel.
- The tracing of HPC counters is relatively simple; it corresponds to reading and writing to predefined CPU registers.

Finally, we evaluate the transparency of our solutions following 3 criteria:

- **no runtime intrusiveness:** The monitoring framework induces no runtime intrusiveness for the monitoree's execution. The monitoree cannot be preempted by monitoring tasks.

  Our CFI monitoring framework must be able to pause the monitoree to monitor all generated CF traces. The system call monitoring solution induces indirect delays in the monitoree's execution, corresponding to instrumentation time for each monitored system call of the monitoree. On the contrary, the HPC based solution comes with no runtime intrusiveness after initialization; the solution can run concurrently with the monitoree.

- **no source code access:** The monitoring framework does not require access to application source code to generate metadata used at runtime to detect intrusions.

  Our CFI monitoring framework requires source code access to check destination addresses of executed branches; it leverages compile-time metadata of the monitoree program.

- **no kernel modification:** The solution can fully run from the user space; the implementation does not induce any modification of the time characteristics of kernel routines.

  The CFI monitoring solution can be fully implemented at user-level. For the ML-assisted solutions which we introduce, HPC monitoring can run fully at user-level but requires initialization at kernel level for security reason; obviously, system call monitoring requires a kernel driver to instrument system call entry and exit routines.

## IV.4 Host Intrusion Detection Integration into an Embedded Mixed-Criticality Environment

We present a general approach to integrate our HIDS solutions into an embedded mixed-criticality system. Section IV.4.1 describes the main components of our system architecture and clarifies how they address the specific environment constraints; in particular with the local deployment of the HIDS on the monitored platform, the system architecture must ensure the trustability of the HIDS. In Section IV.4.2, we discuss how the integration of the HIDS impacts safety-criticality and certification constraints. Finally, even though the problem of system recovery upon a detected intrusion is beyond our scope of work, we provide a list of strategies which can be applied depending on the context of execution in Section IV.4.3.

### IV.4.1 System Design

Figure IV.2 provides an overview of the generic system architecture we propose to integrate the HIDS. We simplify the figure to only represent the software components involved in the monitoring process.

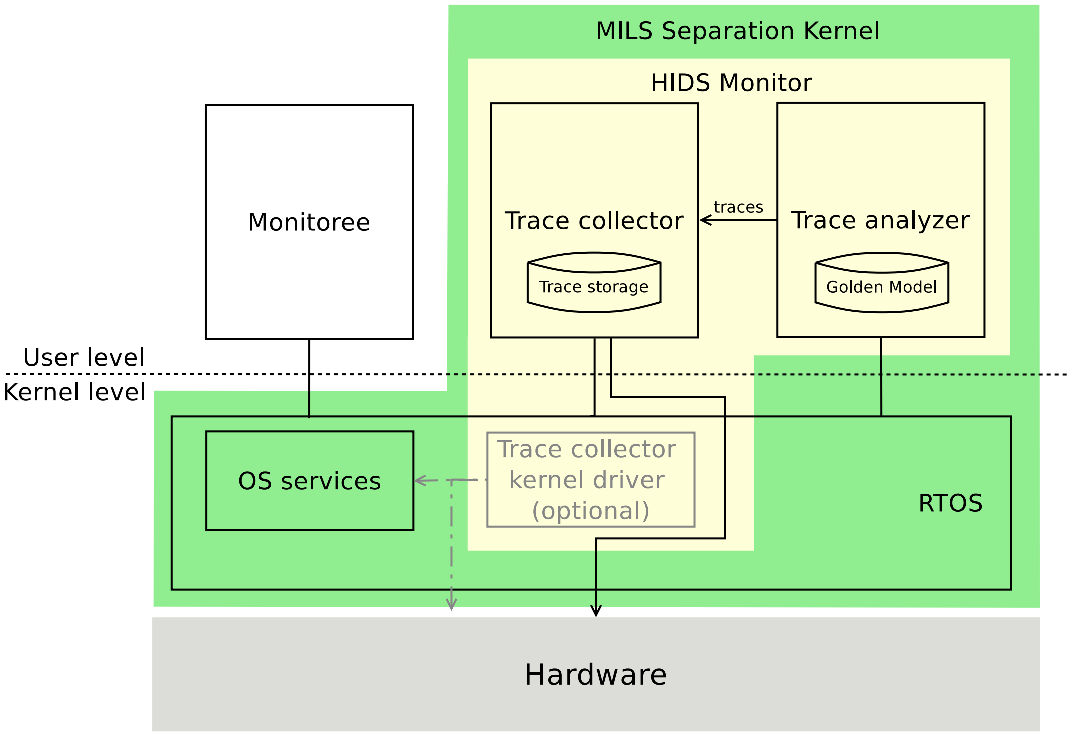


**Figure IV.2:** *Generic system architecture supporting host intrusion detection system*

### Implementation Constraints

Our goal is to monitor the execution of the monitoree at system runtime: the HIDS monitor must be deployed locally, on the system running the monitoree program. Consequently, the system architecture must ensure that potential adversaries, – especially ones controlling the monitored program – are not able to manipulate monitoring functions. To this end, our approach leverages resources separation to assure freedom from interference and independence between the HIDS monitor and other user-level tasks, which may be exposed to attacks.

Regarding the monitoree, since we have as system integrator limited access to the source code and limited rights to modify the binary, our HIDS must cope with limited instrumentation capability. We privilege running HIDS monitoring from user-level to avoid modifications in the OS kernel, which would require to reiterate the certification process of the kernel (details in Section IV.4.2). Due to the embedded characteristics of the deployment environment, we target a light-weight HIDS implementation to fit the limited performance capability. We intend to use generic software and hardware features to facilitate the portability of our solution.

### Monitoree

As introduced in our system asset definition (Section II.2.3), the monitoree is a business logic program running in user-space, potentially provided by an external agent. As system integrator, we receive the monitoree as a black-box program. In that respect, the

HIDS trace collection must be transparent, so that it does not require modification of the monitored program. Our approach is to observe the monitoree's execution through its system interface (e.g. hardware usage, type of executed instructions) at runtime to detect malicious execution behaviors.

An exhaustive HIDS model to characterize the monitoree's execution is crucial to build an accurate intrusion detection system; in theory (Section IV.2.1), the closer the HIDS model is to the monitoree's golden model, the better the intrusion detection quality becomes. The precision of the HIDS model depends on the type of data to be traced for monitoring and the type of analysis to distinguish normal from anomalous execution paths. On one hand for statistical and AI based analyses, the HIDS requires a representative set of traces to build the monitoree's model of execution under normal conditions. On the other hand, for specification-based analyses – in particular with static analysis – we need information on the program characteristics: e.g. set of authorized CF transitions for CFI monitoring. In such case, as we have limited access to the monitoree software, we could provide the application provider with the static analysis program, so that s.he returns the corresponding output metadata necessary for our HIDS analysis.

With specification-based analyses, it can be possible to evaluate the coverage of the HIDS model analytically. Though in many cases, especially for statistical and AI based methods, the evaluation of the HIDS model requires traces of the monitoree's execution under threat. As it is not trivial to implement representative program intrusions, the Common Vulnerability and Exposure database [37] maintained by the MITRE organization can help as a reference baseline.

### Separation Kernel

MILS [152] architecture protects the interfaces between the different system layers: user-level programs, kernel services, and hardware components. The MILS Separation Kernel (SK) ensures the strong separation of the user-level software in confined enclaves and provides secure communication channels. It contains RTOS services and drivers at kernel level. As system integrator and OS provider, we assume that the SK domain is trustworthy; i.e. kernel level services run as expected.

The SK includes the HIDS monitor running in user-space. Because we deploy the intrusion detection solution locally on the monitored target, the HIDS monitor could be vulnerable to adversaries with access to the platform (e.g. physical hardware, some user-level applications): for the intrusion detection to be trustworthy, the attacker should not be able to access the HIDS. In this architecture, the HIDS monitor's execution properties – i.e. such as confidentiality, integrity, and availability – are protected by MILS architecture. Hence, we safely and securely integrate the HIDS monitor locally on the monitored system using MILS architecture and SK inherent properties, to guarantee freedom from interference and independence between user-level components.

### Host Intrusion Detection System

In the first building phase, the HIDS defines a reference model of legitimate execution for the monitoree, which depends on the analysis type: e.g. static code analysis, statistical based approach. In the second runtime phase, it continuously compares upcoming traces of the monitoree's execution with the reference to identify benign from malicious execution paths. Hence, we decompose the HIDS monitor as detailed on Figure IV.2, between:

- **Trace collection:** a user-level application, possibly coupled with a kernel driver for kernel-restricted operations.
- **Trace analysis:** a user-level application.

Both services run privilege operations: adversarial manipulations, like trace falsification and reference model alteration, on one of these HIDS functions must be prohibited by our system architecture. Thus, we integrate the HIDS to MILS SK.

As system integrator, we base our trace collection function on standard hardware features available on common COTS platforms. Recent research has demonstrated the practicability of hardware vulnerabilities like Spectre [87] and Meltdown [100] as well as cache based side-channels. Therefore, we intend to separate the monitor and monitoree enclaves on the hardware. On one hand, the analysis only runs on a separate set of CPU cores; on the other hand, some tracing strategies require to run on the same core like for system call tracing.

We split trace collection in 3 steps:

1 **generating traces:** the HIDS generates the traces from the different selected sources.

2 **storing traces:** the HIDS stores generated traces in a dedicated memory area.

3 **providing traces:** the HIDS provides stored traces to the analysis application upon request.

For HIDS security, the trace collector must have the exclusive access to the tracing hardware, so that no adversary controlling the monitoree can manipulate its execution. For example, HPC must be unwritable from user-space to prevent corruption attempts initiated by an attacker running in user space; HPC configuration should be allowed from kernel-space (in a specific HIDS driver).

The trace analyzer takes the traces from the trace collector and analyzes them using the reference model defined in a building phase prior to the actual deployment. The analyzer can apply recovering strategies, depending on the context of execution, for instance restarting the monitoree application. First, we can decorrelate this data analysis task from tracing time, using trace collector's internal storage. Second, depending on the implementation complexity, it can require the use of a dedicated AI framework. For these two reasons and to prevent any safety impact, the trace analyzer runs in user space inside an isolated partition.

Thus, the interaction between the HIDS monitor and the monitoree is unidirectional, from the HIDS to the monitoree application through the kernel which is secure by design. An attacker controlling the monitoree has no direct access to the HIDS. Finally to limit interferences, the trace analyzer runs on reserved hardware (i.e. dedicated memory region and set of CPU cores). Similarly, we implement the trace collector on separate hardware when this is possible. This architecture also limits the impact on the monitoree's execution – i.e. with a low effect on the scheduling scheme determining the monitoree's execution – as long as the system resources are not needed by the original schedule.

## IV.4.2 Safety-Aware Integration of the Host Intrusion Detection System

### Impact on System Certification for Integrating the Host Intrusion Detection System

Because of its transparency (i.e. it does not instrument the monitored program), our HIDS does not directly affect the certification process of the monitored program. Though, we must reevaluate our system certification when we integrate new kernel add-ons. This induces the following system certification actions:

- **re-iterate kernel time analysis:** This corresponds to two steps which we typically must perform manually:

  1 **kernel worst-case path analysis:** Because of the new-delays introduced by the kernel modifications, we must analyze the impact on the maximum time spent in the kernel.

  2 **estimation of the new WCET in the kernel:** This corresponds to computing all WCET, for kernel functions and the worst-case path in the kernel. If the new WCET is too large for the given deployment use-case, we may have to adapt the monitoring add-on implementation to reduce its impact on system time properties.

- **update system tests:** The kernel source code must be fully covered by the test suite. Hence, we must introduce additional unitary and integration tests to cover kernel-level HIDS software. We then must run all tests; this second step can be automated, while the tests generally require – at least partial – manual implementation.

- **update traceability process:** The whole source code must be covered by the specification. Thus, we need to create formal requirements and link them with the source code and additional tests; this can be automated.

In addition, if the intrusion detection information is involved in further runtime processes which require certification, the HIDS component itself must be certified, so that it is able to detect threats consistently. For example, the HIDS provides the detection result within a maximum time limit and the result is justified from the monitored traces and associated with a defined accuracy level. The certification process of a system component follows the V cycle scheme, which describes the development lifecycle. For example, the avionics standard DO178-C [135] introduces the different aspects involved, which can be used in other safety-related standards:

- **specification:** The specification defines the functional requirements to then describes the component's design. We can perform this stage with semi or fully formal methods.

- **requirements-based testing:** The unitary and integration tests must cover the requirements of the component. It is a manual process.

- **coverage analysis:** This is the evaluation of the structural coverage of system requirements by the component based on the tests; every line of source code must be covered by the tests. This process can be automated.

- **timing analysis:** Using supporting tools, this stage corresponds to WCET analysis of the different component's functions, including stack analysis based on WC call tree evaluation.

- **partition analysis:** The partition covers an application and its corresponding system resources. Partition analysis is a manual process, which represents an inherent part of the MCS design. It assures the independence between applications; i.e. the absence of interference in the system is required to guarantee the coexistence of multiple independent criticality levels at runtime.

- **documentation:** The documentation of all the processes above to support the traceability of the component's development, from the specification to the deployment.

### Schedulability Analysis Integrating Host Intrusion Detection System

We need to update the schedulability analysis of the system to include HIDS monitoring. For tracing as well as for analyzing the monitoree at runtime, the implementation constraints vary across the range of possible approaches, inducing context-dependent impact on system schedulability. For example, the HIDS can require the ability to stop the monitoree to analyze on the fly heavy flow of generated traces, which implies an impact on the timing properties of the system.

Generally, if the HIDS monitor executes on the same CPU core than the monitoree, the system integrator must reiterate the schedulability analysis for the system including HIDS tasks involved. In this case, monitoring tasks' priority must be higher than the monitoree task's priority, but lower than other tasks with higher priority compared to the monitoree. When this is possible, we tend to separate the execution of monitoring and monitored applications, executing them on dedicated CPU cores. In such configuration, aside from the trace collection overhead induced in the monitoree, no further overhead is induced on the application cores: schedulability properties are preserved. In Sections IV.5.2 and IV.5.3, we discuss how to evaluate the HIDS integration impact on system performance and safety properties.

### IV.4.3 System Recovery Strategies

As previously explained, we do not address extensively the reaction of the system upon detection of an anomaly in the monitoree's execution; the decision is indeed highly context-dependent and use-case driven. It depends on two main aspects:

- the criticality level of the monitored application, and more broadly of the criticality level of system.
- the reliability of the detection: from null to high false-positive rate.

In our context of work, we consider the following recovery strategies to apply for attack mitigation:

- **Signal/log:** whenever there is high uncertainty or the risk is too important, the HIDS notifies the system administrator of a potential threat.
- **Suspend/kill/reboot:** the HIDS interrupt the application running the intrusion.
- **Isolate:** the HIDS limits the interactions with other system components to avoid any attack propagation.
- **Migrate:** this is usually not possible in a critical environment, where every user-level application is configured statically.

Several industrial standards specify the measures to apply in the context of critical applications, notably by introducing a health-monitoring component:

- **ARINC-653 (Avionics Application Standard Software Interface) software specification:** "In general, Health Monitor (HM) functions are responsible for responding to and reporting hardware, application, and O/S software errors and failures. [...] The HM helps to isolate errors and to prevent failures from propagating." [138].

  configuratiocalculation,service call
- **AUTOSAR (AUTomotive Open System ARchitecture) specification:** "The Health Monitoring is intended to supervise the execution of supervised entities with respect to timing constraints (alive and deadline supervision) and with respect to the required sequence of execution (logical supervision) and with respect to their health (health supervision)" [17]. It involves among others support for hardware watchdogs and error handling mechanisms.

## IV.5 Host Intrusion Detection Framework Evaluation

We propose 3 main directions with a set of corresponding metrics to evaluate our HIDS framework for deployment in embedded MCS:

- **intrusion detection efficiency:** the ability for the solution to early and accurately detect intrusions.
- **safety requirements guarantee:** the conditions for the solution to guarantee system safety requirements at runtime.
- **performance impact:** the set of resources (hardware devices, memory storage, computing power, ...) required to run the intrusion detection solution, as well as the impact of monitoring on the execution of the monitored software's runtime execution.

## IV.5.1 Intrusion Detection Efficiency Evaluation

### Intrusion Detection Efficiency

An efficient HIDS solution is able to detect runtime intrusions rapidly and accurately. First, to be relevant, an HIDS must be reliable enough; this means it should minimize false alerts, while maximizing the detection of actual threats. We propose to evaluate the solution using the confusion matrix (Table II.1) introduced in Section II.1.3, to analyze detection errors (false-positives and false-negatives).

### Ability to Detect Zero-Day Attacks

Our approach focuses on anomaly-based HIDS solutions in opposition to signature-based approaches (Chapter III): we intend to detect any anomalous execution of the monitored program at runtime, i.e. including unknown attacks. Our goal is then to build for the monitoree a model of normal execution to detect anomalous paths of execution.

We need suitable metrics to define the granularity of the solution for threat detection. We base the evaluation on relevant anomalous scenarios of execution representing actual intrusions. On one hand, we propose to simulate real threat scenarios, to evaluate how efficiently the HIDS detects them. On the other hand, we use anomalous program executions, derived from the normal program execution, to assess the HIDS detection granularity.

### Security Monitoring Coverage

The quality of the HIDS framework depends also on its coverage of the monitoree's CF execution at runtime. We define two main criteria to characterize the coverage of the solution:

- **the monitoring scope:** the range of operations covered by the solution; i.e. all CF transitions or only a subset of operations, which are possibly more sensitive to threats (e.g. system calls, return instructions).
- **the monitoring continuity:** A full protection covers the whole execution of the monitored program, while a discontinuous solution selects a partial set of system events to monitor. A discontinuous HIDS service could handle intermittent

monitoring checks, periodically or at strategic times in the execution when attacks are more likely to occur.

We illustrate in Figure IV.3 the main metrics we introduce to evaluate HIDS coverage in respect of time windows of opportunity for an attack:

- **Maximum detection time ($w_{detect}$):** the maximum duration from the moment the attack starts, until it is detected by the HIDS, which generates an alert. We calculate it with the sampling period for the HIDS to generate and collect the traces and the time for the HIDS to process the set of traces:
- **Maximum uncovered execution time ($w_{uncov}$):** the maximum time duration when HIDS monitoring is inactive while the monitoree is executing. A successful intrusion on the monitoree fitting this time window cannot be detected by the HIDS.
- **Total uncovered execution time of the monitoree task $\tau_A$ ($C_A^{(uncov)}$):** the total time of execution of the monitoree when HIDS monitoring is inactive. This indicator corresponds to the maximum time a knowledgeable adversary can exploit to dissimulate an attack in discontinuous time windows when the HIDS is inactive during the execution of the monitored application.

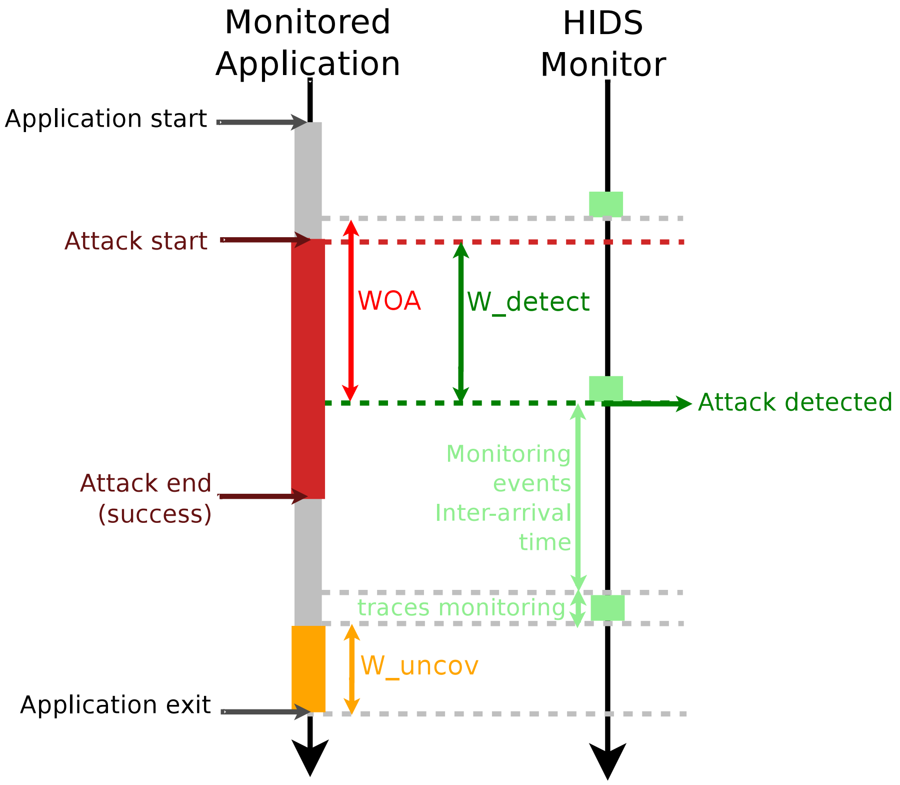


**Figure IV.3:** *Sequence diagram of the execution of an application under HIDS monitoring*

## IV.5.2 Performance Impact Evaluation for Monitoring

### Impact on System Requirements

As discussed in Section IV.4.1, we isolate the HIDS monitor from the monitoree on the platform to limit possible interference between both programs, which could be exploited by an adversary. At software level, this system configuration requires a dedicated user-level partition for the HIDS monitor. At hardware level, this separation induces additional infrastructure costs:

- a dedicated set of CPU cores for monitoring to facilitate HIDS integration, notably for reducing the impact on system schedulability.
- a dedicated memory region for the new HIDS monitor user-level partition.
- optionally, an exclusive access to the hardware possibly needed for trace collection or trace analysis.

Finally, we can expect an overhead in the system binary size for monitoring, which corresponds to:

- the trace collection executable (HIDS trace collector partition)
- the trace analysis executable (HIDS trace analyzer partition)
- the static reference model used by the HIDS analyzer

### Performance Overheads for Monitoring

Our main metric to evaluate the performance overhead is the runtime slowdown of the monitoree's execution under monitoring. This slowdown does not include the trace analysis execution, since the HIDS trace analyzer and the monitoree can run in parallel (i.e. they do not share CPU resources and have no direct dependencies). Hence, the slowdown is null if trace collection can run without interrupting the monitored program, which can be possible for hardware-based tracing solutions.

Let define an application $A$ with the WCET $C_A$ and $C_A^{(monitoring)}$ the WCET when monitoring is enabled and running. We define the application slowdown as the ratio $r_A^{(monitoring)}$:

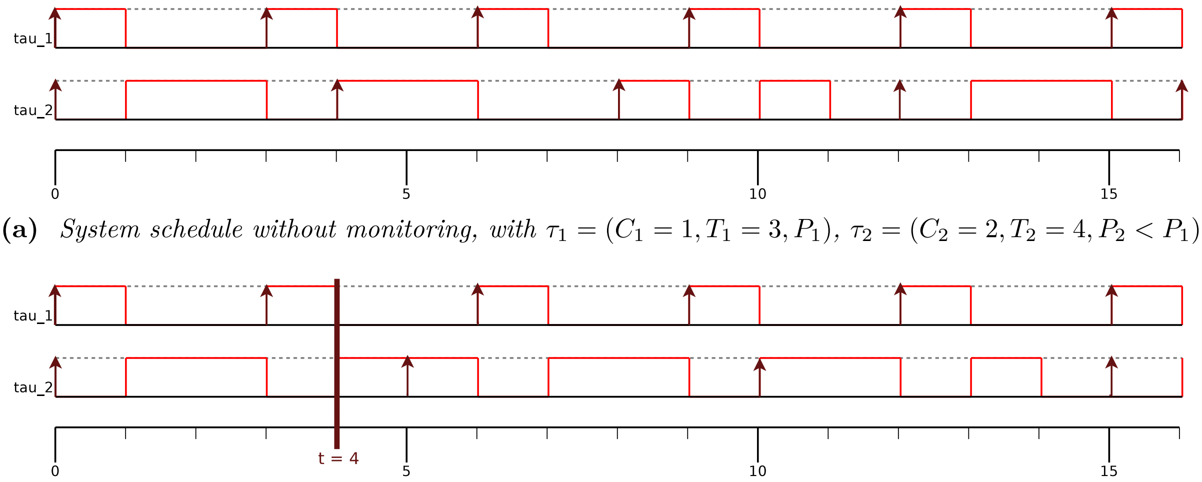
$$r_A^{(monitoring)} = \frac{C_A^{(monitoring)}}{C_A} - 1 \tag{IV.7}$$

If the HIDS solution requires kernel extensions for trace collection, it can induce additional slowdowns affecting some kernel functionalities; these need to be evaluated, to understand the impact for other user-level applications running aside from the monitoree and the HIDS monitor in user-space.

We also evaluate our HIDS framework in respect of memory costs for runtime trace storage. The trace analyzer must consume traces at least as fast as the trace collector produces them. We must assure the synchronization between both producer and consumers to avoid loosing traces. However, if the trace analyzer consumes the traces too slowly compared to the trace generation rate, we can apply different strategies depending on the context:

- stopping the monitored program to pause trace collection, while the trace analyzer is processing generated traces. In this case, the monitored program slowdown must include this additional parameter in the computation.
- configuring sufficient adequate storage for storing all traces at runtime (i.e. the HIDS does not interrupt the monitoree).
- deciding to drop some of the traces to allow a partial coverage of the monitored execution.

### IV.5.3 Safety Requirement Guarantee Evaluation



**(a)** *System schedule without monitoring, with* $\tau_1 = (C_1 = 1, T_1 = 3, P_1)$, $\tau_2 = (C_2 = 2, T_2 = 4, P_2 < P_1)$

**(b)** *System schedule with monitoring, with* $\tau_1 = (C_1 = 1, T_1 = 3, P_1)$, $\tau_2 = (C_2 = 3, T_2 = 5, P_2 < P_1)$

**Figure IV.4:** *Example of system schedules, with and without monitoring*

To perform HIDS monitoring into an embedded MCS, we must guarantee that real-time constraints are maintained while we integrate the solution in the system. Hence, we propose to follow an analytical approach to demonstrate system schedulability.

To limit the impact of the execution of the HIDS on the monitoree, we try to run the HIDS from user-space when no inherent constraint opposes such approach: the certification of the OS kernel – corresponding to the highest certification level in the system – requires the WCET computations to include any add-on in the kernel. For the same reason, we intend to isolate the HIDS on the hardware platform; in particular to avoid adding new monitoring tasks in the system schedule involving the monitoree.

However, the solution may have a timing impact on the monitoree. Figures IV.4 show an example of such issue using two tasks with implicit deadlines running with a preemptive and fixed-priority scheduler policy. Figure IV.4a corresponds to the system without monitoring, while Figure IV.4b shows the system when the task $\tau_2$ runs under monitoring. In the second monitoring case, the execution time of the monitored task increases due to the time overhead for monitoring: $r_2^{(monitoring)} = 1.5$. With the initial period $T_2 = 4$, the task $\tau_2$ would miss the first deadline at time $t = 4$. Therefore in such case, we must adapt the schedule to assure system schedulability; e.g. we increase the period $T_2$ in Figure IV.4b to maintain system schedulability with monitoring.

V

# Machine-Learning Based Anomaly Detection Solutions for Embedded Mixed-Criticality Systems

This chapter introduces anomaly based HIDS methods for embedded MCS, using system events which are common trace sources for intrusion detection in GPC (as discussed in Section III.2). Specifically, we aim at evaluating the use of ML for the trace analysis in our constrained environment. Our literature research highlighted that the majority of ML based HIDS solutions do not provide an evaluation of the impact on system performance for deployment of the framework. For these solutions, deployment constraints represent a secondary concern, compared to the accuracy of the ML engine to perform its assigned function (e.g. detection).

However, integrating a ML engine into an embedded MCS becomes challenging, when considering the deployment impact. On one hand, the more complex the ML model – i.e. increasing the amount of necessary computations – the better the precision of the model; i.e. the better the detection accuracy for an HIDS model. On the other hand, the more complex the ML model, the higher the deployment impact of the ML engine: first due to the limited hardware resources of the deployment platform, second because the latency to execute the ML engine has a direct impact on intrusion detection latency.

Even though several proprietary or open-source solutions, such as Tensorflow-Lite [119] in the past decade, provide a ML framework for embedded systems, we are not aware of any previous work, which studies the use of such adapted frameworks for deploying ML based intrusion detection into embedded MCS. Hence, in this chapter, we propose to address the trade-off between ML model complexity and deployment impact. For this, we introduce two safety-aware anomaly-based HIDS prototypes using ML to monitor an application running in an embedded MCS:

- **A HIDS based on system call and HPC tracing.** This is a novel hybrid approach based on HPC monitoring to contextualize system call trace collection. The framework is composed of two stages:
  - online system call instrumentation and HPC tracing.
  - offline trace analysis of the HIDS with ML.

The solution leverages safety-aware system call instrumentation for trace collection developed in Section V.1. We describe the framework in Section V.2 and evaluate the detection accuracy of the offline ML based trace analysis, using a set of test applications.

- **A transparent online HIDS based on HPC tracing.** This framework analyzes HPC traces periodically at runtime (Section V.3). We evaluate the impact for running ML based trace analysis on the embedded MCS platform. We trace HPC with no monitoree intrusiveness (i.e. on a different set of CPU cores than the monitoree). Contrary to the system call based approach, which only covers monitoree programs executing system calls, this solution addresses any user-level application, including system calls or not.

Section V.4 summarizes the benefits and pitfalls for the two HIDS approaches. In particular, both monitoring solutions consider the monitoree as a black-box: they do not require program instrumentation.

## V.1 System Calls Instrumentation for Anomaly Detection in Embedded Mixed-Criticality Systems

System call relative information such as occurrences, type, parameters, and return values are well established metrics to reveal intrusions in a system software. According to Section III.2.3, this type of defense offers good detection precision for both known and zero-day attacks; recent research focuses on HIDS deployment into GPC. Yet, the integration of such runtime monitoring solution in embedded MCS has not been discussed. Because of the cohabitation of potentially vulnerable non-critical software with critical software, securing such systems is a non trivial but essential issue. Adapting HIDS in this context is very challenging. One one hand, because it is deployed at system level, HIDS can potentially compromise system execution; e.g. a real-time application missing its deadline. On the other hand, HIDS can degrade time performance of non critical applications. Thus, we need to ensure that the add-on does not alter system execution and constraints.

To implement system call monitoring, we have to integrate system call tracing in the mixed-criticality OS kernel. The new feature must be fast and predictable; its impact on real-time system execution needs to be determined. In this section, we provide a study of predictable system call instrumentation framework to be applied for system call and HPC based anomaly detection using ML in Section V.2. We propose an evaluation of the feasibility of system call instrumentation for deployment in a MCS. After introducing our approach to evaluate the impact, we apply it in a concrete example, using the real-time hypervisor PikeOS [128].

Section V.1.1 introduces the methodology. Section V.1.3 describes the experiment, while Section V.1.4 presents the results. We discuss in Section V.1.5 the security impact of intrusion detection as well as further implementation of the detection, We also mention other interesting data to detect intrusions as well as their potential impact on system's real-time constraints. Finally, Section V.1.6 summarizes our findings.

### V.1.1 Methodology for System Call Monitoring in Mixed-Criticality Systems

We propose a methodology to evaluate the impact of intrusion detection based on system call monitoring in a MCS with real-time constraints. System call monitoring causes an additional delay in the kernel at system call handling, which induces two consequences: an impact on the monitored application's performance as well as an impact on system security.

#### Impact on Performance

The trace analysis runs remotely on a separate hardware; we describe the implementation in Section V.2). Consequently for this work, the performance impact corresponds to the time overhead for trace collection.

Let $\tau_A$ be the monitored application, with the corresponding execution times $C_A$ and $C_A^{(monitoring)}$ respectively without and with monitoring. Following Equation (IV.7), we define the time overhead as: $r_A^{(monitoring)} = \frac{C_A^{(monitoring)}}{C_A} - 1$.

A MCS is composed of several software corresponding to diverse criticality levels. Depending on the characteristics of the monitored application, the approach to evaluate the overhead $r_A^{(monitoring)}$ (i.e. to compute the execution time of $\tau_A$) varies; we distinguish two main types of applications:

- **critical application** (software executing under deterministic constraints, e.g. Avionics application): to use system call tracing in a critical application, the additional overhead must be added to the WCET of this application. We can calculate the maximal time overhead for monitoring in system call tracing with WCET analysis; the precise tracing overhead can be estimated for each system call in the kernel. Thus, we can compute the worst case overhead, corresponding to the worst system calls combination for the analyzed software.

- **non-critical application** (software with no deadlines, e.g. Linux OS): WCET analysis does not make much sense for non-critical application such as Linux OS: it would either be too pessimistic or non doable. We instead aim to analyze an average time overhead. However, because of the large diversity of software, it is not possible to get a representative single average value.

As other infrastructure costs, the system call instrumentation framework requires an additional user-level software partition to collect generated traces. Hence, following the measures described in methodology (Section IV.5.2), we assign a dedicated CPU core and reserved memory area for this additional partition to perform monitoring. We also consider a binary size overhead to contain trace collection software (i.e. kernel modification in system call handlers and additional user-level partition). As we only consider OS-level software events, we do not need further specific hardware features for tracing.

### Impact on System Security and Schedulability

In addition to the performance impact, the system call tracing time overhead located in the kernel can potentially compromise the entire system, by breaking time isolation between applications. An application raising a huge amount of system calls induces indeed a non-negligible delay in the kernel. Depending on the kernel implementation, we consider two scenarios:

1 - system call handling can be preempted by context switch.

2 - system call handling can not be preempted by context switch.

In the first case, even though the application induces a huge delay due to system call tracing, it is stopped by the ticker interrupt, responsible for context switching; the system call is possibly aborted. Thus, time isolation is respected.

In the second case, the monitored software can extend slightly its time window of execution, at the expense of other applications: time isolation is not respected anymore. An example of such issue consists in two applications running on the same single CPU core. The first is a non-critical application, while the second is a hard real-time application with firm deadlines. If the non-critical application issues huge amount of system calls, there are high chances that system context switch ticker interrupt is raised during system call handling. The kernel waits for the system call to return to the application before switching context to the next application. Thus, we can assess the maximal time overhead per context-switch to be the length of system call handling, including system call tracing. As the system is designed for guaranteeing time isolation without tracing, the time window of the non-critical application can be extended to a maximum value, which we introduce as $T_{OH}^{(max)} \in \mathbb{N}^*$. After a certain amount of context switches, the critical application possibly misses its deadline. Thus, we need to define the overall time overhead induced by system call instrumentation for typical applications, to be able to evaluate how it could compromise the system schedulability.

## V.1.2 Tracing Time Overhead Estimation

We propose to compute an average time overhead at runtime for a set of representative non-critical applications, with the following approach. We define the total time overhead $c_A^{(OH)} \in \mathbb{N}^*$ of the application $\tau_A$, so that $c_A^{(OH)} = C_A^{(monitoring)} - C_A$. Considering that $\tau_A$ executes a sequence of $N$ system calls ($N \in \mathbb{N}^*$), we deduce (V.1), where $\{t_i^{(OH)}\}_{i=1..N} \in \mathbb{N}^N$ corresponds to the sequential execution time overheads for tracing each of the $N$ executed system calls.

$$c_A^{(OH)} = \sum_{i=1}^{N} t_i^{(OH)} \tag{V.1}$$

The quantity $c_A^{(OH)}$ depends on the number of system calls and on their type. Thus, we apply the following process to estimate the average time overhead for the set of tested applications (represented by $\tau_A$):

1 - For every tested application, we count the amount of system call traces ($N$). Depending on the context of execution, the quantity $N$ can vary for two different executions.

2 - For a set of common system calls, we compute the time overhead caused by tracing. We can expect a small variation of tracing time between different system calls. The difference is caused by variations in the quantity of traced data caused by the number of parameters and return value. We finally define $(t_{min}^{(OH)}, t_{max}^{(OH)}) \in \mathbb{N}^2$, respectively the minimum and maximum time overhead values for tracing a single system call.

3 - Following Definition V.1, we can estimate the total tracing time overhead $c_A^{(OH)}$, simplifying it as a function of the time overhead for tracing one system call (V.2). In particular, $\forall t \in \mathbb{N} \cap [t_{min}^{(OH)}, t_{max}^{(OH)}], C_A^{(minOH)} \leq c_A^{(OH)} \leq C_A^{(maxOH)}$ with $C_A^{(minOH)} = N * t_{min}^{(OH)}$ and $C_A^{(maxOH)} = N * t_{max}^{(OH)}$.

4 - We can not precisely measure $C_A$ (the execution time of $\tau_A$ without tracing): the program execution varies, depending on the context of execution and regardless of system call tracing feature. Instead of $C_A$, we can measure $C_A^{(monitoring)}$ and estimate minimum, average, and maximum values of $c_A^{(OH)}$.

Hence, we redefine the ratio $r_A^{(monitoring)}$ (V.3); in the experiment (Section V.1.4), we later evaluate $r_A^{(monitoring)}(N, t, C_A^{(monitoring)})$ (V.4) in function of:

- $N$: the count of traced system calls
- $t$: the estimate of the time overhead for tracing one system call
- $C_A^{(monitoring)}$: the total execution time of the monitored application.

We measure both $N$ and $C_A^{(monitoring)}$ for every monitored execution of the monitoree. We compute minimum, average, and maximum values of $t$ to infer $r_A^{(monitoring)}$.

$$\forall t \in \mathbb{N} \cap [t_{min}^{(OH)}, t_{max}^{(OH)}], c_A^{(OH)}(t) = N * t \tag{V.2}$$

$$r_A^{(monitoring)} = \frac{C_A^{(monitoring)} - C_A}{C_A} = \frac{C_A^{(OH)}}{C_A^{(monitoring)} - C_A^{(OH)}} \tag{V.3}$$

$$\forall t \in \mathbb{N} \cap [t_{min}^{(OH)}, t_{max}^{(OH)}], r_A^{(monitoring)}(t) = \frac{N * t}{C_A^{(monitoring)} - N * t} \tag{V.4}$$

## V.1.3 Experiment

### System Design Overview

Figure V.1 details the system architecture, which is based on the generic solution introduced in Section IV.4. We run the experiment on PikeOS [128] OS. The experimental setup is composed as follows:

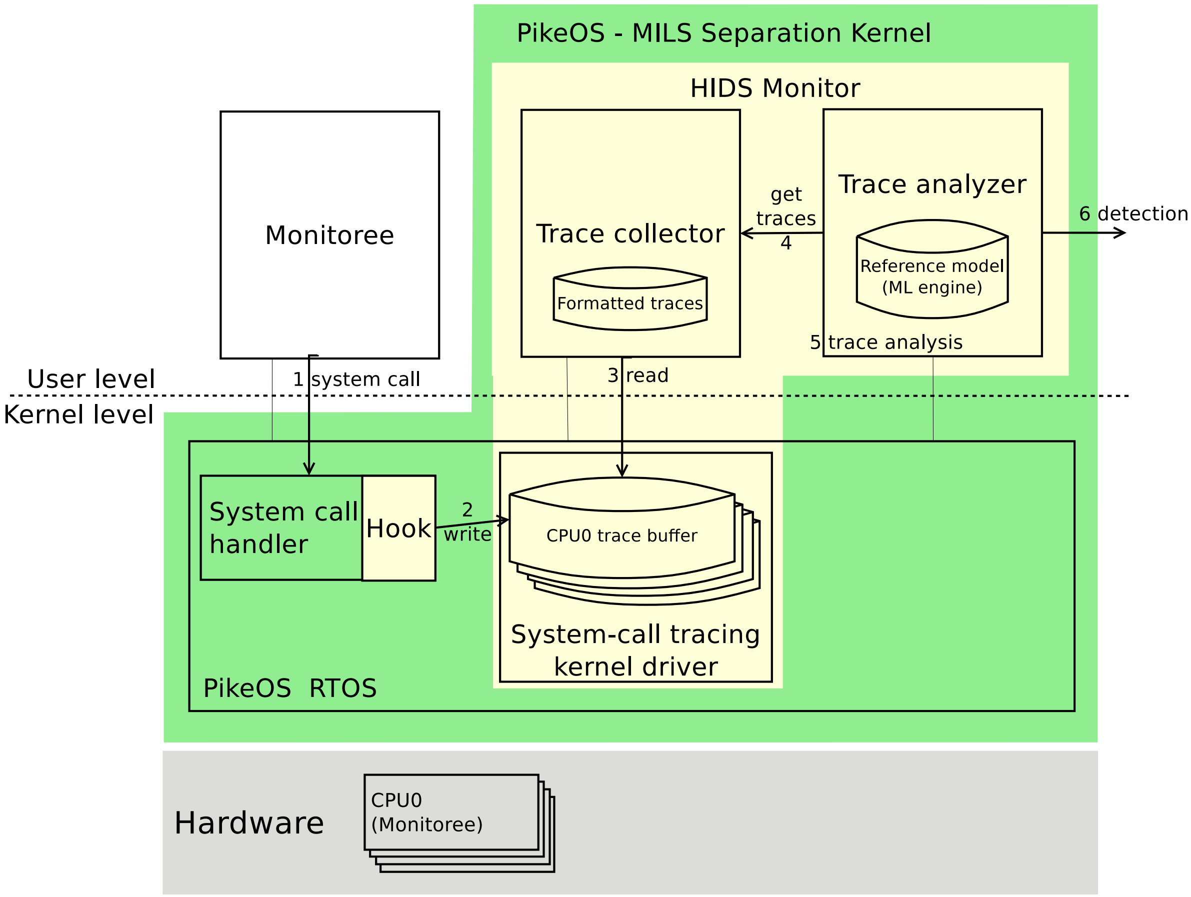


**Figure V.1:** *System architecture with system call monitoring*

- **system call trace store:** For each system call raised, the system call tracing driver stores in a per CPU core trace buffer, the two traces corresponding to the entry and exit hooks inserted in every system call handler:

  1 - **system call entry hook:** Once a system call is received, the entry hook extracts the system call type and its parameters and writes the first trace into the corresponding trace buffer.

  2 - **system call exit hook:** Before returning from the handler, the exit hook gets the return value and stores the second trace into the trace buffer.

- **system call trace restore:** The system call tracing driver provides the system call traces stored in the buffer upon request.

At user-level, monitored programs (i.e. Monitoree application) are defined in the test setup subsection below. In addition to the tracing kernel driver, the HIDS monitor is composed of two main user applications:

- **trace analyzer:** it contains the analytical model to identify a trace as footprint of an intrusion or as normal activity. We develop this part in later sections of this chapter.

- **trace collector:** it collects traces by communicating with the system call tracing driver. It formats and stores the traces in a protected memory area.

### System Call Tracing within PikeOS

The system call tracing process goes through the following steps (see Figure V.1):

1 - The monitoree raises a system call.

2 - The OS kernel handles the system call. From events raised by hooks in the handler, system call information are stored in the trace buffer of the initiating CPU core.

3 - At some point, the trace collector requests traces to the system call tracing driver. The driver reads the trace buffer and returns traces to the trace collector. Traces are then formatted and stored waiting for request from the trace analyzer.

4 - The trace analyzer requests a trace to the trace collector.

5 - The trace is analyzed with the reference model (i.e. using a predefined ML engine).

6 - In the case where an intrusion is detected by the engine, an alarm signal is raised (e.g. an alarm message is printed on the console).

Groups of operations (1,2), (3), (4,5,6) can run in parallel. Operations inside the groups are sequential.

The tracing feature induces an additional delay for system call handling in the kernel. It corresponds to the entry and exit hooks inserted in system call handlers. The hook consists mainly of a memory copy operation into the CPU trace buffer.

For the sake of clarity, we isolate the monitoree on a single dedicated CPU core (core 0). The trace collector runs on a distinct core (core 1).

### Limitations of the Implementation

At this stage, the solution does not monitor hardware virtualized guest OS. Such application indeed directly handles system calls, without any intervention of the hypervisor. Critical instructions raise hypervisor calls and are trapped in the kernel. Our solution could be adapted to support hardware-virtualized applications, by tracing these calls in the hypervisor kernel.

### Test Setup

The experiment is conduced on NXP board QorIQ LS1043A. Time is measured with the CNTVCT_EL0 CPU core counter. It corresponds to the virtual count for execution level 0 (user mode) of the running core and is accessible from user applications. The resolution of this counter is 40 ns. A timestamp is defined in the kernel for every trace before it is stored in the trace buffer. In our tests, we measure the time for running an application, by printing the counter value just before and after its execution. We get the unitary time for system call tracing in the kernel using the same method.

To represent $\tau_A$ the monitoree, we selected diverse applications from native and POSIX applications, as well as paravirtualized Linux. Native and POSIX servers are stimulated with a Linux client running `ping` process. For every test, we run measurements only once the system has booted. Native applications are C program binaries executing directly on PikeOS, using specific libraries. Ping server is a ICMP echo server which returns echo replies upon request. Shared memory client/server application corresponds to two applications, depending on runtime mode. Both applications share a memory buffer to exchange 32 text messages that are emitted by the server and read by the client, using synchronization mechanisms. The POSIX application inetd is a server daemon, which relies on a widely used TCP/IP stack called Lightweight-IP. The Linux software corresponds to a paravirtualized commercial embedded Linux for PikeOS, called ElinOS [53]. Our selection gathers well used processes and common performance benchmarks. Table V.1 lists command lines for all Linux applications.

**Table V.1:** *List of Linux applications and their applied command lines*

| Application | Command line |
|---|---|
| unixbench | `unixbench` |
| dd | `dd if=/dev/zero of=FILE count=CNT bs=BS` |
| ps | `ps -ef` |
| find | `find / > /dev/null 2&>1` |
| netserver | `netserver -4 -L ADDRESS -D -d` |
| netperf | `netperf -H NETSERVER` |
| Pacman | `pacman` |
| gzip | `gzip FILE` |
| iperf | `iperf -s` |
| ping | `ping ADDRESS -c 20` |

## V.1.4 Experiment Results

### Reliability of Measurements

Every test is reproduced five times. Average values of time and amount of collected system calls traces are represented in Table V.2. To evaluate the variability of results, the standard deviation is calculated and compared to the average value. Variability of system call count measurements is high in two situations:

- for applications which raise few system calls; e.g. native ping server and empty Linux OS.
- for short execution time; e.g. ps and dd Linux processes.

In this second category, time measurements are also less precise (approximately 7% of fluctuation).

### System Calls in Application Selection ($\tau_A$)

Table V.2 shows that the amount of system calls varies from 23 system calls to more than 200,000 for one second of application execution, excluding tracing time. The results illustrate that native applications, even with networking, raise very few system calls, compared to POSIX inetd daemon and Linux software.

**Table V.2:** *Results for the test application setups*

| **Application ($\tau_A$)** | | $C_A^{(monitoring)}$ **(s)** | $N$ **(calls/s)** | **var (%)** |
|---|---|---|---|---|
| Native | ping server | 60.45 | 23.2 | 41.16 |
| | ping server (busy) | 63.98 | 552.0 | 9.60 |
| | sh.mem. (client) | 31.47 | 410.0 | 0.00 |
| POSIX | inetd (no requests) | 30.08 | 10,508.0 | 2.75 |
| | inetd (ping requests) | 30.06 | 11,598.2 | 0.39 |
| Linux | empty | 60.34 | 26.7 | 8.80 |
| | dd (64 MiB) | 0.11 | 34,000.0 | 9.57 |
| | dd (128 MiB) | 0.22 | 19,398.1 | 7.52 |
| | ps -ef | 0.01 | 225,876.8 | 12.10 |
| | find / | 0.31 | 115,113.0 | 5.86 |
| | gzip (128 MiB) | 3.48 | 382.0 | 2.70 |
| | ping (20 packets) | 19.03 | 375.5 | 5.58 |
| | netserver | 10.01 | 6,120.9 | 0.74 |
| | netperf | 10.02 | 6,289.6 | 0.94 |
| | iperf (server) | 10.02 | 6,196.5 | 0.87 |
| | iperf (client) | 10.04 | 6,170.6 | 1.11 |
| | pacman | 20.00 | 4,129.5 | 2.95 |
| | pacman | 30.00 | 4,687.3 | 0.87 |
| | unixbench | 1,677.24 | 4,799.5 | 0.14 |

### Unitary System Call Tracing Overhead

Our goal is to define the interval $[t_{min}^{(OH)}, t_{max}^{(OH)}]$, the interval of possible values for the unitary tracing time overhead $t$. The granularity of the hardware counter does not allow high precision; with the 40 ns threshold, we count less than 20 ticks per unitary tracing time overhead. Nevertheless, because of the high variability in collected values (around 20%), we can still define a range of the time overhead for a set of common system calls. We measure tracing overhead for the system calls described in Table V.3. They indeed appear as good candidates for intrusion detection as they have a direct impact on system execution; i.e. scheduling of applications, memory access, inter-thread communication.

- Memory mapping (MMAP): maps a number of pages from one application's address space to another.
- Inter-process communication (IPC): sends a message to a thread and receives an IPC message from another.
- Thread Yield: forces the running thread to yield the CPU.
- Thread Scheduling: exchanges thread priority and execution time window.
- Sleep: suspends the calling thread for an amount of time.

For every system call, we run 1000 measurements. For a single system call, tracing time can take a wide range of values between 240 ns and 760 ns: minimal, average, and maximal values are respectively: $t_{min}^{(OH)} = 240\ ns$, $t_{av}^{(OH)} = 376\ ns$, and $t_{max}^{(OH)} = 760\ ns$.

**Table V.3:** *Tracing overhead measurements for a selection of system calls*

| System call | $t_{min}^{(OH)}$ (ns) | $t_{av}^{(OH)}$ (ns) | $t_{max}^{(OH)}$ (ns) | var (%) |
|---|---|---|---|---|
| MMAP | 240 | 391 | 760 | 20 |
| IPC | 240 | 399 | 760 | 19 |
| Thread Yield | 240 | 343 | 720 | 24 |
| Thread Sched. | 240 | 362 | 640 | 19 |
| Sleep | 240 | 382 | 760 | 23 |

## V.1.5 Discussions

### Interpretation of Results

Following the approach described in Section V.1.2, we estimate the tracing time overhead in Table V.4, which shows the minimal, average, and maximal computed values of $r_A^{(monitoring)}$.

$r_A^{(monitoring)}$ values can vary significantly in function of the application from $10^{-4}$% to 17%. Thus, we can not easily define a factor to predict time overhead in function of

**Table V.4:** *Time overhead for tested applications*

| **Application** ($\tau_A$) | | $r_A^{(monitoring)}$ (%) **min** | **average** | **max** |
|---|---|---|---|---|
| Native | ping server (busy) | 0.01 | 0.02 | 0.04 |
| | sh.mem. (client) | 0.01 | 0.02 | 0.03 |
| POSIX | inetd (ping requests) | 0.28 | 0.44 | 0.88 |
| Linux | empty | $< 0.01$ | $< 0.01$ | $< 0.01$ |
| | dd (128 MiB) | 0.47 | 0.73 | 1.47 |
| | ps -ef | 5.42 | 8.49 | 17.20 |
| | find / | 0.36 | 0.57 | 1.15 |
| | gzip (128 MiB) | $< 0.01$ | 0.01 | 0.29 |
| | ping (20 packets) | $< 0.01$ | 0.01 | 0.29 |
| | netperf | 0.15 | 0.24 | 0.48 |
| | iperf (server) | 0.15 | 0.23 | 0.47 |
| | pacman | 0.11 | 0.18 | 0.36 |
| | unixbench | 0.12 | 0.18 | 0.37 |

simple parameters such as time of execution and amount of static code instructions. The time overhead depends on the type of system calls and context of execution (especially for Linux virtual machines).

Because the amount of system calls for native applications is very low, the time overhead is negligible. More complex applications, e.g. Linux and POSIX programs, present much higher time overhead. `ps` Linux process spends particularly the longest time for tracing system calls ($5\% \leq C_A^{(monitoring)} \leq 17\%$). The majority of the selected applications correspond to a ratio $r_A^{(monitoring)}$ in the order of $10^{-3}$, which represents a tracing time overhead of 60 ms for one minute of program execution time. The overhead affects application time performance, but remains reasonable in the majority of cases.

### Comparison of Tracing Impact with Previous Work

Some previous work [94] implements a HIDS for monitoring a Linux virtual machine running on a type II hypervisor called User-Mode Linux (UML) Monitor. The authors compute the total time overhead of system call monitoring for three well used Linux processes: ps, find, and ls. The overhead corresponds to the whole implementation, system call tracing and detection. Their results point already a wide diversity of time overhead, from 3% with `ps -ef` to 77% for `find / > /dev/null 2&>1`. **Our implementation introduce a comparatively low overhead: average values are 8% in the first case and less than 1% for the second operation.**

Another approach [105] analyzes the occurrences of system calls in applications. From the high throughput of system calls processed by their solution, the authors deduce a visible but negligible impact at runtime: they argue that their HIDS can process

**Table V.5:** *Influence of the context switch period on tracing overhead*

| Context Switch Rate | Call/period ($s^{-1}$) | $C_A^{(OH)}$ (for $C_A^{(monitoring)} = 1s$) |
|---|---|---|
| 100 $\mu$s | 2.26E-04 | 172 ms |
| 100 ns | 2.26E-00 | 8 $\mu$s |

from 12000 to 22000 system calls per seconds when common operating system's services usually run around 2000 system calls per second (six to ten times lower). This argument seems questionable for modern virtual machines, in light of our results; in our test, many of the applications raise more than 4000 system calls per second (Pacman, dd, netserver, etc.).

### Security Impact

As highlighted in Section V.1.1, in the case where the kernel implementation can not preempt system call handling, time isolation is compromised in the system. In the worst-case situation, the system call handling starts just before context switch: i.e. the context switch can be delayed by a maximum duration corresponding to the execution of the longest system call handling. Thus, according to our measurements, an application can extend its time window to maximum $T_{OH}^{(max)} = t_{max}^{(OH)} = 760$ ns between two context switches.

The risk of system call tracing impact on system security increases by reducing the context switch period. The shorter the time period for an application running system calls is, the more probable a context switch to be delayed by system call tracing is. Table V.5 shows an example of worst case execution, where every context switch is delayed by a system call. The system runs two applications, one with real-time constraints and a non-critical program raising continuously system calls. This second software corresponds to a Linux thread repeating the command `ps -ef` in an infinite loop. In function of the context switch rate, the table shows average amount of system calls raised by the monitored application between two context switches and the final tracing delay for 1 s of non-critical application cumulated execution time. This vulnerability could be exploited by an attacker running exclusively system calls from a non-critical program to compromise the availability of a critical program.

PikeOS kernel is non-preemptive. Thus, context switch can not directly preempt system call handling in the kernel. Nevertheless, to control time spent in the kernel, it includes preemption points, notably for long operations and before returning to user-space. Some long system call handlers, such as creating or destroying a task (above 10 $\mu$s in average), can include preemption points; though most of them do not, because of their speed (generally less than 1 $\mu$s). Hence, the tracing impact described in Section V.1.5 also applies to PikeOS real-time hypervisor.

#### Considerations for the Detection Engine Implementation

Intrusions in MCS most likely target non-critical programs: unlike closed critical applications, they usually contain vulnerabilities. The attacker aims either to steal data from the system or to disturb its execution. In both cases, she eventually has to interact with the system (OS kernel, other applications) from the non-critical software.

Thus, our strategy consists in monitoring all applications using a single detector. The detector would analyze system call generic information (parameters, type, etc.), correlated to the thread 's criticality level and its execution context: e.g. the running CPU core, and interactions with other applications.

### V.1.6 Conclusion

We evaluated the impact of system call instrumentation for deploying intrusion detection in embedded MCS, through a concrete implementation based on PikeOS hypervisor. The implementation shows a reasonable time overhead for the majority of applications. However, because of new delays induced for tracing in the OS kernel, time isolation between applications is not guaranteed anymore.

## V.2 Offline Safety-Aware Hybrid Anomaly Detection Service for Embedded Mixed-Criticality Systems

In this section, we introduce an offline anomaly-based HIDS – or anomaly detection system (ADS) – which is based on the safety-aware system call tracing infrastructure developed in Section V.1. The goal is for the ADS to cover both hardware level and OS level footprints of the monitoree's execution: **to our knowledge, this is the first framework to contextualize the system call representation with HPC values.** We analyze the performance impact for adding HPC data to system call traces and evaluate the anomaly detection accuracy on 3 use-cases in different trace configurations (with and without HPC values).

Section V.2.1 describes the monitoring infrastructure, while Section V.2.2 introduces the trace analyzer solution. Section V.2.3 defines the experimental setup, for us to perform the evaluation of the system in regards to performance impact in Section V.2.4 and in respect of security efficiency in Section V.2.5.

### V.2.1 Hardware and Software Based Monitoring Infrastructure

As developed in Chapter IV, our goal is to represent the abstract states of the execution of the monitoree by actual traces of system events at runtime. Referring to the literature review in Chapter III, system call and HPC are two well-known alternative sources of system events for intrusion detection. Yet to our knowledge, no previous work proposed to couple both sources of information; the advantage of monitoring these two types of events is to extend the coverage of the execution monitored by the HIDS: i.e. on system hardware (cache activity, type of executed instructions, etc.), as well as at RTOS software

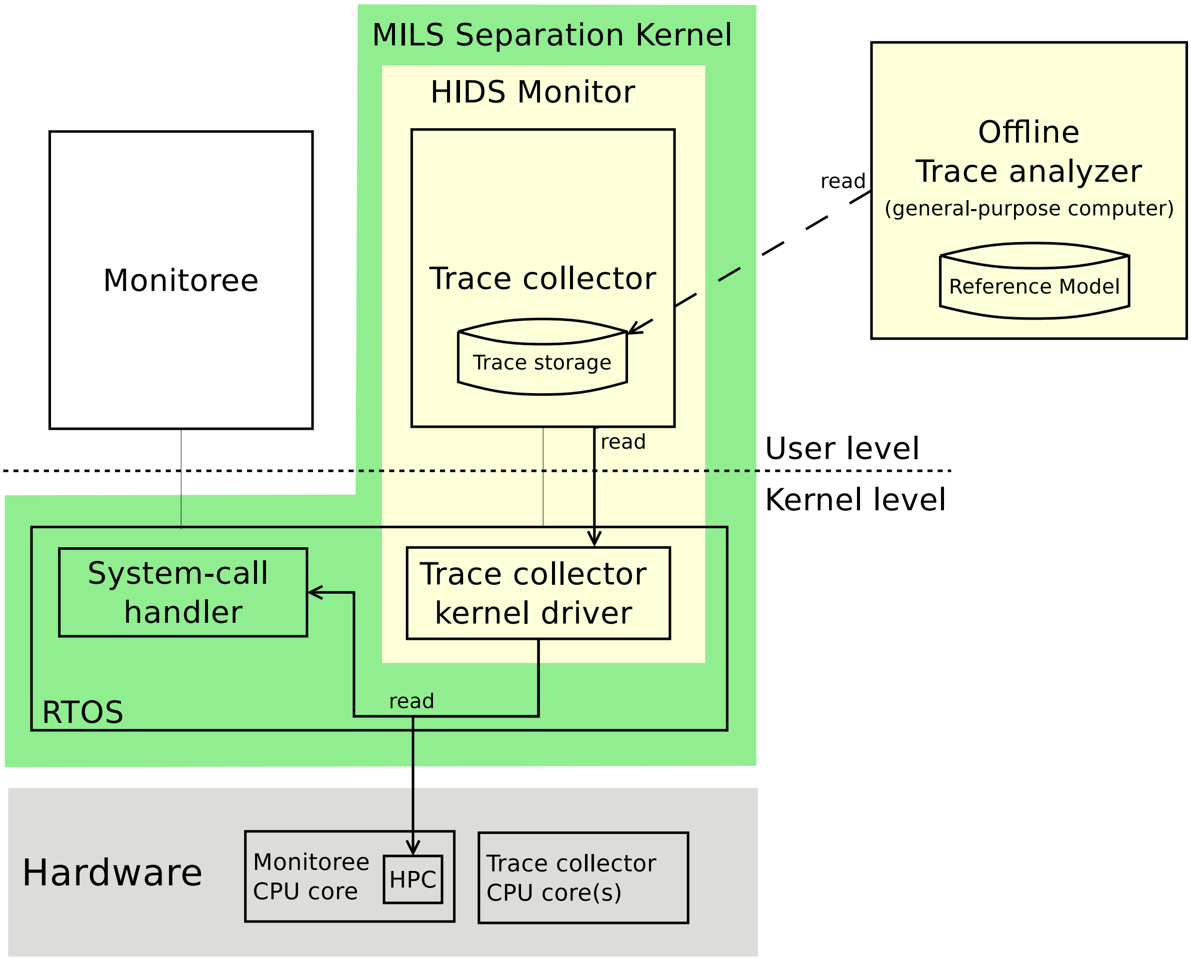


**Figure V.2:** *System call and hardware counters based offline HIDS architecture*

level with privilege operations requests. Hence, we build a hybrid HIDS, monitoring both the OS interface through system calls and the hardware usage with HPC.

We propose to trace a program execution at each system call: a state in the execution graph corresponds to a trace composed of the system call $sc_{type}$ and $(c1, ..., c_n, t)$, respectively the $n$ traced HPC and time values counted since previous system call. We assimilate traces as vectors of shape $(sc_{type}, c_1, c_2..., c_n, t)$.

### System Architecture with the Monitoring Infrastructure

Figure V.2 provides an overview of our system architecture, which follows the main principles developed in Section IV.4. More specifically, we base the HPC and system call monitoring framework on the system call instrumentation based HIDS described with Figure V.1 (Section V.1.3). In addition, we integrate to this initial framework a new feature to read and store HPC values from the system call tracing kernel driver on the monitored core, when a system call is trapped in the kernel. For this study, the detector application runs offline, on a remote machine.

### Hardware Performance Counters Selection

To model normal and anomalous execution, we need to select relevant HPC events, which provide valuable information regarding security. We apply an application profiling based approach to identify anomalies in the execution of the monitored application. For this, our objective is to select the best counter events presenting a low variability for different executions of the same application and high variability between executions of different applications.

We apply the following filtering process, later detailed in Section A, to select relevant HPC candidates:

1 - We implement a normal test application and a set of programs deviating from this application. We collect traces offline for each test case and for every traceable HPC event on the target platform. This operation is repeated to estimate standard deviation and mean values of the events.

2 - We calculate the standard deviation of the mean values over all test cases for each HPC event. These information allow an initial filtering of irrelevant events, which show very close patterns for both normal and anomalous programs, or are too unstable. We also remove events that provide redundant information.

3 - From the remaining selection, depending on hardware constraints for tracing, we manually choose the HPC events to provide the most meaningful information on the hardware state.

## V.2.2 System Events Traces Analysis for Anomaly Detection

### Anomaly Detection Framework Overview

The trace analyzer, or Anomaly Detection System (ADS), compares a reference model of normal execution with the actual execution of the program at runtime. It defines a representative white list of benign execution paths for a traced program, in order to detect anomalous deviations from the baseline. From a monitored program execution trace sequence given as input, it returns a Boolean identifying the sequence either as benign or anomalous. Figure V.3 describes the ADS architecture. It is composed of three main blocks:

- **Trace formatter:** System call IDs are translated to an understandable representation for the machine.

- **Trace predictor:** System call IDS, together with time and HPC values, feed the trace predictor to compute the prediction of the next trace.

- **Trace comparator:** The comparator evaluates predicted and actual trace sequences to finally identify the presence of anomalies in the monitored execution.

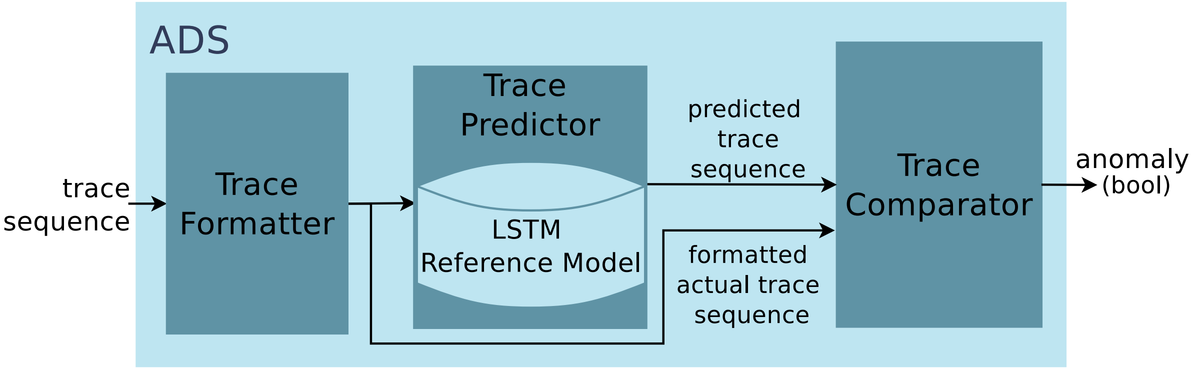


**Figure V.3:** *Anomaly detection system engine*

### Trace Formatter

System call type $sc_{type}$ variable can take a limited set of $V$ values. At runtime when a trace contains a system call unknown from the reference model, the ADS can already raise an anomaly without further analysis. In the case of identified system call, the formatter uses one-hot encoding method to convert the one-dimension variable into a vector of dimension $vec_{sc}$ in $\{(1,0...0),(0,1,0...0),...,(0...0,1)\}$.

We count HPC events between two consecutive system calls. In the next step, the predictor manipulates values in the interval $[0,1]$; the formatter normalizes HPC and time values in function of maximum and minimum values of each variable in the reference model trace. Writing $x^{norm}$ the normalization operation, the transformed trace finally fits the following format: $(vec_{sc}, c_1^{norm}, c_2^{norm}, ..., c_n^{norm}, t^{norm})$.

### Trace Predictor

We assimilate the traces as words in a language, where sentences are sequences of a program execution. The execution time and execution order properties in the trace sequences motivate the use of Natural Language Processing (NLP) techniques for predicting a future trace from a preceding sequence: on Figure V.3, the predictor needs $p$ traces to predict the next trace. Thus, we apply a subclass of recurrent neural networks called Long Short Term Memory (LSTM), to predict next traces. For example, Kim et al. apply this method on system call streams for anomaly detection [85]. Our model is a neural network, which arranges LSTM nodes of depth $p$ in $L$ layers of distinct sizes $\{n_1, n_2, ..., n_L\}$; Section V.2.3 details the selected parameters. The predictor models benign execution; training must be substantial in order to maximize the execution graph coverage: the training set is composed of benign traces to describe normal execution.

### Trace Comparator

To identify a trace sequence as anomalous, the trace comparator computes for each trace element the normalized deviation $e$, between the actual and predicted values. When $e$ exceeds a predefined deviation threshold $e^{(ref)}$, we identify an anomaly. If the total of detected anomalies $n_{anomalies}$ is higher than a predefined threshold $n_{anomalies}^{(ref)}$, the trace comparator considers the trace sequence as anomalous. Reference threshold values are defined empirically in Section V.2.5.

## V.2.3 Experimental Setup

### Platform Setup

We develop the security framework on PikeOS real-time hypervisor [128], which supports a total of 122 system calls. We base our implementation on Cortex-A53 processor [132], which is derived from ARMv8-A processor architecture, because it is well used for embedded system deployments. This processor has a large amount of HPC events to trace; the selection process is detailed in Section A. Table V.6 shows the final set of traced counters.

**Table V.6:** *Hardware performance counters selection*

| Type | HPC | Description |
|---|---|---|
| Data-Flow | L1 Data Cache accesses<br>L2 Data Cache accesses | memory accesses (load/store)<br>and missed accesses |
| Control-Flow | Instructions | amount of instructions |
| | Exceptions | privileged operation requests |
| | Mispredicted branches<br>Indirect branches spec. executed | branch predictor trace |

### Test Set

We test the ADS through three use-cases: they correspond to periodic applications running a limited set of system calls. For each configuration, we define a normal program execution and deviations to identify as anomalies. Every case (normal execution and deviations) is traced for a certain amount of time, to be then analyzed by the ADS engine.

**Hello:** the monitoree is a simple hello world, with one system call type for printing a message on the console (Table V.7). Implemented anomalies modify the program execution flow to mimic code injection or return oriented programming (ROP) based intrusions. Since we built the counters selection on this test set (Section A), our objective is to assess the relevance of our model compared to HPC monitoring alone.

**Linpack:** this use-case, based on Linpack benchmark [51], extends the previous set: it is another example of ROP or code injection execution footprint, which raises only one type of system calls for sleeping between two Linpack task iterations. Table V.8 describes task profiles. We expect the time to be a reliable source of information to detect anomalies. Though, this indicator is easier to manipulate than hardware events such as processor's caches or branch predictor related events. Therefore, we implement

**Table V.7:** *Hello application profiles*

| **Type** | **Test** | **Description** |
|---|---|---|
| Normal | print_loop | in a loop of $N = 100$ iterations, print a message with the iteration index `i` |
| Add | add_qp | open and close a queuing port after the loop |
|  | add_iter | run $2 * N$ loop iterations |
|  | add_tab | write into an integer table of size $N$ at each loop iteration: `tab[i] += i` |
| Remove | rm_iter | run $N/5$ loop iterations |
|  | rm_print | remove `i` parameter in the print call |
| Replace | rp_qp | initialize, write, read, and close a queuing port |
|  | rp_sleep | sleep for 5 second |

anomalies whose execution time is equivalent to the normal reference. This test aims at verifying that HPC remain suitable features to detect these anomalies.

**Table V.8:** *Linpack application profiles*

| **Test** | **Description** |
|---|---|
| Normal | run Linpack for different combinations of inputs:<br>$arsize = 10$ and $nreps = 2 * k$, for $k \in [1, 8]$ |
| Wrong Inputs | run normal test with $arsize = 11$ |
| Fake Linpack | instead of Linpack, write a value into a table of $2,1KB$<br>the time execution approximates the normal configuration |

**Network File Reader:** for this use-case, the monitoree is a UDP server application receiving a file through the network. The monitored program remains unchanged, contrary to both previous use-cases. With this test set, we consider anomalies based on the misuse of its interface; i.e. we apply different client profiles (Table V.9). The monitoree is more complex than in previous use-cases; it raises 7 types of system calls.

### Anomaly Detection System Engine Configuration

We implement the detector in Python language using TensorFlow 1.11.0 framework [2] with the module `nn.rnn_cell.LSTMCell`. After testing different parameters, we empirically select a depth $p = 150$ for each LSTM cell. The neural network is composed of $L = 3$ layers of $n_1 = 300$, $n_2 = 200$, and $n_3 = 30$ LSTM cells. For faster convergence, the training minimizes the cross-entropy between the prediction and the actual trace.

The predictor is trained with a dataset composed of $70,000$ traces of normal execution. 200 training iterations are necessary to approximate the lowest prediction error.

**Table V.9:** *Network file reader client profiles*

| Test | Description |
|---|---|
| Normal | send $1MB$ file in an infinite loop,<br>using sync messages before and after the tranfer. |
| Wrong Sync (WS) | continuously send wrong sync message |
| Fast Send (FS) | faster time synchronization between messages |
| Bigger File (BF) | send $20MB$ file |
| Bad Packets (BP) | send normal file twice, using bigger UDP packet size |

### V.2.4 Performance Evaluation of the Framework

#### Time Overhead

The framework impacts the time execution of the monitoree. The overhead is the time for writing the trace to the collector driver. For a given application $\tau_A$, with the execution time $C_A$ without monitoring, we define $C_A^{(monitoring)}$ the execution time of $\tau_A$ under monitoring (i.e. active trace collection), so that:

$$C_A^{(monitoring)} = C_A + \sum_{i=1}^{N_A^C} t_i \Rightarrow r_A^{(monitoring)} = \frac{1}{C_A} * \sum_{i=1}^{N_A^C} t_i \tag{V.5}$$

where $N_A^C$ and $t_i$ respectively represent, during the execution time of $\tau_A$, the total amount of generated trace collection events and the time for generating the $i^{th}$ trace in the system call handler. We define the upper boundary for the time overhead $r_A^{(monitoring)}$ as $R_A$, so that:

$$\begin{aligned} &\forall i \in \mathbb{N}^*, 1 \leq i \leq N_C^A, t_i \leq T_C \\ &R_A = \frac{N_A^C * T_C}{C_A} \geq r_A^{(monitoring)} \end{aligned} \tag{V.6}$$

with $T_C$ the maximum time for generating a trace in our configuration.

Table V.10 shows that adding HPC monitoring requires approximately 32% of the initial time for tracing a system call. This previous work highlighted also the volatility of the system call rate for different applications, from less than 30 to $225,000$ system calls per second (Section V.1.4). In the experiment, we observe system call rates of $1,090/s$ for network file reader, $646/s$ for `Hello` test, and $104/s$ for `Linpack` test. From these measurements and our system time overhead, we finally compute a total time overhead lower than 0.2% of monitoree's execution time (i.e. $r_A^{(monitoring)} < 0.2\%$).

The collector stores the traces in a specific memory location. The user-level trace collector is lock-less, since reading the traces does not directly affect the monitoree: because the monitoree and user-level trace collector run on different CPU cores, both

**Table V.10:** *Maximum tracing time overhead with 99% confidence interval (1500 samples)*

| **Tracing** | $T_C(\mu s)$ | $R_A$ (%) | | |
|---|---|---|---|---|
| | | Network File reader | Hello | Linpack |
| **SC only** | 0.76 | 0.08 | 0.05 | $< 0.01$ |
| **SC and HPC** | 1.00 | 0.11 | 0.06 | 0.01 |

tasks can run in parallel.

### Runtime Memory Overhead

According to the infrastructure costs for system call monitoring described in Section V.1.1, the framework induces a runtime memory overhead caused by the additional user-level trace collection partition to collect generated traces in a protected memory area before they are analyzed; in our case, we need to store all generated traces, since we perform offline analysis on a remote system.

One trace element size is $48B$: it stores the system call type, hardware performance counter values, and time since last system call. For our experiment, we infer an average rate of $52kB/s$ for network file reader test, $31kB/s$ for `Hello` test, and $5kB/s$ for `Linpack` test. Our system should either be able to process the data-flow with this rate, or it should have sufficient storage capacity.

An application which generates a high rate of system calls can represent an issue for runtime deployment of the framework, notably in term of trace storage. This problem is particularly relevant for embedded MCS with limited hardware resources. As a workaround, we can add selection mechanisms in the framework, to only analyze a set of traced events: for example applying tracing on certain sensitive regions of the monitored program or restricting the set of sensitive system call types to monitor. Alternatively, we could also monitor the system call rate information itself: upon a predefined threshold of system call rate (i.e. count of system call events executed withing a given time period), we detect an anomaly. We do not develop these solutions further in this section: the aspect of runtime trace analysis is addressed in Section V.3, where we present an approach to port the analyzer locally, on the target.

## V.2.5 Security Evaluation

### Security Evaluation Approach

In the experiment, we test and compare different trace configuration combinations with system call, HPC, and time. For each configuration, we compose the test set of one normal execution and several deviations from this baseline. We run 10 iterations for every test case, extracting $1,000$ execution traces each.

We note from early experiment that the detection varies much depending on the trace deviation threshold $e^{(ref)}$ defined for the trace comparator (Section V.2.2). Therefore, we assimilate the final set of anomalous detected dataset as the union of detected datasets

**Table V.11:** *Anomaly detection (%) for network file reader test*

| Trace Type | WS | FS | BF | BP | $R$ (%) |
|---|---|---|---|---|---|
| SC only | 10/10 | 10/10 | 0/10 | 0/10 | 50.0 |
| SC + HPC | 10/10 | 9/10 | 9/10 | 9/10 | 92.5 |
| SC + Time | 10/10 | 8/10 | 2/10 | 4/10 | 60.0 |
| SC + HPC + Time | 10/10 | 9/10 | 9/10 | 8/10 | 90.0 |

for several values of $e^{(ref)}$, which we define empirically as the median, $3^{rd}$ quartile, and maximum values of $e^{(normal)}$ on the normal test dataset. By using the normal test instead of training dataset, our objective is to show what anomalies can be distinguished from normal execution, without false-positives.

### Anomaly Detection Results

Since `Hello` and `Linpack` monitoree references include one type of system call, their model contains HPC and time features only. Applications raising other system call types are considered as anomalous: i.e. `rpsleep`, `addqp`, and `rpqp` tests are automatically detected for `Hello` test. In the context of `Hello` simple use-case, all anomalies are detected, for all configurations: HPC alone, time alone, and both HPC and time together. As expected in `Linpack` test, time alone configuration does not detect any anomaly. However, models including HPC (with and without time) detect all anomalies. Table V.11 shows the detection results for the network file reader test: we provide $R = \frac{TP}{TP+FN}$ the recall, as defined in Equation (II.1). In this case, the combination of system call and HPC provides the best detection score, while adding the time to the model slightly degrades the detection for one test. We observe that system call only configuration shows better detection score for Fast Send test (FS).

Experimental results confirm the validity of our model's features combination: tracing HPC between successive system calls succeeds to precise system call based detection of anomalies for our use-cases. Though, depending on the application profile, the ability of features (system call type, HPC, time) to reveal anomalies fluctuates. Hence, we could refine the features selection in function of monitoree's properties. Alternatively, we could run models with different features sets in parallel, merging their results to optimize the detection.

### Security Monitoring Coverage

Following the evaluation methodology from Section IV.5.1, we distinguish three criteria:

- **monitoring scope:** we check the execution state at system call instructions. This means that out monitoring scope starts from the beginning of the monitored application's execution until the last system call is called. Hence, a major limitation of our solution is that a monitored application executing no system call instruction cannot be covered by our detection method; more generally, when no system call is executed in the end of the monitored application's execution, a possibly large portion of the execution path cannot be protected, offering an attractive window of opportunity for an attack. In such case, we can define $w_{uncov}$ dynamically for a given application $\tau_A$ as the time period between the moment the last system call is called until the application returns. $w_{uncov}$ is the unique time window in the monitored execution when monitoring cannot apply: i.e. $w_{uncov} = C_A^{(uncov)}$.

- **monitoring continuity:** in our current setup with offline analysis, we collect all generated traces: monitoring is continuous. This may change when trace analysis runs locally on the system; i.e. in case of performance limitation (for example due to too many traces to analyze), we may apply discontinuous monitoring to reduce the performance impact of our solution.

- **maximum detection time:** this indicates the maximum window of opportunity for an attack, which is the time for the attack to execute and reach its goal before being detected by the ADS framework. The current solution performs trace checking on a remote system. Hence, we define $w_{detect}$ as the maximum time from the start of the attack until the corresponding trace is ready for the analysis; i.e. the trace has been stored by the trace collection application in the predefined protected storage.

  For our solution, $w_{detect}$ corresponds to the time to reach the next executed system call instruction after the attack has been initiated – i.e. the worst-case time is the maximum time between the execution of two successive system call instructions – added to the time for generating and collecting the trace. we can determine $w_{detect}$ dynamically, since execution states (or checkpoints) are system call instructions in the runtime execution path. Consequently, the more frequently the monitored application executes system calls, the more detection checkpoints, the faster the detection; but also the higher the performance overhead (for generating and storing traces).

### V.2.6 Conclusion

This study highlights the relevance to combine HPC and system call traces for anomaly detection in the execution of a given program. We introduced a novel HIDS, based on the safety-aware kernel level tracing approach developed in Section V.1. The solution comes with a low performance overhead for trace collection, which highly depends on the monitored application; i.e. the amount of system calls at runtime.

However, as we perform the trace analysis offline on a remote system, we cannot determine the performance impact and detection latency metrics for monitoring. Another limitation is the limited scope of the evaluation. The monitored program must execute system calls to be monitored: this constrains possible tested applications and we are not aware of any open test benchmarks or datasets for the RTOS used in this experiment.

Hence, in Section V.3, we propose an alternative ML assisted HIDS solution based on HPC to widen the scope of tested programs. For the first time, we also evaluate the ported online ML based trace analysis on an embedded MCS to detect threats at runtime.

## V.3 Online Safety-Aware Hardware Performance Counters Based Anomaly Detection System

The HIDS introduced in the previous section comes with several pitfalls. In particular, it does not support online detection, which represents a key issue – especially when heavy computations are involved – for integrating the solution into a system with real-time constraints and limited hardware resources. In addition, the previous framework relies on the execution of system calls by the monitored program. Because of the specific deployment environment, we intend to base our implementation on an industrial proprietary RTOS; this makes the solution difficult to evaluate, since we cannot easily use existing open-source test sets and datasets, we instead need to create relevant test applications.

Hence, we introduce an alternative online HIDS framework implementing ML-assisted detection. This approach leverages HPC tracing to limit the intrusiveness with the monitored program. We propose an evaluation of the cost and impact on system' safety, for porting the ML model into the monitored system.

The remainder of this section is organized as follows: Section V.3.1 introduces the online HIDS framework, which is based on the architecture introduced in methodology (Section IV.4). Section V.3.2 details the LSTM based ML model used to identify anomalies in the monitored program's execution. Section V.3.4 describes our evaluation of the solution in terms of security efficiency and performance impact, following the method discussed in Sections IV.5 and IV.5.2. In Section V.3.3, we provide a time analysis to integrate the framework into an embedded MCS. Finally, we summarize our findings in Section V.3.5.

### V.3.1 Online Host Intrusion Detection System Architecture

Figure V.4 provides an overview of the HIDS framework. Like the other approaches discussed in this chapter, the system architecture is based on the generic solution described in Section IV.4. In contrast to system call tracing based HIDS, this approach implements a fully transparent monitoring of the monitoree's execution: all monitoring components run into an isolated user-level partition, which is part of the MILS separation kernel. The monitoree is a user-level program executing alone on a single core. The system HIDS monitor is composed of two main applications running on a distinct set of CPU cores:

- **trace collector:** reads periodically the HPC values of the monitoree CPU core, at a predefined sampling rate. It then writes the data into the trace storage.
- **trace analyzer:** reads periodically the traces from the trace storage, to analyze them on the fly, using the ML based reference model generated offline and configured

before the final deployment.

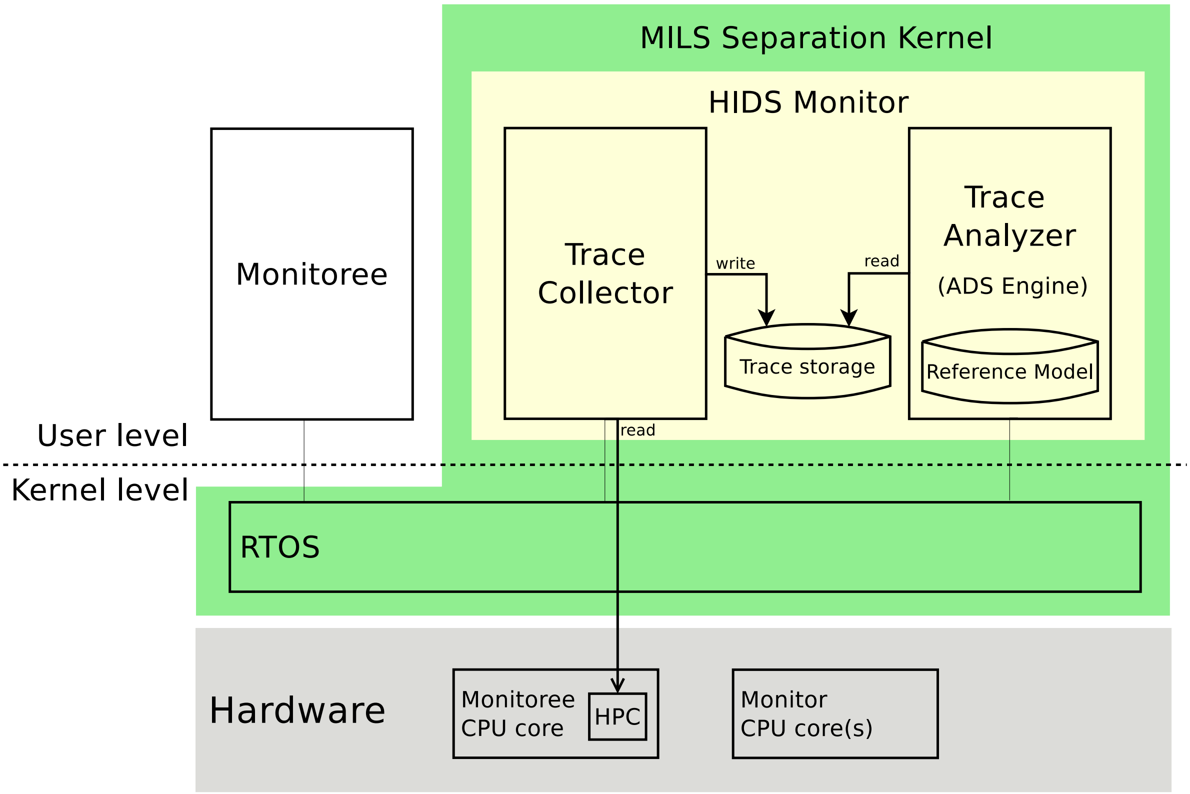


**Figure V.4:** *Online hardware counters based host intrusion detection system architecture*

The **trace collector** corresponds to a periodic tasks, which reads the HPC counter values and stores these data into the trace storage. In line with the previous HPC and system call based ADS framework, we select the set of HPC events to trace as described in Section V.2.1. We determine the HPC tracing period configuration experimentally (Section V.3.4) using the system timing analysis developed in Section V.3.3.

We initialize the HPC counters at boot time from the kernel; i.e. no user-level application is able to modify the HPC events selection nor to overwrite the counters values. Unlike the previous ADS approach which requires execution from kernel space, this monitoring solution runs – after initialization – exclusively from user-space, since HPC traces are readable from user-space. This transparent tracing method leverages a SoC based monitoring unit (Performance Monitoring Unit on ARM architecture), which is able to map the individual HPC counters of each CPU core. Hence, we provide the trace collector application with exclusive access to the SoC monitoring unit, so that it can map HPC counters of the monitored CPU core from the distinct set of monitor cores. Consequently, this approach does not require to interrupt or disturb the monitoree's execution to generate and store traces. We must qualify the transparency of the tracing solution with the security exposure: as HPC values are readable from user-space, an adversary attempting to run malicious activity in the monitoree could leverage these information to adapt and hide the intrusion from the HIDS monitor.

The **trace analyzer** runs from a Linux guest OS, using TensorFlow Lite [119], an open-source ML framework. The reference model is trained offline, so that only the final inference model is configured on the embedded MCS for online detection.

### V.3.2 Machine-learning Based Trace Analyzer

Similarly to the previous solution (Section V.2.2), the ML based trace analyzer leverages a LSTM reference model; it also relies on a sliding window to analyze every successive traces, using a set of preceding traces. It is composed of three main modules, as described on Figure V.3:

- **trace formatter:** it formats the traces so that they can be used by the trace predictor. With the same approach developed in Section V.2.2, we decompose this stage into two successive operations applied to each traced HPC:
  - computing the difference value between the preceding and current HPC reads.
  - normalizing this difference value, which is converted from an integer into a float in the interval $[0, 1]$.
- **trace predictor:** from a sequence of consecutive traces, it returns a prediction of the next trace. Following the previously described offline ADS framework, we use a LSTM model to predict time series data.
- **trace comparator:** taking the prediction from the trace predictor and the corresponding following observed trace, it computes the prediction error; it indicates an anomaly when this error exceeds a predefined error threshold.

In the ADS engine, we define the reference model as the combination of the LSTM model of the trace predictor and other configuration data (i.e. normalization data from the trace formatter as well as the error threshold of the trace comparator); these are computed in a learning phase (i.e. LSTM model training and validation) prior to deployment, in order to minimize the false-positive and false-negative rates on the given dataset. We perform this learning phase offline, because of the related heavy costs (time and hardware resources). Our goal is then to deploy the generated reference model into the embedded MCS platform, to support real-time threat detection during the execution of the monitoree. In the rest of this subsection, we describe the configuration and the steps to build the ADS engine.

#### Trace Formatter

We count HPC events within fixed time windows. The predictor manipulates values in the interval $[0, 1]$; thus, the formatter normalizes HPC and time values in function of maximum and minimum values of each variable in the reference model trace.

**Algorithm 1:** Pseudocode to return the lower bound for the time-steps of a RNN-LSTM.

```
Input  : Full Monitoree's execution trace sequence $s = \{c_1, c_2, ..., c_S\} \in (\mathbb{N}^*)^{N*S}$
Output : Minimum LSTM node depth $d_{min} \in \mathbb{N}$, so that $1 < d_{min} < S$. 0 is an
         erroneous value.

$d \leftarrow 0$;
while $d < S$ do
    $d \leftarrow d + 1$;
    for $i \in [1, S - d]$ do
        past_instance ← SliceSequence($s$, $d$, $i$);
        value_to_predict ← $c_{i+d}$;
        found_ambiguity ← $false$;
        for $j \in [i + 1, S - d]$ do
            other_past_instance ← SliceSequence($s$, $d$, $j$);
            other_value_to_predict ← $c_{j+d}$;
            if past_instance ≈ other_past_instance then
                if value_to_predict ≠ other_value_to_predict then
                    found_ambiguity ← $true$;
                    break;
        if found_ambiguity then
            break;
    if $not$ found_ambiguity then
        break;
$d_{min} \leftarrow d$;
return $d_{min}$;
```

### Trace Predictor

The trace predictor corresponds to a LSTM model. On one hand, many standard parameters are involved to build ML models, such as learning rate, learning bias, and batch size; for these, we use default values (respectively 0.001, 0, and 32). On the other hand, LSTM models come with specific configuration parameters:

- LSTM model topology: the count of layers ($n_{layers} \in \mathbb{N}^*$) and the count of nodes per layer ($n_{nodes} \in \mathbb{N}^*$).
- LSTM node depth: the count of necessary consecutive traces to return a trace prediction ($d \in \mathbb{N}^*$).

The configuration of the LSTM model depends on the use-case: hardware platform, type of monitored application, amount of data to build the model.

When building the LSTM model topology, our goal is to avoid the two following configurations:

- **overfitting:** the model is too sensitive (i.e. it contains too many layers or nodes per layer). It fits the training set of traces so precisely, that the model cannot accurately predict the tested traces

- **underfitting:** the model is too simple (i.e. it contains too few layers or nodes per layer) to accurately predict system trace events.

For a given topology (i.e. $n_{layers}, n_{nodes} \in (\mathbb{N}^*)^2$), we can estimate the formal overhead to perform a prediction in function of the computing complexity of the ML model; typically, the more complex the topology, the longer the prediction, i.e. the longer the detection delay. In practice, the overhead depends on the implementation: i.e. type of hardware platform and AI framework, bare-metal or virtualized implementation. Thus, we experimentally measure the time overhead in Section V.3.4.

For simplification reasons, we define the same depth $d \in \mathbb{N}^*$ for all the nodes of the LSTM model. The trace predictor receives a sequence of $d$ consecutive traces to compute a trace prediction. We build the model using training samples corresponding to sequences of $d+1$ traces: $d$ input traces to make the prediction in addition to the trace to predict. For low $d$ values, ambiguous samples are more likely to be found in the training set: i.e. two samples with the same input sequence but different output traces to predict. We suppose that a high amount of ambiguities in the training set is likely to deteriorate the prediction accuracy; logically, increasing $d$ would improve the ability of the trace predictor to capture pattern sequences in the training set.

Concurrent to the topology complexity increase, the higher $d$, the more computations the prediction involves, i.e. the higher the prediction overhead and the detection latency. The trace predictor induces an initial delay from the time when the monitored application starts, when the HIDS cannot perform trace prediction as it waits for the first set of $d$ consecutive traces to be collected. We define this initial delay $T^d_{uncov} \in \mathbb{N}$, so that $T^d_{uncov} = d * t_s$, with $t_s$ the sampling rate for HPC trace collection. During the delay $T^d_{uncov}$, the HIDS monitor is unable to detect threats, letting the opportunity for an adversary to hijack the application before monitoring starts. Hence, the higher $d$, the longer $T^d_{uncov}$.

Hence, to reduce the detection latency and the initial uncovered execution $T^d_{uncov}$, while avoiding ambiguities in the training set, we provide an approach to compute the minimum depth $d_{min} \in \mathbb{N}^*$ for a given training set with Algorithm 1. We compute the minimum depth $d_{min}$ as follows. Let define the given execution sequence of collected counter values $s = \{c_1, c_2, ...., c_S\}$, so that $S \in \mathbb{N}, S \geq 2$ and $\forall i \in \mathbb{N}, 1 \leq i \leq S, c_i \in \mathbb{N}^N$, with $N \in \mathbb{N}^*$ the amount of HPC counters for each trace. We start setting the initial minimum depth $d = 1$ (Lines 1-3); n.b. such LSTM model corresponds to a standard forward neural network as the LSTM nodes have no internal state. We then increment $d$ (Line 3) until there is no ambiguity in the dataset or until we have reached the maximum size $d = N - 1$. To evaluate the ambiguities in the dataset, we split the sequence $s$ of $S$ traces into $S - d$ slices $\{s_1, s_2, ..., s_{S-d}\}$ of $d$ consecutive traces to train the trace predictor; the function `SliceSequence(s, d, i)` (Line 10) computes the sequence $s_i$ for a given depth $d$ and sequence $s$, as follows:

$$\forall d \in \mathbb{N}, 1 \leq d < S, \forall i \in \mathbb{N}, 1 \leq i \leq S - d, s_i = \{c_i, c_{i+1}, ..., c_{i+d-1}\} \tag{V.7}$$

### Trace Comparator

For each monitored time window, the trace comparator compares the trace prediction made by the LSTM model with the actual formatted trace; upon an empirically defined threshold, the ADS engine determines whether the actual trace corresponds to normal or anomalous execution. Considering, $S$ the full sequence of traces for the entire execution of the monitoree, the threshold $E_{max}$ corresponds to the vector: $E_{max} = (E_1, E_2, ...E_{|S|}) \in (\mathbb{R}^+)^{|S|}$.

Given a time window $w \in \mathbb{N}^*$ so that $w \leq |S|$, we represent the corresponding prediction and the actual trace respectively by two vectors $v_p^{(w)} \in (\mathbb{R}^+)^N, v_a^{(w)} \in (\mathbb{R}^+)^N$, where $N$ is the amount of traced HPC events. Let $i \in \mathbb{N}, 1 \leq i \leq N$ and $v^{(w)} \in (\mathbb{R}^+)^N$, where $v^{(w)}(i)$ designates the $i^{th}$ element of the vector $v$ in the time window $w$. We define $e^{(w)}$ the error between the actual and predicted traces, so that $e^{(w)} = RMSE(v_p^{(w)}, v_a^{(w)})$, with $RMSE$ the root mean-squared error function defined in (V.8).

$$\forall w \in [1, |S|].RMSE(v_p^{(w)}, v_a^{(w)}) = \sqrt{\frac{1}{N} * \sum_{i=1}^{N} (v_p^{(w)}(i) - v_a^{(w)}(i))^2} \tag{V.8}$$

We set $E_{max}$ in the offline building phase, computing for each window $w \in [1, |S|]$ the error's average and standard deviation values on a given training dataset. Considering a dataset of $B$ monitoree execution sequences for building the ADS engine, we set $(e_i^{(w)})_{w=1..|S|, i=1..B} \in (\mathbb{R}^+)^{|S|*B}$ the set of error vectors for the given dataset.

Let $x = \{x_1, x_2, x_3, ..., x_N\}$, where $\forall i \in \mathbb{N}, 1 \leq i \leq N, x_i \in \mathbb{R}^+$. We define $\mu_{i=1..N}(x_i)$ and $\sigma_{i=1..N}(x_i)$, respectively as the arithmetic mean and the standard deviation of the vector $x$. Hence, we define the threshold $E_{max}$ in (V.9) with the coefficient $A \in \mathbb{N}^*$, to empirically set in the building phase for reducing the false-positive and false-negative rates in the detection.

$$E_{max} = (E_w)_{i=1..|S|} \in (\mathbb{R}^+)^{|S|}, \forall w \in [1, |S|].E_w = \mu_{i=1..B}(e_i^{(w)}) + A * \sigma_{i=1..B}(e_i^{(w)}) \tag{V.9}$$

### Anomaly Detection Engine Building Steps

We build the ADS engine offline on a general-purpose computers, prior to deployment of the monitor into the target hardware. We decompose the building stages into three main phases and consider an additional phase for testing the detection accuracy of the solution:

- **initialization:** we define the parameters of the ADS engine:
  - **trace collector:** sampling rate of HPC trace collection
  - **trace formatter:** normalization data.
  - **trace predictor:** configuration of LSTM model hyperparameters (i.e. LSTM model topology, LSTM node depth).
- **training:** we generate the LSTM model with a training dataset containing normal data only (i.e. no anomalies). The goal is to minimize prediction error to maximize accuracy of the prediction on the given dataset. We then infer the value of $E_{max}$ on the training set.
- **validation:** we evaluate the trace predictor error on the validation dataset, which contains normal and anomalous traces of execution. We infer the values of $E_{max}$ with the coefficient $A$, so that the ADS engine is able to distinguish normal and anomalous traces accurately. If the detection results are ineffective, we return to the initialization step, changing the ADS engine configuration to improve the detection.
- **test:** we test the ADS engine ability to accurately detect anomalies on a test dataset.

The training, validation, and test datasets are distinct. We apply the ratio: 70% for training, 20% for validation, and 10% for test.

### V.3.3 Real-Time Analysis of the Monitoring Framework

In this section, we formalize the system as a real-time system to describe how to integrate the monitoring framework into a mixed-criticality system. We evaluate the impact of the framework on system real-time properties and define a set of requirements to deploy the solution.

#### System and Task Model

We define a task as a stream of jobs: $\tau_i = \{J_{i,1}, J_{i,2}, ..., J_{i,j}, ...\}$. We represent the real-time task $\tau_i$ with implicit deadlines as $\tau_i = (C_i, T_i, P_i)$, where $P_i$, $C_i$ respectively represent the priority and the worst-case execution time (WCET) for a job $J_{i,j}$, and $T_i$ the minimum inter-arrival time between successive jobs $J_{i,j}, J_{i,j+1}$ of the task $\tau_i$. We note $a_{i,j}, s_{i,j}$, and $d_{i,j}$, respectively the arrival, start, and deadline times of the job $J_{i,j}$.

More specifically, using the previous task definition, we define the periodic task $\tau_i$ as $\tau_i = (C_i, T_i, O_i, P_i)$. We introduce the additional parameter $O_i$ as the offset of the initial job $J_{i,1}$ relative to the start time of the system: i.e. $\forall j \in \mathbb{N}, a_{i,j} = O_i + j * T_i$.

For this analysis, we simplify the framework described in Section V.3.1, considering a system with two CPU cores executing the following set of tasks:

- **Application CPU core (A):** $\tau_A = (C_A, T_A, O_A, P_A)$ the monitoree task. For this study, we consider the monitoree as a critical periodic real-time task.

  No other tasks run on this CPU core, since these would impact the HPC values, which are continuously traced on the core. Such interfering activity would possibly degrade the application profiling with the trace prediction. To apply HPC tracing on a single task running with other tasks on a single CPU core, we would need to modify our solution design to support task-related HPC context save and restore operation in the RTOS scheduler. Such solution comes with more system intrusiveness, because of the kernel modification requirement.

- **Monitor CPU core (M):** the HIDS monitor, corresponding to $\tau_C = (C_C, T_C, O_C, P_C)$ the trace collector and $\tau_M = (C_M, T_M, O_M, P_M)$ the trace analyzer periodic tasks, in addition to a set $\Gamma$ of $N$ unrelated aperiodic tasks: $\Gamma = \{\tau_i\}_{i=1..N}$. Both $\tau_C$ and $\tau_M$ are periodic tasks, because the security monitoring is based on a periodic check of HPC traces.

All system tasks execute using fixed priority based preemptive scheduling. In the rest of the analysis, we consider the Rate Monotonic [102] scheduling policy as an example.

As mentioned in Section V.3.1, our HIDS monitor framework induces no timing effect on the monitoree's execution: the response time of the monitored application and the system schedulability remain unchanged on the CPU core A. Additionally, even though monitoring activity could theoretically interfere with the CPU core A (e.g. via cache side-channels), we do not consider hardware interference for simplification reasons: we expect the RTOS and MILS separation kernel architecture to provide a proper isolation between system tasks.

Consequently, we have two tasks system: $\Gamma_A = \{\tau_A\}$ and $\Gamma_M = \{\tau_C, \tau_M\} \bigcup \Gamma$. Hence, with RM scheduling policy, the system is schedulable when the following sufficient condition is respected [102]:

$$\frac{C_A}{T_A} \leq 1 \text{ and } \frac{C_C}{T_C} + \frac{C_M}{T_M} + \sum_{i=1}^{N} \frac{C_i}{T_i} \leq (N+2) * (2^{\frac{1}{N+2}} - 1) \tag{V.10}$$

Table V.12 provides an overview of the tasks parameters to determine before system deployment. The monitoree configuration depends on the use-case: as system integrator, we possibly receive $C_A, T_A, O_A$ from an external application provider, as described in our work environment (Section II.2.1). Additionally, since $\tau_A$ runs alone on core A, we set its priority $P_A$ to an arbitrary value. The monitoring tasks $\tau_C$ and $\tau_M$ are constrained by the HIDS framework design: $C_C$ and $C_M$ values depend on the software implementation (user-level HIDS partition), the RTOS, and system hardware. We experimentally evaluate these metrics in Section V.3.4, which condition the practicability of the detection solution. In the following subsections, we develop our analytical approach to configure the rest of the variables.

**Table V.12:** *System task configurations*

<table>
<tr><th>Task</th><th>WCET</th><th>Period</th><th>Offset</th><th>Priority</th></tr>
<tr><td>$\tau_A$</td><td colspan="3">$C_A, T_A, O_A$: given by the use-case (Section V.3.4)</td><td rowspan="3">$P_A, P_C, P_M, O_C, O_M, T_M$: determined via time analysis (this section)</td></tr>
<tr><td>$\tau_C$</td><td rowspan="2">$C_C, C_M$: constrained by the implementation (Section V.3.4)</td><td>$T_C$: constrained by the trace analyzer (this section)</td><td rowspan="2"></td></tr>
<tr><td>$\tau_M$</td><td></td></tr>
</table>

### Trace Collection Period Configuration

On one hand, the shorter $T_C$ (the period for reading HPC counters), the shorter the detection latency. On the other hand, the detection accuracy constrains the minimum value of $T_C$. The trace analyzer evaluates the HPC incremented values in execution time windows of period $T_c$. Because of hardware complexity, we expect that the shorter $T_c$ is, the more hazardous at a given time the execution and corresponding HPC observations are, and so the more difficult the trace sequence becomes to analyze. Additionally, we suggest to limit the lower bound of $T_C$ to avoid stationary window sequences, i.e. when in an observation window, no HPC counter is incremented.

For the monitoring solution to cover the full execution of the monitoree, the trace collector must start simultaneously with the monitoree jobs (V.11). The goal is to maintain consistency in the reference model, avoiding to add complexity by shifting the monitoring windows among successive monitoree jobs.

$$O_C = O_A \text{ and } \exists k \in \mathbb{N}^*, T_A = k * T_C \tag{V.11}$$

### Trace Analyzer Period Configuration

The deployability of the online monitoring solution is a first criteria to constrain the trace analyzer period $T_M$. On one hand, $T_M$ must be set in line with the period $T_C$ of the trace collector task: the trace analyzer must analyze collected traces at least as fast as they are generated in the trace storage (V.12). This corresponds to the basic producer and consumer problem, where the trace collector writes data into the trace storage and the trace analyzer consumes them.

$$T_C \geq T_M \tag{V.12}$$

On the other hand, the trace collection must not be delayed while reading HPC counters at fixed execution times, so that HPC traces remain consistent among execution samples, i.e. $\forall j \in \mathbb{N}^*, s_{C,j} = a_{C,j}$. Hence, $\tau_C$ has the highest priority on the CPU core M; furthermore, we give the priority to $\tau_M$ over the set of other tasks $\Gamma$ (V.13). We define $P^{(max)}$ the maximum task priority in the system. Thus, we set $P_C = P^{(max)}$ and $P_M = P^{(max)} - 1$.

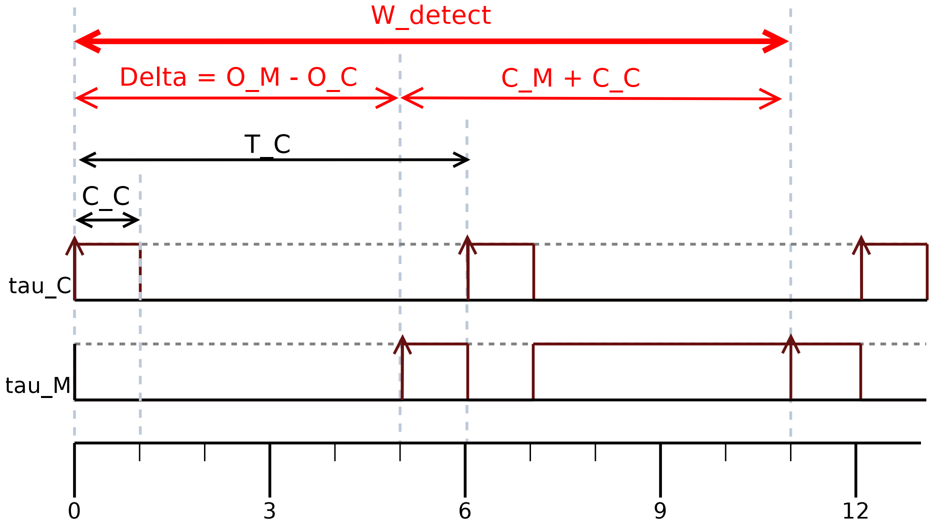


**Figure V.5:** *Example of monitoring execution with with* $C_C = 1, T_C = 6$, $C_M = 5, T_M = 6$, *and* $\Delta O_{C,M} = 5$

$$P_C > P_M > P_i, \forall i \in \mathbb{N}, 1 \leq i \leq N \tag{V.13}$$

With RM scheduling policy, $P_C > P_M \Rightarrow T_C \leq T_M$; thus, with (V.13) and (V.12, we deduce that $T_M = T_C$. Both trace collector and trace analysis tasks execute within the same period, if we set the same value to each task's offset, i.e. $O_C = O_M$. This means that the multi-mode HIDS determines the presence of an intrusion in high security mode (using $\tau_M$), before the next trace can be collected by the task $\tau_C$.

Hence, we can deduce the minimum values $T_M^{(min)}$, $T_C^{(min)}$: $T_M^{(min)} = T_C^{(min)} = C_C + C_M$. We can then describe the condition on $T_C$ for the deployability of the solution with (V.14). When $T_C = T_M = T_M^{(min)}$, the CPU core M is fully utilized ($U = \frac{C_M}{T_M} + \frac{C_C}{T_C} = \frac{C_M + C_C}{T_M^{(min)}} = 1$); i.e. we cannot add other tasks on the CPU core M, while guaranteeing system schedulability. This implies $\Gamma = \emptyset$.

$$T_C \geq T_C^{(min)} \geq C_C + C_M \tag{V.14}$$

### Detection Latency Analysis

We introduce $w_{detect}$ the worst-case detection latency for the monitoring framework composed of tasks $\tau_C$ and $\tau_M$. We note $\Delta O_{C,M}$ the offset difference between $\tau_C$ and $\tau_M$: $\Delta O_{C,M} = |O_C - O_M|$. For simplification reasons, we set $0 \leq \Delta O_{C,M} < T_M$, so that $\forall k \in \mathbb{N}^*, a_C^k \leq a_M^k < a_C^{k+1} = a_C^k + T_C$; i.e. the trace analyzer directly handles the HPC trace after it has been stored in the trace storage by the trace collector.

Let $k \in \mathbb{N}^*$, we note $w_{detect}^k$ the detection latency for the $k^{th}$ HPC trace received by the HIDS monitor. Figure V.5 illustrates the detection latency measurement with an example. We formalize the definition of $w^k_{detect}$ (V.15), and deduce the equivalence (V.16).

$$\forall k \in \mathbb{N}, w^k_{detect} = \begin{cases} max(C_C, r^k_M - r^k_C) + C_M + C_C, & \text{if } r^k_M + C_M > r^{(k+1)}_C = r^k_C + T_C \\ max(C_C, r^k_M - r^k_C) + C_M, & \text{else} \end{cases} \tag{V.15}$$

$$\iff w_{detect} = \begin{cases} max(C_C, \Delta O_{C,M}) + C_M + C_C, & \text{if } \Delta 0_{C,M} > T_C - C_M \\ max(C_C, \Delta O_{C,M}) + C_M, & \text{else} \end{cases} \tag{V.16}$$

We note that increasing $\Delta O_{C,M}$ value induces the detection latency $w_{detect}$ to increase. Thus, to reduce $w_{detect}$, we define the following arrival times condition:

$$0 \leq \Delta O_{C,M} < C_C \tag{V.17}$$

Consequently, we set $\Delta O_{C,M} = 0$; i.e. with (V.11), $O_M = O_C = O_A$, and $w_{detect} = C_C + C_M$.

Finally, Table V.13 summarizes the configuration of our monitoring tasks. In the next section, we evaluate the variables $C_C, C_M, T_C$ in respect to a practical deployment of the solution.

**Table V.13:** *Final monitoring task configurations*

| Task | WCET | Period | Offset | Priority |
|---|---|---|---|---|
| $\tau_C$ | $C_C$ | $T_C$ | $O_A$ | $P^{(max)}$ |
| $\tau_M$ | $C_M$ | $T_C$ | $O_A$ | $P^{(max)} - 1$ |

### V.3.4 Experiment

The goal of our experiment is to evaluate the HIDS monitoring for online detection of anomalies in an embedded MCS. We focus our analysis on the trace analyzer, to determine its impact on the system security and performance. More specifically, we evaluate how the ML based trace predictor configuration influences the detection latency and accuracy.

Let $x$ be a measurable variable and $N \in \mathbb{N}^*$, we note $\mu_N(x)$ the average value of $x$ for a set of $N$ measurements and $X$ the WCET computed value of $x$ variable; we compute the WCET value $X$ as the worst-case measured value of $x$ at runtime. Following the real-time analysis of the framework (Section V.3.3), we study the trace predictor configuration variables:

- $c_M, C_M$: respectively a time measurement and the WCET values to make a prediction
- $C_C$: the WCET for a trace collection iteration
- $T_C$: the period to collect the HPC traces.

On one hand, $c_M, C_M, C_C, T_C$ depend on the execution environment (hardware platform, RTOS and software implementation). On the other hand, $C_A, T_A, O_A$ depend on the tested use-case and system tasks configuration. Hence, for simplification we do not consider $T_A$ and $O_A$ in the analysis (e.g. setting $O_A = 0$ and $T_A$ to a suitable value to synchronize the HIDS monitor with the monitoree execution)

### Platform Setup

Our goal is to implement the HIDS monitor framework described in Section V.3.2 on a typical industrial embedded MCS. Thus, we use the same platform as in previous work (Section V.2.3): i.e. with SYSGO's PikeOS RTOS and Xilinx Zynq Ultrascale+ SoC. The hardware platform includes an ARM Cortex-A53 processor (4 CPU cores), on which we implement the system. This processor provides 6 configurable counters per core, which we set using events of Table V.6. In addition, the SoC supports a performance monitoring unit (PMU), that we use to develop our transparent trace collector task.

### Tested Monitoree Programs

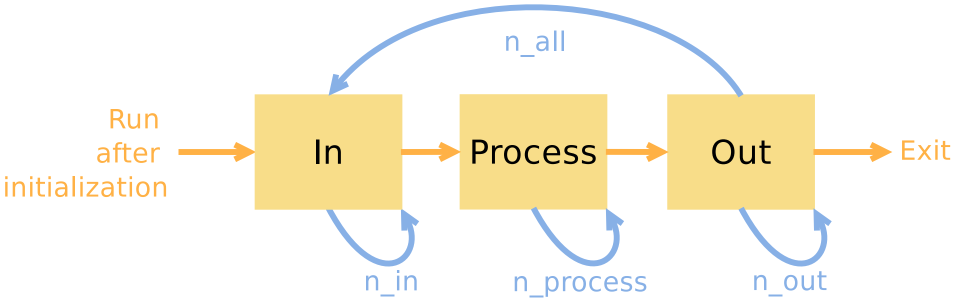


**Figure V.6:** *Generic program representation*

As previously mentioned, we consider $\tau_A$ as a periodic task defined as $\tau_A = (C_A, T_A, P_A, O_A)$; for simplification, we perform the evaluation for a job execution of $\tau_A$, so we do not consider $T_A, P_A, O_A$ in the analysis.

We intend to evaluate the HIDS monitor with a monitoree periodic critical task $\tau_A$, which is representative of embedded MCS applications. We represent the generic monitoree program which implements $\tau_A$ jobs by a sequence of three phases (Figure V.6):

1 **In**: acquiring input data.

2 **Process**: processing input data.

3 **Out**: returning output data.

We define $\mathcal{E}_A$ as the set of tested execution configurations for the task $\tau_A$. Let $e \in \mathcal{E}_A$ be a program execution configuration; we simplify the representation of $e$, so that $e = \{p, l\}$, with:

- $p \in \mathbb{N}$ **a program type:** a unique variable designating a combination of In, Process, and Out phases. We implement In and Out phases with a static data structure (e.g. table), to read from and write to. The Process phase corresponds to a complex sequence of computing operations.

  For our evaluation, we consider three program configurations $p_{matmul}, p_{bsearch}, p_{sort} \in \mathbb{N}^3$; these correspond to applications of TACLeBench [56] benchmark, as described on Table V.14. Hence, we define $\mathcal{P}_A = \{p_{matmul}, p_{bsearch}, p_{sort}\} \in \mathbb{N}^3$ as the set of tested programs.

- $l \in (\mathbb{N}^*)^4$ **a combination of loop iterations:** $l = (n_{in}, n_{process}, n_{out}, n_{all}) \in (\mathbb{N}^*)^4$, with $n_{in}$, $n_{process}$, $n_{out}$ the count of loop iterations respectively for In, Process, Out phases, and $n_{all}$ the amount of times the full sequence is executed by $e$.

**Table V.14:** *Tested monitoree configurations*

| $p$ | **Application name** | **Description** |
|---|---|---|
| $p_{sort}$ | SORT-256 | A descending sort over a buffer of length 256 |
| $p_{matmul}$ | MATMUL-32X32 | A multiplication between 2 matrices of size $32 * 32$ |
| $p_{bsearch}$ | BSEARCH-8192X2048 | A sequence of 2048 binary searches over a set of length 8192 |

We divide $\mathcal{E}_A$ the set of tested configurations into two distinct sets of normal (base) and anomalous tested execution configurations, respectively $\mathcal{E}_A^{(base)}$ and $\mathcal{E}_A^{(anom)}$, so that $\mathcal{E}_A = \mathcal{E}_A^{(base)} \cup \mathcal{E}_A^{(anom)}$ and $\mathcal{E}_A^{(base)} \cap \mathcal{E}_A^{(anom)} = \emptyset$. For this evaluation, we use $l$ to simulate normal and anomalous execution configurations of a given program. Hence, we define the configuration sets in (V.18), with:

- $l_A^{(base)} \in (\mathbb{N}^*)^4$: a single configuration to represent normal execution. As described on Table V.15, $l_A^{(base)} = (n_{in}^{(base)}, n_{process}^{(base)}, n_{out}^{(base)}, n_{all}^{(base)}) \in (\mathbb{N}^*)^4$.

  We select this configuration to study the detection on a relatively long execution time of the monitored program (i.e. multiple time windows): for each tested program, the WCET $C_A \approx 1s$. For the experiment, we set 3 execution configuration of balanced In, Process, and Out phases.

- $\mathcal{L}_A^{(anom)} \in (\mathbb{N}^*)^{4*n}$: the set of $N \in \mathbb{N}^*$ tested anomalous loop configurations $\mathcal{L}_A^{(anom)} = \{l_1, l_2, ..., l_N\} \in (\mathbb{N}^*)^{4*N}$.

$$
\begin{aligned}
\mathcal{E}_A^{(base)} &= \{\{p, l_A^{(base)}\}, \quad \forall p \in \mathcal{P}_A\} \\
\mathcal{E}_A^{(anom)} &= \{\{p, l^{(anom)}\}, \quad \forall p \in \mathcal{P}_A, \forall l^{(anom)} \in \mathcal{L}_A^{(anom)}\}
\end{aligned}
\qquad \text{(V.18)}
$$

**Table V.15:** *Base monitoree loop configuration for all tested use-cases*

| | $n_{in}^{(base)}$ | $n_{process}^{(base)}$ | $n_{out}^{(base)}$ | $n_{all}^{(base)}$ |
|---|---|---|---|---|
| $\mathrm{l}_A^{(base)}$ | 232 | 200 | 232 | 4 |

**Table V.16:** *Anomalous monitoree loop configurations compared to the reference* $l_A^{(base)}$

| $l_i$ | Anomaly name | $\Delta n_{in}$ | $\Delta n_{process}$ | $\Delta n_{out}$ | $\Delta n_{all}$ |
|---|---|---|---|---|---|
| $l_1$ | $\Delta n_{in} + 1$ | +1 | +0 | +0 | +0 |
| $l_2$ | $\Delta n_{in} + 2$ | +2 | +0 | +0 | +0 |
| $l_3$ | $\Delta n_{in} + 3$ | +3 | +0 | +0 | +0 |
| $l_4$ | $\Delta n_{out} + 1$ | +0 | +0 | +1 | +0 |
| $l_5$ | $\Delta n_{out} + 2$ | +0 | +0 | +2 | +0 |
| $l_6$ | $\Delta n_{out} + 3$ | +0 | +0 | +3 | +0 |

We define the anomalous program execution $e_A^{(anom)} = \{p, l^{(anom)}\} \in \mathcal{E}_A^{(anom)}$, with $p \in \mathcal{P}_A$ and $l^{(anom)} \in \mathcal{L}_A^{(anom)}$, as a deviation of the reference monitoree execution $e_A^{(base)} = \{p, l_A^{(base)}\} \in \mathcal{E}_A^{(base)}$. Such anomaly could correspond to an attack, which corrupts the CF to modify data manipulated by the monitoree: e.g. sensor data provided as input of the monitoree, or decision making data returned by the monitoree. Typically, we model anomalous execution by incrementing loop iterations of the different execution phases; this approach induces a modification in the execution time of the executing program. Thus, we introduce $c_A^{(base)}, c_A^{(anom)} \in \mathbb{N}^* * \mathbb{N}^*$, respectively the execution time for the programs $e_A^{(base)}$ and $e_A^{(anom)}$. Our motivation is to evaluate the granularity of our framework to detect anomalies. As an anomaly level criteria, we compute $\delta_A^{(anom)} \in \mathbb{R}^+$ the ratio of additional time spent in the anomalous program compared to the normal execution, for a set of 200 measurements:

$$\delta_A^{(anom)} = |\mu_{200}(c_A^{(anom)}) - \mu_{200}(c_A^{(base)})| \tag{V.19}$$

Table V.16 provides an overview of loop iteration combinations for anomalous executions; considering 6 anomalous configurations $\{l_1, l_2, l_3, l_4, l_5, l_6\}$, we represent the difference of added iterations compared to the base: $\forall j \in \mathbb{N}^*, j \leq 6, \forall i \in \{in, process, out, all\}, \Delta n_i^j = n_i^j - n_i^{(base)}$. For our set of tested programs, the Process phase takes a very long time to execute compared to In and Out phases; hence, we only modify $n_{in}$ and $n_{out}$ variables. Table V.17 shows the anomalous level criteria values ($\delta_A^{(anom)}$) for the set of tested programs. The panel of tested configuration corresponds to variations of the execution time from $3ms$ to $409ms$.

### HIDS Monitor Configuration

Measuring 200 samples, we compute $2.3\mu s$ and $2.7\mu s$, respectively the average and maximum execution times for the trace collector to generate and store the HPC traces

**Table V.17:** *Monitoree execution configurations with $T_c = 10ms$*

| $p$ | $\mu_{200}(c_A^{(base)})$ (ms) | $n_w$ | $d_A^{(min)}$ | Anomaly name | $\delta_A^{(anom)}$ (ms) |
|---|---|---|---|---|---|
| $p_{sort}$ | 979 | 98 | 7 | $\Delta n_{in} + 1$ | 3 |
| | | | | $\Delta n_{in} + 2$ | 7 |
| | | | | $\Delta n_{in} + 3$ | 10 |
| | | | | $\Delta n_{out} + 1$ | 6 |
| | | | | $\Delta n_{out} + 2$ | 12 |
| | | | | $\Delta n_{out} + 3$ | 19 |
| $p_{matmul}$ | 1,074 | 108 | 15 | $\Delta n_{in} + 1$ | 8 |
| | | | | $\Delta n_{in} + 2$ | 19 |
| | | | | $\Delta n_{in} + 3$ | 26 |
| | | | | $\Delta n_{out} + 1$ | 12 |
| | | | | $\Delta n_{out} + 2$ | 19 |
| | | | | $\Delta n_{out} + 3$ | 26 |
| $p_{bsearch}$ | 1,081 | 109 | 13 | $\Delta n_{in} + 1$ | 139 |
| | | | | $\Delta n_{in} + 2$ | 272 |
| | | | | $\Delta n_{in} + 3$ | 409 |
| | | | | $\Delta n_{out} + 1$ | 14 |
| | | | | $\Delta n_{out} + 2$ | 28 |
| | | | | $\Delta n_{out} + 3$ | 41 |

into the trace storage. Hence, we set $C_C = 3\mu s$ the WCET for trace collection.

For the trace collector, we empirically set the sampling rate to the value $T_C = 10ms$, since this configuration complies with the following requirements based on the approach described in Section V.3.3:

- $T_C \geq C_M + C_C$: we determine $T_C$ experimentally to test this inequality in different monitoring configurations.
- At least one counter is incremented for each monitored time window.
- $T_C = 10ms$ represents a realistic value compared to the execution time of the base monitored application reference: i.e. $T_C$ complies with the constraint (V.11) ($T_C << c_A \leq C_A$). We count indeed $n_w \in \mathbb{N}^*$ the amount of observation windows $T_C$ covering the execution of the program $p_A^{(base)}$: $n_w * T_C \gtrapprox c_A, n_w > 100$.

To implement the trace analyzer, and more particularly the trace predictor, we use TensorFlow-Lite software [119, 2], which runs inside a Linux guest OS user-level partition. It is a light-weight open source framework designed to perform inference on embedded systems.

With the method described in Section V.3.2 (Algorithm 1), we compute for each program execution configuration $e_A \in \mathcal{E}_A$, $d_A^{(min)} \in \mathbb{N}^*$ the minimum depth of the LSTM model. We generalize $d_{min}$ the minimum depth used for all our use-cases, so that it corresponds to the maximum value computed for the tested use-cases:

**Table V.18:** *Detection results summary*

| **Configuration** | | | | **Detection Results** | | | | |
|---|---|---|---|---|---|---|---|---|
| $n_{layers}$ | $n_{nodes}$ | $p$ | $A$ | **FPR** (%) | **WC** | **FNR** (%) | $F_1$ (%) | $F_1^{global}$ (%) |
| 1 | 30 | $p_{sort}$ | 7 | 2.5 | $\Delta n_{in} + 1$ | 2.5 | 97.5 | 86.6 |
| | | $p_{matmul}$ | 6 | 3.5 | $\Delta n_{out} + 1$ | 1.0 | 97.8 | |
| | | $p_{bsearch}$ | 5 | 6.5 | $\Delta n_{out} + 1$ | 58.0 | 56.6 | |
| | 50 | $p_{sort}$ | 6 | 5.5 | - | 0.0 | 97.3 | 93.0 |
| | | $p_{matmul}$ | 5 | 10.0 | - | 0.0 | 95.2 | |
| | | $p_{bsearch}$ | 4 | 22.5 | $\Delta n_{out} + 1$ | 6.0 | 86.8 | |
| | 100 | $p_{sort}$ | 6 | 5.5 | $\Delta n_{in} + 1$ | 4.0 | 95.3 | 90.1 |
| | | $p_{matmul}$ | 6 | 4.0 | - | 0.0 | 98.0 | |
| | | $p_{bsearch}$ | 5 | 10.0 | $\Delta n_{out} + 1$ | 34.0 | 75.0 | |
| | 150 | $p_{sort}$ | 6 | 03.0 | $\Delta n_{in} + 1$ | 5.5 | 95.7 | 90.3 |
| | | $p_{matmul}$ | 8 | 3.5 | $\Delta n_{in} + 1$ | 1.5 | 97.5 | |
| | | $p_{bsearch}$ | 5 | 6.0 | $\Delta n_{in} + 1$ | 36.0 | 75.3 | |
| 2 | 80 | $p_{sort}$ | 8 | 3.5 | - | 0.0 | 98.3 | 98.5 |
| | | $p_{matmul}$ | 9 | 4.0 | - | 0.0 | 98.0 | |
| | | $p_{bsearch}$ | 6 | 1.5 | - | 0.0 | 99.3 | |
| | 160 | $p_{sort}$ | 10 | 3.0 | - | 0.0 | 98.5 | 89.3 |
| | | $p_{matmul}$ | 7 | 11.5 | $\Delta n_{out} + 1$ | 3.5 | 92.8 | |
| | | $p_{bsearch}$ | 5 | 1.5 | $\Delta n_{out} + 1$ | 41.5 | 73.1 | |

$$d_{min} = \min_{\forall p_A \in P_A} (d_A^{(min)}) = 15 \tag{V.20}$$

Let define $n_{layers} \in \mathbb{N}^*$ the count of layers and $n_{nodes} \in \mathbb{N}^*$ the count of nodes per layers. We experimentally constrain the set of tested topologies for the LSTM based trace predictor:

- $n_{layers} \leq 2$: ML network topologies with more layers usually require big training data; thus they are more suitable for deep-neural network use-cases, which are out of scope for this work.

- $n_{nodes} \leq 200$: we make the count of nodes per layer $n_{nodes}$ vary between 30 and 160, to limit the trace prediction time impact on detection latency (i.e. limiting the value of $C_M$).

### Impact of the Model Topology on the Detection Accuracy

Table V.18 summarizes the detection accuracy results for all tested programs in function of the LSTM model topology. $A \in \mathbb{N}$ corresponds to the coefficient value in the range $[1, 5]$, which induces the best detection results for a given use-case (i.e. program and trace predictor topology). We note $F_1$ the F1-score (Section II.1.3) computed for a single configuration, evaluating a single trace analyzer model built for a given monitoree

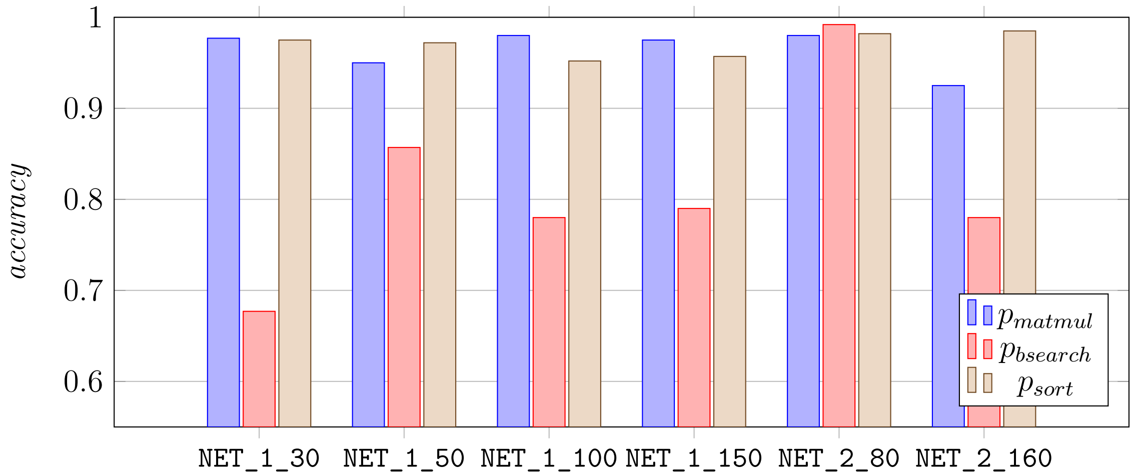


**Figure V.7:** *Impact of the network topology on detection accuracy results*

program and network topology. We introduce the derived F1-score metric $F_1^{global}$, which involves the detection measurements of all tested models for a given network topology configuration. On Table V.18 as well as on Figure V.7, we only show the results corresponding to the worst anomaly use-case (WC). We note that the worst-case systematically correspond to the anomalous program with lowest execution time variations: with $\Delta n_{in} = 1$ or $\Delta n_{out} = 1$.

Figure V.7 shows the detection accuracy for the tested use-case configurations. We cannot highlight a direct and systematic relation between the network topology and the detection results, such as the greater $n_{layers}$, the more accurate the results. Nevertheless, we note that the configuration with $i(n_{layers} = 2, n_{nodes} = 80$ offers the best results, considering the three tested programs (accuracy above 95% for all). We instead show that the detection accuracy varies depending on the monitored program: while $p_{matmul}$ and $p_{sort}$ show accuracy above 90% in all configurations, the accuracy varies between less than 70% and more than 85% for $p_{bsearch}$. From Table V.18, we notably show that for $p_{matmul}$ and $p_{sort}$, the false-positive and false-negative rates are low (respectively between $[2\%, 11\%]$, $[0\%, 5\%]$).

### Impact of the Model Topology on the Detection Latency

The topology of the LSTM model has a direct impact on the detection latency: the more complex the topology, the more important the impact (i.e. $C_M$ increases). Table V.19 shows the worst-case and average measured execution times to make a prediction, for a selection of network topologies. We found that the WCET for trace collection is negligible compared to the time for trace analysis: i.e. $C_C = 3\mu s << 1ms \leq C_M$. To simplify, we approximate $C_C + C_M \approx C_M$. Figure V.8 shows the distribution for 200 measurements. We observe that for all tested topologies but one, the condition (V.14) is respected. Hence, for our use-case, we do not consider the case $(n_{layers} = 2, n_{nodes} = 160)$ for deployment: this is fine, since we generally observed better detection accuracy results in other topology configurations.

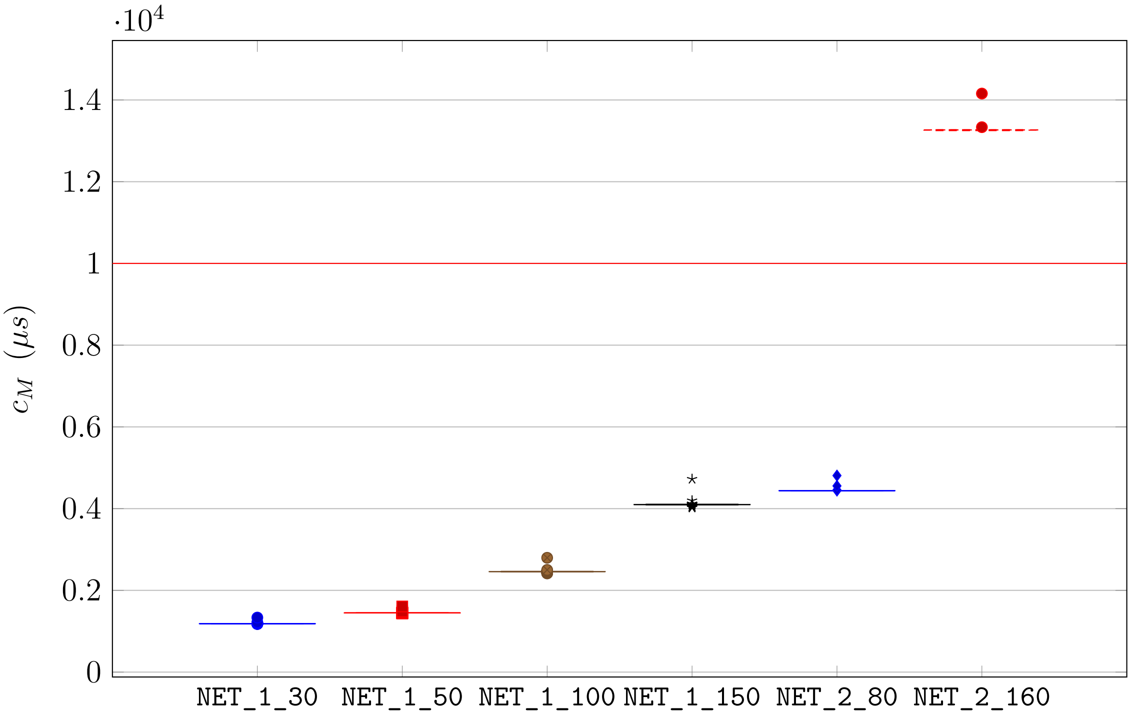


**Figure V.8:** *Execution time of the trace analyzer ($c_M$)*

In particular, increasing the depth $d$ of LSTM nodes increases the count of operations for a prediction. Figure V.9 represents the evolution of the execution time of the trace analyzer in function of the depth $d$. We note $c_M^d$ the observed execution time for a network with LSTM nodes of depth $d$. Compared to the maximum value $C_M^{15}$ observed for the reference $d_{min} = 15$, the figure shows the added execution time $\Delta c_M^d = c_M^d - c_M^{15}$ with $d \in \mathbb{N}^*, d_{min} = 15 < d \leq 20$. For the range of tested $d$ values, we observe a linear increasing impact on the execution time. Logically, we note that the coefficient of the linear relation increases, when the network contains more LSTM nodes (i.e. 100 to 150 nodes).

**Table V.19:** *Trace analyzer execution time in function of the LSTM network topology (for* 200 *measurements)*

| **Network Topology** | | **Time measurements (ms)** | |
|---|---|---|---|
| $n_{layers}$ | $n_{nodes}$ | $C_M$ | $\mu_{200}(c_M)$ |
| 1 | 30 | 1.3 | 1.2 |
| 1 | 50 | 1.6 | 1.4 |
| 1 | 100 | 2.8 | 2.5 |
| 1 | 150 | 4.7 | 4.1 |
| 2 | 80 | 4.8 | 4.4 |
| 2 | 160 | 14.1 | 13.3 |

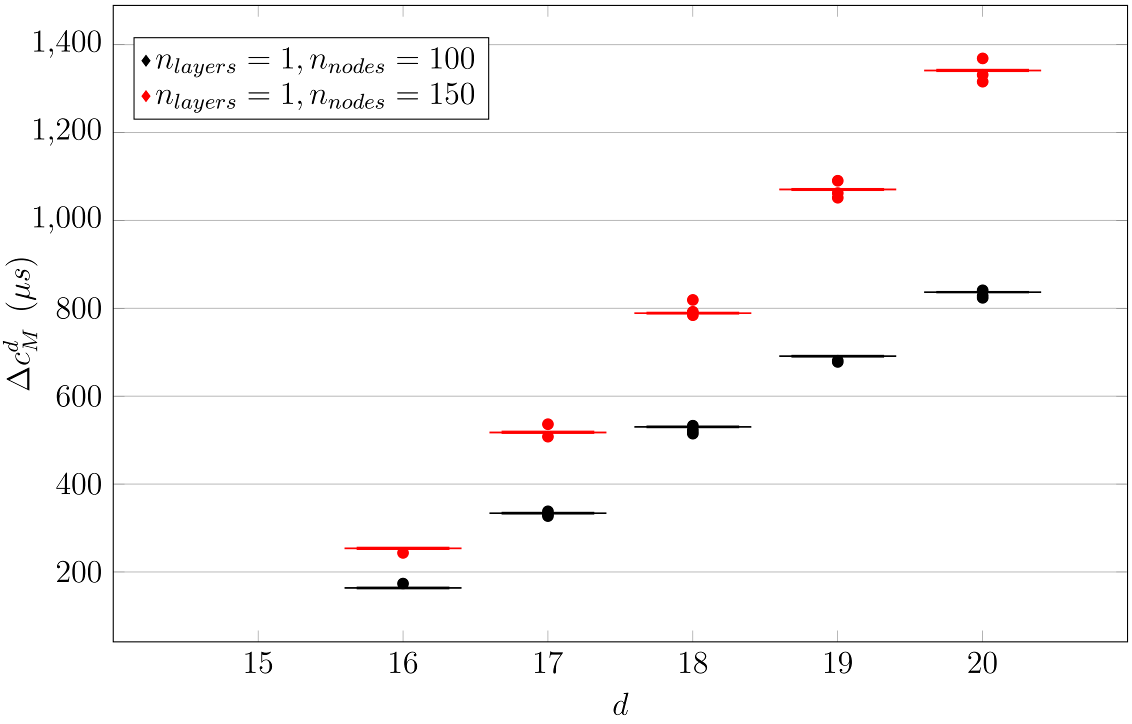


**Figure V.9:** *Impact of $d$ the depth of LSTM nodes on the execution time $c_M$ (added execution time compared to $d_{min} = 15$)*

### Discussion on the Deployability of the Intrusion Detection Solution

Referring to the real-time analysis of the HIDS solution, our experiment shows that the HIDS framework complies with the deployability condition (V.14), considering the trace collection period $T_C = 10ms$, for five of the six tested LSTM network topologies of the trace analyzer. This means that we should either consider network topologies with less nodes than the configuration ($n_{layers} = 2, n_{nodes} = 160$), or increase the period $T_C$. On one hand, increasing $T_C$ mechanically causes a longer detection latency $w_{detect}$ (V.16). On the other hand, we expect that increasing $T_C$ to increase the complexity of the LSTM network topology is not a good strategy to improve the detection accuracy of the HIDS:

- in our experiment scope, we cannot analytically determine how incrementing $T_C$ would affect – positively or negatively – the detection accuracy.

- we note from our results that increasing the complexity of the LSTM network topology does not systematically improve the detection accuracy, for a given monitored application.

From our experiment, because of the poor detection accuracy in two of three monitored applications using the most complex tested topology, we suggest to configure network topologies consisting of comparatively lower amounts of nodes.

For the deployable network configurations, we observe an important time margin between $C_M$ and $T_C = 10ms$. The HIDS would still respect the deployability condition when decreasing $T_C$ to $5ms$, as $C_M + C_C < 5ms$ for the five relevant network topologies. Nevertheless, we would need to evaluate the impact of this new trace collection period on the detection accuracy of the solution.

To reduce the detection latency, we could decrease the depth of LSTM nodes ($d = 15$ in our experiment). However, such modification might cause a degradation of the detection accuracy because of the possible ambiguities in the training dataset (as explained in Section V.3.2). Alternatively, we may improve the detection accuracy by increasing the value of $d > 15$; the trace predictor would potentially be able to capture longer temporal patterns of execution. However, we have shown the significant impact of incrementing $d$ on the WCET $C_M$, which mechanically increases with the amount of LSTM nodes of the network (Figure V.9). For network topologies requiring $d > 15$, We would probably have to increase $T_C$ to comply with the deployability condition (V.14); i.e. as discussed above, the modification of $T_C$ may have a negative impact on detection accuracy.

We tested relatively simple LSTM network topologies (less that 160 nodes in total). As future improvement, we could adapt the trace predictor configuration further, i.e. combining several layers containing various amount of nodes. We would need to evaluate the impact on HIDS deployability, as well as on detection latency and accuracy. Another strategy is to adapt the implementation of the trace predictor to reduce the detection latency: because of the simplicity of the LSTM network, we could directly implement the LSTM network in bare-metal software (i.e. not using API provided by standard libraries) to reduce $C_M$. Our solution could also involve hardware acceleration, e.g. with GPU or FPGA support.

### V.3.5 Conclusion

**To our knowledge, this is the first work to explicitly evaluate the cost for online ML assisted HIDS monitoring towards the deployment into certified embedded MCS.** The solution is transparent, as it runs concurrently to the monitoree on a different set of CPU cores, and because it does not require program instrumentation. The HIDS observes the monitoree application using hardware events traced via hardware internal counters of the processor. We formalized the representation of our solution with a time analysis of the system, and defined a set of parameters for the system integrator to tune the HIDS, in function of runtime cost and security (i.e. anomaly detection accuracy and latency) requirements.

Our experimental results first demonstrate the practicability of our HIDS framework on a use-case application. Second, they highlight the impact of the HIDS configuration – in particular of the trace collection period and the trace analysis execution time – on the performance of the solution:

- the hardware cost (i.e. CPU utilization for monitoring) increases with increasing trace analysis execution time and decreasing trace collection period.

- the detection efficiency (i.e. anomaly detection accuracy and latency) improves with increasing trace analysis execution time and increasing trace collection period.

However, our solution comes with several limitations related to its implementation. By design for running HPC tracing from user-space, the framework requires the monitoree application to run alone on a single CPU core. We set the monitoring tasks to the highest priorities to run on the monitoring CPU core, to assure consistent HPC tracing and upper bounded detection latency. Nevertheless, the system integrator can still use remaining CPU time to perform background activity; i.e. adding tasks with low priority. We also consider that the readability of HPC traces from the monitoree represents a further security exposure: an adversary controlling the monitoree could attempt to hide malicious activity by checking the corresponding footprint on HPC traces to avoid detection.

## V.4 Conclusion on Machine-Learning Assisted Anomaly Detection for Embedded Mixed-Criticality Systems

Through this chapter, we addressed statistical and ML assisted methods to deploy HIDS into industrial embedded MCS. We developed two main HIDS approaches:

- **An offline HIDS based on system call and HPC tracing.** We introduced a safety-aware framework to monitor system calls into a RTOS. In addition, to enhance the anomaly detection, we integrated contextual hardware data in system call traces using HPC counters. We experimentally confirmed the relevance of this approach for detecting more anomalous executions, compared to solutions based on system call related traces only.

  Despite the good detection results, the framework still induces the limitation that it only covers monitoree programs, which execute system calls. Consequently, it reduces considerably the scope of use-case programs for a valid evaluation; producing a substantial set of applications representing realistic – potentially complex – use-cases is difficult, especially as we work with a proprietary RTOS.

- **A transparent online HIDS based on HPC tracing.** We proposed an evaluation of a HIDS framework supporting online ML-assisted detection. For the first time to our knowledge, we introduced a HIDS solution explicitly designed for embedded MCS, which is based on HPC tracing and online ML assisted detection. We provided an analytical and experimental evaluation of the solution's performance, in regards to its detection efficiency and hardware cost, towards the deployment on an industrial platform.

Our experiments show that the complexity of the implementation of the ML model directly affects its applicability: the more complex the ML model, the higher the time overhead for analyzing a set of traces. As a workaround, the efficient usage of hardware accelerated ML frameworks or optimized low-level routines and libraries could improve the performance of a given ML model, by reducing the analysis time overhead.

We run our experiments on a simplified setup, monitoring a unique single-threaded application. In future work, we could develop support multi-threaded and multi-core applications. We could also adapt the monitoring solutions to monitor several applications simultaneously.

We can use the ML-based HIDS for **passive monitoring** on a given task in an embedded MCS: i.e. observing and logging the detection of anomalies without making adjustments on the monitoree's execution. For example, we could store detection related data into a protected log storage, for a later analysis of the logs by a human or a qualified software entity which can legitimately apply corrective measures. However, for now, ML based monitoring approaches cannot be used for **active monitoring**; we define active monitoring as an intrusive method to modify the execution of a system component (e.g. applying recovery actions as described in Section IV.4.3), upon detection of an anomaly in the execution of the monitoree. The use of ML components in a safety-critical system is indeed a current research topic [149, 86, 146, 7]: because an anomaly can corresponds to a false-positive, it is difficult to reliably justify and validate the information of a detected anomaly to base future high-critical decisions. Hence, ML based monitoring features cannot directly be integrated into system operation with control abilities on the execution of the monitored component.

Finally, as ML based HIDS solutions are limited to passive monitoring, we could combine such approach with additional active monitoring methods, to build multi-security modes of executions. Notably, as the ML assisted solutions described in this chapter introduce limited intrusiveness and low performance impact in the system, they could be used as background security tasks to raise early security alerts; when they detect an anomaly, the system would switch to a suspicious execution (i.e. active monitoring) mode to apply more intrusive security checks, in order to confirm or infirm the identification of a threat in the system. Specification-based anomaly detection methods are typically suitable to perform active monitoring. We develop such approach with a CF based detection framework in Chapter VI.

VI

# A Safety-Aware Control-Flow Integrity Framework for Embedded Mixed-Criticality Systems

In this chapter, we introduce a CF monitoring framework which leverages hardware-level processor tracing to transparently monitor the execution of a given monitoree application; it uses ARM CoreSight, a common hardware tracing feature in ARM-based SoC. With this solution, we develop the following contributions:

- We implement CFI checking through state-of-the-art methods described in the literature survey (Section III.2.1).
- We propose a predictable periodic-server based solution to safely integrate the hardware-assisted framework into a MCS. Our approach mitigates the potential performance overhead for CF tracing, by controlling a partial CF monitoring coverage at runtime.
- We introduce an anomaly detection service to monitor the full scope of the task's execution, identifying anomalies from the observation of inter-arrival times of CF monitoring instances.

We validate our monitoring framework on an industrial MCS platform using a set of applications from TACLeBench benchmark. Defining metrics to assess the trade-off between security and time overhead, we evaluate the performance impact and validate the anomaly detection system for the set of tested applications.

The remainder of this chapter is organized as follows. Section VI.1 introduces our CF monitoring system architecture. Section VI.2 defines task and system model as well as problem statement. Section VI.3 presents our predictable CF monitoring solution. After describing our experimental setup in Section VI.4, we evaluate the performance overhead of our framework for full runtime CF coverage on a set of representative and pessimistic applications (Section VI.5). Section VI.6 focusses on the security evaluation. Section VI.7 discusses lessons learned and alternative security deployment strategies using our framework. Section VI.8 gives an overview of related work. Finally, Section VI.9 summarizes this work.

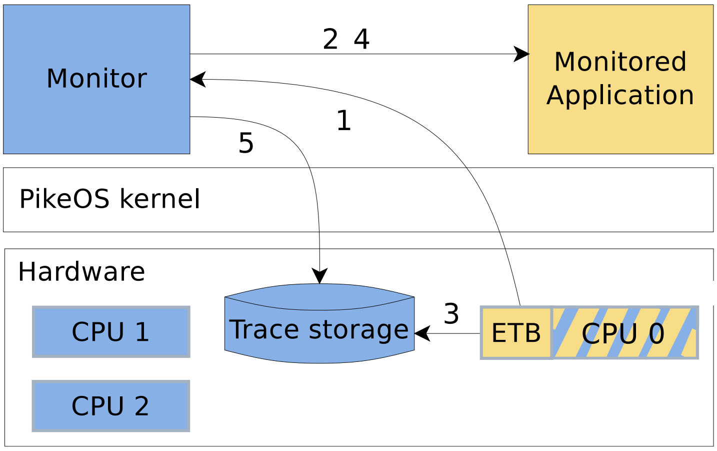


**Figure VI.1:** *System overview*

# VI.1 Hardware-Assisted Control-Flow Monitoring Framework

## VI.1.1 Control-Flow Monitoring Framework Overview

### Mixed-Criticality System Software

We implement our system using the PikeOS [128] real-time hypervisor together with the framework described on Figure VI.1. The framework combines a partitioned architecture with a separation kernel [137] to assure freedom from interference and independence between system tasks: a task can only access system resources after explicit allocation at system design. Such architecture, in which the CFI monitoring is separated from the monitored application, is well suited for a MCS because of two reasons—(i) the adversary is not able to interfere with the CFI monitoring due to the separation, and (ii) at runtime, CFI monitoring tasks and other safety-critical tasks are restricted to their allocated resources, thereby preserving their independence. The PikeOS kernel implements a preemptive fixed-priority thread-level scheduling with static bounds for the range of thread priorities. The system integrator assigns ranges of priority values for user-level components at system design; at runtime, a user-level component can modify the priority of one of its thread, or a thread belonging to another task within its pre-defined range of values.

### ARM CoreSight Hardware Feature

ARM CoreSight [16] is a common hardware feature supported on many modern ARM-based SoCs (e.g. Xilinx Ultrascale+, NXP i.MX8) for tracing the execution of a program at instruction level in real-time. Each core on the SoC is attached to an embedded trace macrocell (ETM) responsible for trace collection. The traces from multiple ETM are merged and stored in an embedded trace buffer (ETB). When the queue reaches a configurable threshold, the ETB can raise a signal directed to a core as an interrupt or to an external debugger. The software or external debugger can use this trigger to copy/flush the traces from the ETB queue to system memory or to external pins. ARM CoreSight collects traces in form of *packets*, either containing branch decisions for direct control-flow transfers (i.e., branch taken or not taken) or the target address for indirect transfers. Packets additionally include per-thread UID that enable precise execution tracking of different (potentially multi-threaded) applications. Based on the packets provided by CoreSight and with additional binary information about the executed application, it is possible to reconstruct the entire execution control-flow including the exact sequence of branches taken. For our work, we use CoreSight to monitor the control-flow of the program at runtime.

#### System Applications

The monitored application is a single-threaded periodic application running with fixed-priority, on the CPU core 0 where we enable CoreSight tracing. It executes the same routine program periodically.

The monitor is composed of three main threads of execution:

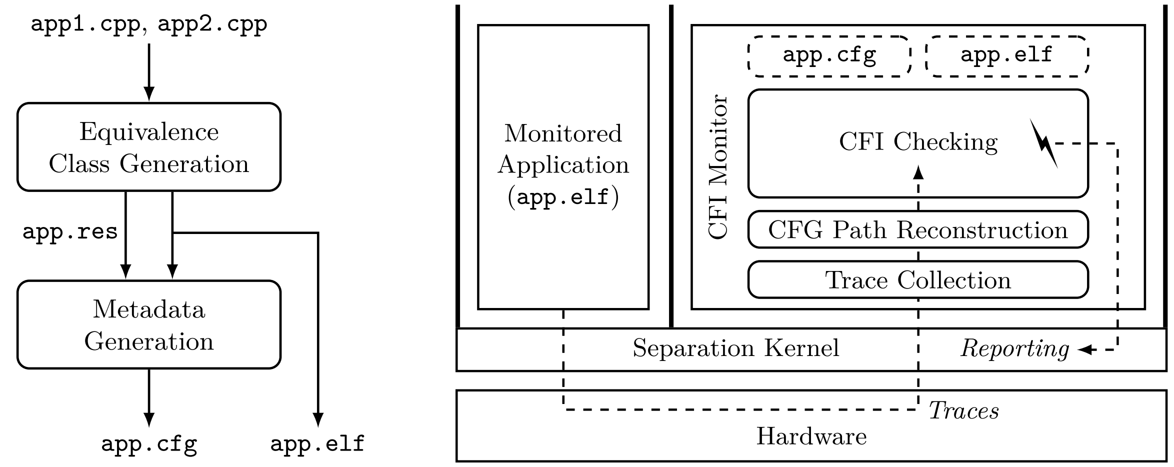
- **Server** (CPU core 1) – It decides whether an upcoming monitoring request is to be handled or rejected.
- **Trace collector** (CPU core 1) – this thread copies the traces from CoreSight local buffer to a predefined storage area in main memory.
- **Spinner** (CPU core 0) – It preempts and stops the execution of the monitored application during trace collection.
- **Trace analyzer** (CPU core 2) – This application reads traces from the trace storage in main memory to reconstruct the corresponding CFG path and perform CFI checking.

We allocate the same fixed priority for all threads running on CPU core 1, while spinner's priority can take several values at runtime. The trace analyzer runs in parallel of other monitored and monitoring threads on a dedicated CPU core 2 with a fixed priority.

### VI.1.2 Control-Flow Monitoring Mechanisms

#### Monitoring Framework Overview

The architecture of our CFI monitoring solution is shown in Figure VI.2. During *runtime monitoring*, after a stream of traces has been collected by the monitor, the current CFG path of the monitored application thread is reconstructed by decoding the trace packets into the sequence of executed basic blocks. Using the reconstructed path and the trace's per-thread identifier, the monitor maintains per-thread shadow stacks for backward-edge CFI enforcement. For the protection of forward edges, the monitor validates that the source and destination addresses of transitions between basic blocks belong to the same pre-computed equivalence class. We compute the per-application equivalence classes



(a) *Static pre-processing*

(b) *Runtime monitoring*

**Figure VI.2:** *Control-flow integrity monitoring design*

according to a type-based approximation of the CFG during a dedicated *static pre-processing* phase and load them into our CFI monitor as metadata before starting the monitored application.

For our work, we assume the SK does not randomize the memory layout of monitored applications and statically inferred addresses are still valid during runtime[1]. To ensure the security provided by our solution, we implement the forward-edge and backward-edge CFI policies according to [151] and [3], respectively. Both policies are well-established and known to thwart code-reuse attacks effectively [28, 29].

### Static Pre-processing and Framework Initialization

The pre-processing phase statically computes a type-based approximation of the CFG based on the concept of equivalence classes. We generate equivalence classes only for indirect jumps and calls, but handle calls to ordinary functions, calls to virtual methods (C++ only), and intra-function jumps differently.

For ordinary functions, we define equivalence as the signature of a call site matching the signature of the called function, where the signatures are defined as a combination of the function's return type and parameter types. Hence, two functions `int add(int a, int b)` and `int sub(int a, int b)` are considered equivalent by our CFI monitor and may be called from any indirect call site having the same signature. For virtual methods, we define equivalence analogously with the addition of taking class hierarchies into account. Two functions `int Math.add(int a, int b)` and `int Math.sub(int a, int b)` are considered equivalent if and only if they belong to the same class `Math` or any derived or base class of `Math`. Lastly, for simple jumps, we define equivalence based on code locality, i.e., restricting jumps to the switch statement or function body they belong to.

[1] This is not a limitation of our CFI monitoring solution, as providing randomization details to the monitor would suffice to support memory layout randomization.

In our final metadata, we store the equivalence classes as a simple lookup table indexed by the addresses of branch targets. For every branch target within the table, we store the list of call site addresses allowed to branch to the that target—effectively representing the equivalence class the target belongs to. Hence, for the two functions `add()` and `sub()`, the table contains two identical entries specifying all call sites allowed to branch to `add()` and `sub()`.

We generate the metadata in a two-step process during the compilation of the application. We adopt this process, as computing equivalence classes at compile-time is more precise than a binary-only computation [155]. Figure VI.2a depicts the compilation and generation of the binary `app.elf` and the metadata `app.cfg` for an application consisting of two source code files `app1.cpp` and `app2.cpp`.

**Step 1: Equivalence Class Generation.** The first step is built upon the LLVM compiler infrastructure and computes the equivalence classes for indirect calls to ordinary functions and virtual methods. During this step, we essentially generate a version of our metadata containing descriptive identifiers instead of actual address locations. Branch targets (i.e., functions and virtual methods) are represented as string literals by their names and call sites by the offset into their parent functions. Indirect jumps do not require compile-time information and are only processed during the second step.

For the generation of equivalence classes, we utilize the type metadata mechanism already present in LLVM [150]. The mechanism computes the equivalence classes according to [151] and as required by our definitions above. Because the type metadata is only available in LLVM's intermediate representation we deploy a custom LLVM pass that passively collects and stores the equivalence classes to a temporary file `app.res`. To resolve the equivalence on an application-wide level, this step must be performed on a combined version of the application's intermediate representation (i.e., in our example, simultaneously for `app1.cpp` and `app2.cpp`) or during link-time optimization. The data extracted by our LLVM pass is already represented in the form of our metadata, i.e., for every branch target, we have a list of possible call sites. Because the extracted data still contains equivalence classes represented as string literals, we translate the data into actual address locations in the following, second pre-processing step.

**Step 2: Metadata Generation.** The second step generates the final metadata file `app.cfg` based on a binary analysis of the compiled application `app.elf` and the extracted equivalence classes `app.res`. We use Capstone [30] to inspect the application binary and translate the string-based equivalence classes to address-based equivalence classes. The extracted data represents equivalence classes in form of a lookup table indexed by function and virtual method names, i.e., the targets of indirect call sites. Each entry in the table contains a list of call sites allowed to branch to that function or virtual method. The call sites are identified by the name of their parent function and their offset within the function. Looking up the function names and virtual methods in the disassembled application binary, we are able to translate the entire lookup table into actual address locations.

As the first pre-processing step only generates equivalence classes for ordinary functions and virtual methods, we still have to generate legitimate branch targets for indirect jumps. For this, we use the binary inspection to identify the type of indirect jumps (e.g., a switch statement) and then extract all addresses of legitimate jump targets within the scope of the identified type. Like this, a switch statement is only allowed to perform jumps within the switch body. Unfortunately, for some indirect branches, the target addresses cannot be identified statically, because, for example, they are inferred from program input during runtime. In such cases, like for other state-of-the-art forward-edge CFI solutions, we refrain from generating metadata, leaving those branches unchecked during runtime. Note that indirect jumps to functions, as generated by tail call optimizations, are handled the same way indirect calls are handled.

**Step 3: Framework initialization.** After the pre-processing phase, the application is ready to be executed normally with our CFI monitor passively verifying its indirect CF transfers. To start monitoring the application, the generated CFG metadata file together with the application binary are loaded into the monitor. The monitor itself first initializes the CoreSight subsystem by configuring the ETM to generate traces whenever threads of the monitored application are schedule on the corresponding cores (CoreSight does not trace applications by default). Next, the monitor invokes the application, which then gets initialized and scheduled by the SK as usual. At this point, our monitor moves into a waiting state until the first traces are ready to be processed. As shown in Figure VI.2b, processing traces is done in two main steps, the *trace collection* and *trace analysis* with CFI checking.

### Trace Generation

We leverage ARM CoreSight to trace the runtime CF of the monitored program, i.e. sequences of basic-blocks, which we define as blocks of linear instructions ending by a CF transition such as exception, branch, or return instruction. At runtime the hardware generates CF traces in the Embedded Trace Buffer (ETB). We define a threshold lower than the maximum buffer size in the ETB: when the ETB write pointer reaches the threshold, CoreSight subsystem raises an ETB full hardware interrupt (monitoring request) for the buffer to be read and emptied. Traces are generated even when the ETB is full; in this case, we either loose them or they overwrite the old traces in the buffer.

### Trace Collection

Figures VI.1 and VI.3 describe the following steps applied by the system when a monitoring request is raised by CoreSight hardware interrupt:

0 The monitored application executes alone on CPU core 0, because its priority is higher than spinner thread's one.

1 CoreSight framework generates a hardware interrupt, routed to the server application running on CPU core 1. The server thread decides whether the monitoring request is ignored or handled. Section VI.3.1 describes the decision process.

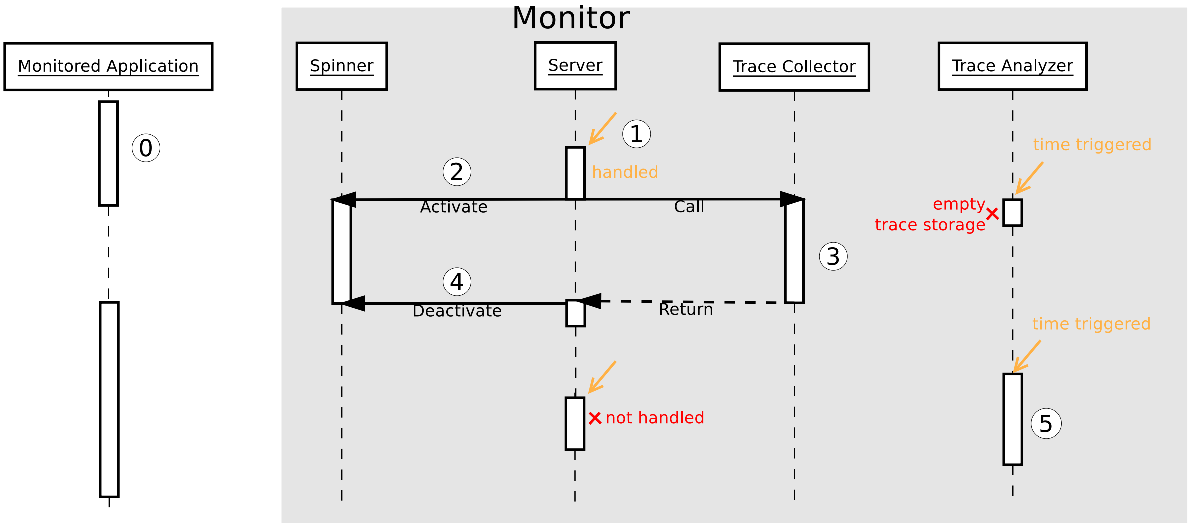


**Figure VI.3:** *Sequence diagram for monitoring request handling*

2 The monitoring request has been accepted. First, the server indirectly stops the monitored application by setting spinner thread's priority to a higher value than the one of the monitored application; i.e. spinner thread preempts the monitored application. Second, it disables CoreSight tracing. If we first disable CoreSight tracing and then pause the monitored application execution, the application may execute CF transitions that would not be traced after tracing has been disabled and until the application is paused. The server then calls the trace collector, and waits for it to return.

3 The spinner thread executes alone on CPU core 0. It indefinitely suspends the core by executing a WFI (wait for interrupt) instruction in a loop. At the same time, the trace collector reads traces from ETB CoreSight local storage and copies them to the trace storage in main memory. Thus, a monitoring request cannot be triggered during the trace collector execution, since the monitored program has been stopped.

4 After the trace collector has returned, the server stops spinner thread by resetting its priority to its initial value.

5 The trace analyzer is triggered by a statically defined periodic timer. If the trace storage in main memory contains data, it analyzes them, performing CFG path reconstruction and CFI checking (more details in the following section).

In the time to handle an ETB full interrupt (i.e. when the interrupt is triggered until the server stops the monitored program), the monitored application could run CF instructions to be stored in the ETB. CoreSight subsystem would generate further traces to be stored into the ETB, until the buffer is full. Once the ETB is full, upcoming traces are lost. Thus, the ETB threshold configuration should allow a sufficient remaining buffer capacity to contain the maximum amount of traces that can be generated during the time to handle an interrupt (this configuration step is detailed in Section VI.4). An ETB threshold misconfiguration is a resource allocation issue: PikeOS system raises a health-monitoring event to take further action, for example stopping the monitored program and the monitoring framework.

### Trace Analysis with CFI Checking

The purpose of the trace analysis is to determine whether the observed CF traces correspond to a normal or a malicious execution of the monitored program. This operation can split into 2 main steps: runtime CFG path reconstruction from CoreSight traces and CFI checking with the reference defined in the pre-processing stage.

**CFG Path Reconstruction.** The traces generated by the ETM and received by our monitor include common data such as the application thread's UID and the virtual address of the recorded instruction. Additionally, traces of indirect branches include the destination address, while direct branches omit the address and only include the branch decision, i.e., whether the branch was taken or not. Other data in the traces is not relevant to our CFI monitor.

We use the data provided in the traces to reconstruct the path taken by the application through the CFG. For this, our monitor utilizes the OpenCSD library [99] that takes the application's binary as input and generates a sequence of basic blocks resembling the application's execution path. A basic block is identified by its start and end address, as taken from the application's binary. The end address of a basic block always points to a set of control-flow instructions such as direct or indirect branch (including function returns). The reconstruction also preserves the UID and attaches them to basic blocks so that our monitor is able to enforce CFI on a per-thread granularity. Finally, the reconstruction sequentially emits the basic blocks to the CFI checking, where CFG conformity is validated.

**CFI Checking.** With the sequence of executed basic blocks, our monitor is able to perform per-application forward-edge and per-thread backward-edge CFI checking. For forward-edge CFI, the monitor consults the CFG metadata provided through the pre-processing phase. For every basic block terminating in a direct or indirect function call, the monitor looks up the next basic block's start address in the metadata. This lookup yields a list of call site addresses that are allowed to branch to the next basic block. To validate CFG conformity, the monitor verifies that the current basic block's end address (i.e., the call site address) is present in the list. If the address is not found, a CFI violation has been detected.

For backward-edge CFI, the monitor does not rely on static metadata, but maintains a per-thread shadow stack. The monitor first evaluates the UID attached to a basic block and then looks up the shadow stack corresponding to the identifier (i.e., the thread). Next, for every basic block terminating in a direct or indirect function call, the call's return address is pushed onto the shadow stack. For every basic block terminating in a function return, the following basic block's start address is compared against the top-most address stored on the stack. If the addresses are not equal, a CFI violation has been detected. Otherwise the return is valid and the top-most address is removed from the stack. To cope with tail call optimization or otherwise shortened return sequences (e.g., `longjmp`), where the top-most address on the shadow stack is not equal to the next basic block's start address, the monitor pops addresses from the stack until a match is found or the bottom of the stack is reached. Like this, the monitor is able to ignore all return addresses skipped by the `longjmp`.

**CFI Violation Reporting.** Finally, if our monitor detects a CFI violation, it notifies the SK, which then is able to react accordingly, as described in Section IV.4.3.

## VI.2 Model and Problem Statement

The objective is to formalize the problem of integrating the security monitoring framework in a mixed-criticality system with hard timing constraints.

### VI.2.1 Threat Model

We consider any threat targeting the monitored application's CF at runtime. A classical example is a memory corruption exploit such as Return-Oriented Programming (ROP)[131]. Because memory errors are unlikely to hide in certified software, non-critical open software (e.g. multimedia, networking) are easier targets in mixed-criticality systems. We also scope intrusions bypassing standard CF defenses, such as Data-Oriented Programming [71]. Similarly to ROP based threats, the attack exploits a memory corruption to finally perform privilege escalation; it can bypass CF defenses, since it hijacks indirectly the CF through the modification of non-control data like function arguments or internal parameters such as loop conditions. Hence, we identify a threat as a misuse of a program at runtime, so that it deviates from its intended execution.

Any task in the MCS can be exposed to attacks and our framework must be able to monitor any task in the system. Though, we prioritize threat scenarios where the attack is initiated from low-criticality software, because in a MCS, these software are generally more accessible to an adversary (e.g. networking, user interfaces...) than high-critical software. Additionally, since high-critical software follow rigorous processes for the development and verification/validation, they are less likely to have software bugs compared to non-critical software.

### VI.2.2 Task Model

#### Task Definition

We define a task as a stream of jobs: $\tau_i = \{J_{i,1}, J_{i,2}, ..., J_{i,j}, ...\}$. We represent the real-time task $\tau_i$ with implicit deadlines as $\tau_i = (C_i, T_i, P_i)$, where $P_i$, $C_i$ respectively represent the priority and the worst-case execution time (WCET) for a job $J_{i,j}$, and $T_i$

the minimum inter-arrival time between successive jobs $J_{i,j}, J_{i,j+1}$ of the task $\tau_i$. We note $a_{i,j}$, $s_{i,j}$, and $d_{i,j}$, respectively the arrival, start, and deadline times of the job $J_{i,j}$.

#### Monitored Application

We model the monitored application with a periodic task $\tau_A$, so that $\tau_A = (C_A, T_A, P_A)$. For this task, a job missing its deadline potentially leads to reduced availability and poor user experience: e.g. delayed frames in a rear-view camera, discontinuous video playback.

#### Secure Monitor Application

The system executes the secure monitor after trapping a hardware interrupt, which notifies that CoreSight Trace buffer is full (Section VI.1.1). Thus, we represent the secure monitor with an event-triggered task $\tau_M$, so that $\tau_M = (C_M, T_M, P_M)$. The secure monitor's execution is intrinsically linked to the monitored application's execution, since the amount of triggered monitoring requests depends on the amount of branch instructions (e.g. function call, direct branch, return) executed by the monitored application at runtime. Hence, once the monitored application has finished its execution and until its next release time, no more monitoring requests will be generated: $\tau_A$ idle $\Rightarrow$ $\tau_M$ idle (Property VI.1).

The inter-arrival time of monitoring requests depends on the rate of generated ARM CoreSight interrupts (i.e. full trace buffer events), which depends on the execution path of the monitored program. Since the minimum buffer filling rate corresponds to the minimum inter-arrival time, $\tau_M$ is a sporadic task.

The secure monitor $\tau_M$ is a soft task: a job of $\tau_M$ missing its deadline does not provoke serious consequences, though it affects security monitoring coverage. A monitoring request should either be immediately served or ignored. It cannot be delayed: if $\tau_A$ continues to execute after the trace buffer is full (i.e. after a monitoring request arrives), old monitoring traces in the buffer will be overwritten by newly generated traces. Therefore, $\tau_M$ should be able to preempt and cause temporal interference on the monitored task $\tau_A$. Since we have to contain the interference so that $\tau_A$ can meet its deadlines, even under the worst-case monitoring (minimum inter-arrival rate), we propose to host $\tau_M$ in a periodic server (description in Section VI.3.1).

### VI.2.3 System Model

For our analysis, we simplify the framework described in section VI.1 by considering only one processor core, running $\tau_A$ the monitored task and $\tau_M$ the monitor. As described in Section VI.1.1 with Figure VI.3, the spinner is the actual thread running on the same CPU core as the monitored application to prevent it from running while trace collection is executing on an other core. The spinner thread's execution depends on the trace collector thread: the server enables the spinner at the same time as the trace collector and deactivates it when the trace collector returns. Thus, to simplify the representation, we identify the monitor task as the trace collector.

**Table VI.1:** *Evaluation metrics for one period of execution of Task $\tau_A$*

| | Metric | Definition |
|---|---|---|
| **Performance** | $r_M^A = \frac{B_M^A}{C_A}$ | $\tau_A$'s response time slowdown |
| **Security** | $r_{cov}^A = \frac{n_{mon}^A}{n_{mon}^A + n_{ignored}^A}$ | runtime monitoring coverage with $n_{ignored}^A$, $n_{mon}^A$ respectively the count of ignored and accepted monitoring requests |
| | $w_{uncov}$ | maximum time window without monitoring |
| | $w_{detect}$ | maximum time window from the time the attack starts until it is detected |

In this system at any time of execution, only one of $\tau_A$ and $\tau_M$ tasks is executing at the most. Both tasks execute using fixed priority based preemptive scheduling (e.g. Rate Monotonic).

## VI.2.4 Problem Statement

### Objective

Our objective is to propose a sporadic task guarantee test to decide whether an upcoming job request of $\tau_M$ can be taken or ignored, in function of pre-identified monitoring coverage issues (Section VI.2.4), without compromising the timing constraints of task $\tau_A$. In summary, we target the following objectives:

- Limit and predict the monitoring overhead to guarantee task $\tau_A$'s timing constraints.
- Minimize the attack surface on the execution of $\tau_A$.

Table VI.1 introduces performance and security metrics to evaluate our monitoring framework.

### Monitoring Time Overhead

We define a bounded execution time for $\tau_M$, during the execution of a job of $\tau_A$ (i.e. during the period $T_A$). Therefore, we assign the budget $B_M^A$ to the monitoring task $\tau_M$: $B_M^A = N_M^A * C_M$, where $N_M^A \in \mathbb{N}^*$ is the maximum amount of jobs of $\tau_M$, that can be scheduled during the execution of one job of $\tau_A$. Hence, we could represent both monitoring and monitored applications by a single periodic task $\tau_A^{(monitoring)} = (C_A^{(monitoring)}, T_A, P_A)$, with $C_A^{(monitoring)} = C_A + B_M^A$. Consequently, using the time overhead definition from methodology (Section IV.5.2) we derive:

$$r_A(monitoring) = \frac{C_A^{(monitoring)}}{C_A} - 1 = \frac{B_M^A}{C_A} \tag{VI.2}$$

For the rest of the chapter, we use the notation $r_M^A = r_A^{(monitoring)}$ to designate the time overhead for monitoring.

### Control-Flow Monitoring Coverage

**Table VI.2:** *Matrix of monitoring coverage issues*

| $n_{ignored}^{(j)}$ | $b_M^{(j)}$ | **Coverage issue(s)** |
|---:|---:|---|
| 0 | 0 | none |
| 0 | $> 0$ | |
| $> 0$ | 0 | minimize $w_{uncov}$ |
| $> 0$ | $> 0$ | minimize $w_{uncov}$, maximize $r_{cov}^A$ |

For a given time budget $B_M^A$, different monitoring strategies can apply, depending on the characteristics of the monitored task $\tau_A$:

- **Maximizing runtime monitoring coverage** ($r_{cov}^A$) for a period of execution $T_A$, the objective is to avoid ignoring $\tau_M$'s job requests, while the budget $B_M^A$ has not been completely used after $\tau_A$'s job has terminated.

- **Minimizing the maximum time window when monitoring is disabled** ($w_{uncov}$) for a period $T_A$. We consider it easier for an attacker to perform a malicious operation in larger unprotected time windows. Hiding an intrusion becomes more complicated when the security protection is non-contiguous: malicious activity should be synchronized to run benign execution in the non-contiguous time slots when monitoring is running.

  To illustrate the point above, let's take the example of a configuration where all monitoring requests are accepted until the budget $B_M^A$ is exhausted. An attacker could typically attempt a Denial of Service attack against the monitoring task $\tau_M$, so that $\tau_M$ is highly involved in a short period after the release time of $\tau_A$'s job and quickly becomes idle. The runtime coverage of the monitored job's CF can be disabled for a maximum time $w_{uncov} = max(0, T_A - N_M^A * T_M)$.

  To adapt the monitoring strategy for the monitored task, we observe two metrics (VI.3) at deadline time of the job $J_{A,j}$, which we use for evaluating coverage issues described in Table VI.2.

$$\begin{cases} n_{ignored}^{(j)}\text{: the amount of ignored jobs of } \tau_M \\ b_M^{(j)}\text{: the remaining budget for } \tau_M \end{cases} \tag{VI.3}$$

### Threat Detection Time

As defined in Table VI.1, $w_{detect}$ is the time for the framework to detect an attack from the moment it was initiated. To be able to set an upper bound for $w_{detect}$, we should set a time constraint on $\tau_M$, so that if no monitoring request has been received during a specified time window, we generate an additional monitoring request. Since we do not introduce such mechanism in this work, we can only provide the lower bound for detecting a threat as the minimum time for accessing CF traces: $w_{detect} \geq T_M + C_M$.

# VI.3 Safety-Aware Control-Flow Monitoring Solution Design

## VI.3.1 Predictable Control-Flow Monitoring

### Solution Overview

Similarly to Hasan et al. [67], we propose to use a server for executing security tasks: we limit the CF monitoring execution time with a periodic server for handling sporadic monitoring job requests of the task $\tau_M$. The server has a predefined time budget to serve monitoring requests, which is consumed when monitoring jobs execute. When the budget is lower than the WCET of monitoring jobs ($C_M$), the server rejects monitoring requests. As described in Section VI.2.4, our goals are, given a static monitoring time budget, firstly to minimize the maximum time window when monitoring is disabled $w_{uncov}$, secondly to maximize CF coverage $r^A_{cov}$.

We introduce the following approach to **limit the maximum time with disabled CF monitoring** $w_{uncov}$. With a fixed time budget $B_w$ so that $C_M \leq B_w < T_A$, the system can accept a maximum of $N_w$ monitoring requests, with $N_w = \lfloor \frac{B_w}{C_M} \rfloor$. We define the monitoring time window $T_w = \frac{T_A}{N_w}$. Hence, without prior knowledge of the monitored execution, if we allocate a budget $B_w$ for monitoring jobs during a period $T_A$ of the monitored task, the lower bound of the maximum uncovered time window is: $min(w_{uncov}) = T_w - C_M$. This means that for each monitoring time window $T_w$, we can accept at least one of the received monitoring requests; We split $T_A$ in $N_w$ periods $\{T_{w,1}, T_{w,2}, ..., T_{w,N_w}\}$ of time $T_w$, which we call monitoring windows, and guarantee a minimum budget to handle one request in each. To improve CF coverage ($r^A_{cov}$), the solution can accept more than one monitoring request in a monitoring window: $\forall i \in \mathbb{N}^*$ and $i < N_w, \forall j \in \mathbb{N}$ and $i < j \leq N_w$, if no request is received during a monitoring window $T_{w,i}$, the budget $B_{w,i}$ can be used in future monitoring windows $T_{w,j}$ included in the period $T_A$ of the job being currently monitored. We allocate the budget $B_{w,1} = C_M$ to the first monitoring window $T_{w,1}$; then, $\forall i \in \mathbb{N}^*$ and $i < N_w$, if the budget $B_{w,i}$ is not consumed in the $i^{th}$ period $T_{w,i}$, it is passed to the next period $T_{w,i+1}$, so that $B_{w,i+1} = C_M + B_{w,i}$.

We then adapt our solution to **increase CF coverage** $r^A_{cov}$: we set an additional time budget $B_r$ for handling irregular monitoring requests. Our objective is to avoid the situation when monitoring requests are rejected in the beginning of the period $T_A$, while the server has not entirely consumed its time budget at the end of $T_A$ (Table VI.2). Thus, we propose to accept the $N_r$ first monitoring requests in the period $T_A$ of the monitored task (with $N_r = \lfloor \frac{B_r}{C_M} \rfloor$), For a predefined time budget $B_M^A = B_r + B_w$ the ratio between $B_r$ and $B_w$ must be adapted in function of the monitored task execution.

### System Model

We substitute the sporadic monitoring task $\tau_M$ by server tasks, which accept or reject monitoring requests following the approach given in the previous section. Our system is now composed of the following set of periodic tasks $\{\tau_A, \tau_{S1}, \tau_{S2}, ..., \tau_{SN_M^A}\}$, where $\tau_A$ is the monitored task, and $\{\tau_{S1}, \tau_{S2}, ..., \tau_{SN_M^A}\}$ the set of server tasks, with $B_M^A$ the total monitoring time budget for one period $T_A$ so that $B_M^A = N_M^A * C_M$. All server tasks have same parameters: $\forall i \in \mathbb{N}^*$ and $i \leq N_M^A, \tau_{Si} = (C_{Si}, T_{Si}, P_{Si})$, so that $C_{Si} = C_M$, $T_{Si} = T_S$ and $P_{Si} = P_S$. Each server runs with the same period as the monitored task ($T_S = T_A$), can handle on monitoring request ($C_S = C_M$), and is able to preempt $\tau_A$, ($P_S > P_A$).

We divide the set of server tasks in two subsets: $\{\tau_{S1}, ..., \tau_{SN_r}\}$ the set of tasks for improving $r_{cov}^A$ and $\{\tau_{SN_r+1}, ..., \tau_{SN_r+N_w}\}$ the set of tasks for reducing $w_{uncov}$, as described in Definition (VI.4).

$$\begin{aligned}
&\forall i \in \mathbb{N}^*, \forall j \in \mathbb{N} : \\
&\quad 1 \leq i \leq N_r \Rightarrow \begin{cases} a_{Si,j} = a_{A,j} \\ d_{Si,j} = d_{A,j} = a_{A,j} + T_A \end{cases} \\
&\quad 1 \leq i - N_r \leq N_w \Rightarrow \begin{cases} a_{Si,j} = a_{A,j} + (i - N_r - 1) * T_w \\ d_{Si,j} = d_{A,j} - C_M * (N_w - (i - N_r)) \end{cases}
\end{aligned} \tag{VI.4}$$

### Server Implementation

By construction, it is not possible to receive a monitoring request while an other request is currently handled: at any time $t$, only one server task $\tau_{Si}$ can be active. Thus, we define the periodic server $\tau_S = (C_S, Q_r, Q_w, T_S)$, as the set of monitoring periodic tasks $\tau_{Si}, \forall i \in \mathbb{N}^*$ and $i \leq N_M^A$, where $Q_r = N_r * C_S, Q_w = N_w * C_S, B_M^A = Q_r + Q_w$. Figure VI.4 provides a schedule example involving a periodic task monitored by such server: the $5^{th}$ monitoring request is ignored to allow the monitored task to meet its deadline, traces generated between $t = 16$ and $t = 17$ are lost.

We introduce $b_r(t)$ and $b_w(t)$ the budget values at time $t$ of execution for the task $\tau_S$. At each new release time $t$ of $\tau_S$, we reinitialize budget values. $b_r(t) = Q_r$, and $b_w(t) = C_S$. Then, for each monitoring window $T_w = \frac{T_S}{N_w}$ in the monitoring period $T_S$, the budget $B_w$ is incremented by $C_S$, The budget $b_w(t)$ is cumulated during a job's execution: if no monitoring request is received, it can reach the maximum value in the last monitoring window: $\forall t, max(b_w(t)) = N_w * C_S = Q_w$. While the replenishment strategy of $b_r(t)$ corresponds to the Deferrable server algorithm [147, 97] (with $\tau_{DS} = (Q_{DS} = Q_r, T_{DS} = T_S)$), the one of $b_w(t)$ budget follows the priority exchange server approach [145] (with $\tau_{PE} = (Q_{PE} = C_S, T_{PE} = T_w)$). Even though our method behaves like a PE server during a monitoring period $T_S$, time budget cannot be cumulated through consecutive periods $T_S$ (i.e. $b_w(t)$ is periodically reinitialized).

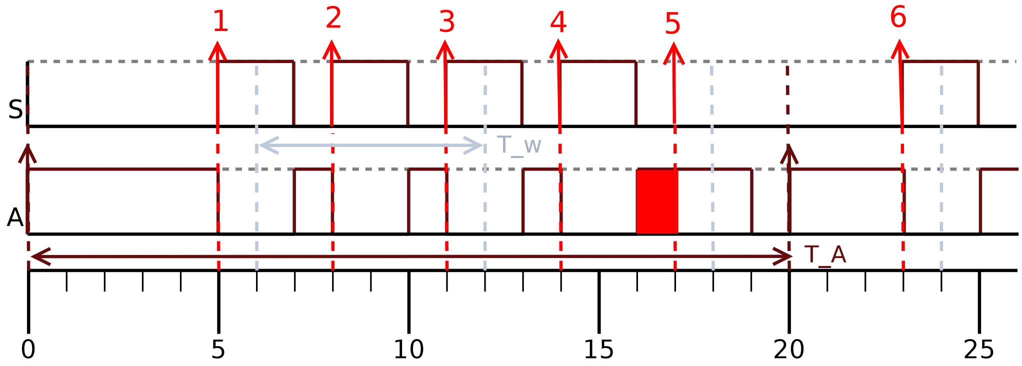


**Figure VI.4:** *A system schedule example, with* $\tau_A = (C_A, T_A), \tau_M = (C_M, T_M), \tau_S = (C_M, C_M, \lceil \frac{T_A}{T_w} \rceil * C_M, T_A)$, *and defining* $C_M = 2, T_M = 3, C_A = 12, T_A = 20, T_w = 6$

### System Schedulability

We first consider the system composed of the periodic server $\tau_S$ and monitored $\tau_A$ tasks. Since monitoring jobs run only by preempting the monitored task $\tau_A$ (Property VI.1), we can represent the server execution time as an extension of the monitored task's execution time: we define the periodic task under monitoring as $\tau'_A = (C'_A, T'_A, P'_A)$, with $C'_A = C_A + B^A_M$, $P'_A = P_A$, and $T'_A = T_A$. In this context, the system can be modeled as a unique periodic task: the system is schedulable if $\frac{C'_A}{T'_A} = \frac{C_A + B^A_M}{T_A} \leq 1$.

We then generalize the system, adding a set of $N$ periodic non-monitored tasks. The task set can now be modeled as $\{\tau'_A, \tau_1, \tau_2, ..., \tau_N\}$. The criticality level of the $N$ non-monitored tasks is independent of the criticality of the monitored task; i.e. $\forall i \in \mathbb{N}, 1 \leq i \leq N$, the priority $P_i$ can be higher or smaller than $P_A$. As an example, applying RM scheduling policy induces that the system is schedulable if [102]:

$$\frac{C_A + B^A_M}{T_A} + \sum_{k=1}^{N} \frac{C_k}{T_k} \leq U_{lub} = (N+1) * (2^{\frac{1}{N+1}} - 1) \qquad \text{(VI.5)}$$

As $P_S > P_A$, to avoid breaking timing constraints of tasks with higher priority compared to $\tau_A$, $P_S$ must be configured so that: $\forall \tau = (C, T, P)$, $\tau \notin \{\tau_A, \tau_S\}, P_A < P \Rightarrow P_A < P_S < P$. Hence, we can apply CF monitoring on a low-critical task without compromising the schedulability of the system.

## VI.3.2 Application-Profiling Based Anomaly Detection

### Motivation

We intend to use the framework to limit the performance impact for monitoring, without considering the worst-case monitoring overhead in the system design. For such use-case, we cannot guarantee a full coverage of the monitored CF at runtime. Even though we cannot collect all CF traces, we are still able to monitor the behavior of the CF through the observation of inter-arrival times of monitoring (trace collection) requests. With this metric, we can observe how many basic-blocks the monitored application executes for a fixed time period. Therefore, our framework combines two levels of observation of the monitored CF: at basic-block level with CF trace collection and at higher level observing the rate of monitoring request inter-arrival times. While trace collection is discontinuous, high-level CF observation applies on the full execution of the monitored task.

#### Anomalous Trace Generation

CF and Data-Flow based attacks involve the use of gadgets chains, which are sequences of short basic-blocks. Because it is likely to execute shorter basic-blocks compared to a benign program, the execution of such malicious program should cause a higher amount of monitoring traces over a given time period; i.e. a shorter inter-arrival times of monitoring requests. Therefore, we expect to detect malicious activity at runtime from the observation of anomalous inter-arrival times of monitoring jobs.

#### Application Profile Generation

We propose to build an application-profiling based ADS, which counts the number of monitoring requests received during a fixed period of time. We select an ADS observation window corresponding to the monitoring window $T_w$, although other values could apply, because our goal is to detect attacks attempting to hide malicious activity in unmonitored time windows $w_{uncov}$. For each time monitoring window $T_w$ of the server within the period of a monitored job ($\tau_A$), we monitor $n_{ignored}^{(T_w)}$, $n_r^{(T_w)}$, and $n_w^{(T_w)}$ the counts of monitoring requests respectively rejected, accepted by the server using $Q_r$ budget, accepted with the budget $Q_w$. For each monitoring requests count signal, we define a range of authorized values (from the minimum to the maximum), after the observation of the monitored task execution under normal conditions. The set of window data defines the application profile to be used. After each monitoring window of execution, we compare signal values with the application profile: whenever one value is out of the predefined range of authorized values, the ADS identifies the monitored application as anomalous.

### VI.3.3 Monitoring Framework Implementation

#### Predictable Monitoring Server Integration

We integrate the periodic server solution to the hardware-assisted tracing framework described in Section VI.1 (Figure VI.3), by introducing two additional routines in the monitor application: the synchronizer and the ADS service. The synchronizer thread updates $b_r(t)$ and $b_w(t)$ budget values in function of the time of execution in the system (Section VI.3.1). The ADS service specified in Section VI.3.2 runs sequentially after the synchronizer (i.e. after monitoring windows $T_w$) on the monitoring CPU core 1, so that it does not cause temporal interference with the monitored task.

The server accepts a monitoring request for trace collection at time $t$, if at least one of the remaining budget times $b_w(t), b_r(t)$ is higher than $C_M$. It then increments the corresponding counter of requests in the current anomaly detection window $T_w$ (either $n_{ignored}^{(T_w)}$, $n_r^{(T_w)}$, and $n_w^{(T_w)}$). After measuring the time spent in the trace collector thread, the server subtracts it from remaining budgets $b_w(t), b_r(t)$. If the trace collector has not returned before a new job activation of $\tau_A$, only the time consumed after the new job activation is considered for updating server's budget.

### Server Configuration Process

$C_M$, $T_A$, and $B_M^A$ server parameters must be known before starting the configuration process. The objective is first to define the ratio $r_{wr} = \frac{Q_w}{B_M^A}$ to assure a workable trade-off between monitoring coverage and detection time and accuracy, second to define the application profile for the ADS. We divide the configuration process in 2 phases: training and validation. We start the training, by determining the ratio $r_{wr}$, so that the maximum budget $B_M^A$ is consumed for CF monitoring. We initialize the server with $r_{wr} = 100\%$ (corresponding to the shortest monitoring window for fine-grained monitoring). We run a set of executions of $\tau_A$ monitored with this server setup. After each execution, if the coverage ratio $r_{cov}^A$ does not reach its maximum value (i.e. the budget time is not totally used) and if the monitoring window does not reach the maximum value $T_w = T_A$, we update monitoring budgets of the server as follows: $Q_r = Q_r^{(old)} + C_M$ and $Q_w = Q_w^{(old)} - C_M$. Once the ratio $r_{wr}$ is known, we generate the application profile for the ADS as explained in Section VI.3.2, using a set of system execution samples. In the validation phase, we test the application profile on additional normal traces of execution: if the ADS detects anomalies (i.e. false-positives), we go back to the training step, setting the ratio $r_{wr}$ manually; our approach consists of decreasing the ratio $r_{wr}$, because we expect the observation of monitoring requests to be more stable for longer monitoring time windows.

### System Deployment Process

We propose the following method to deploy our monitoring framework.

1 We derive WCET values $C_A$ and $C_M$ at design time from a predefined amount of repeated measurements.

2 We define a maximum time overhead ratio $r_M^A$ for monitoring, so that $r_M^A = \frac{B_M^A}{C_A}$. $r_M^A$ value depends on the trade-off between the performance impact limitation and the CF monitoring coverage at runtime.

3 Knowing $r_M^A$ and $C_A$ the worst-case execution time of the monitored application, we can derive the period $T_A$ as $T_A = C_A * (1 + r_M^A) = C_A + B_M^A$.

4 We build the application profile for the ADS, as described in Section VI.3.3, using a set of execution samples. After determining the ratio between $Q_r$ and $Q_w$ so that $B_M^A = Q_w + Q_r$, we configure the final server as $\tau_S = (C_M, Q_r, Q_w, T_A)$.

5 We finally deploy the framework with the server configuration and application profile defined in previous step.

The security evaluation in Section VI.6 provides an example of system deployment.

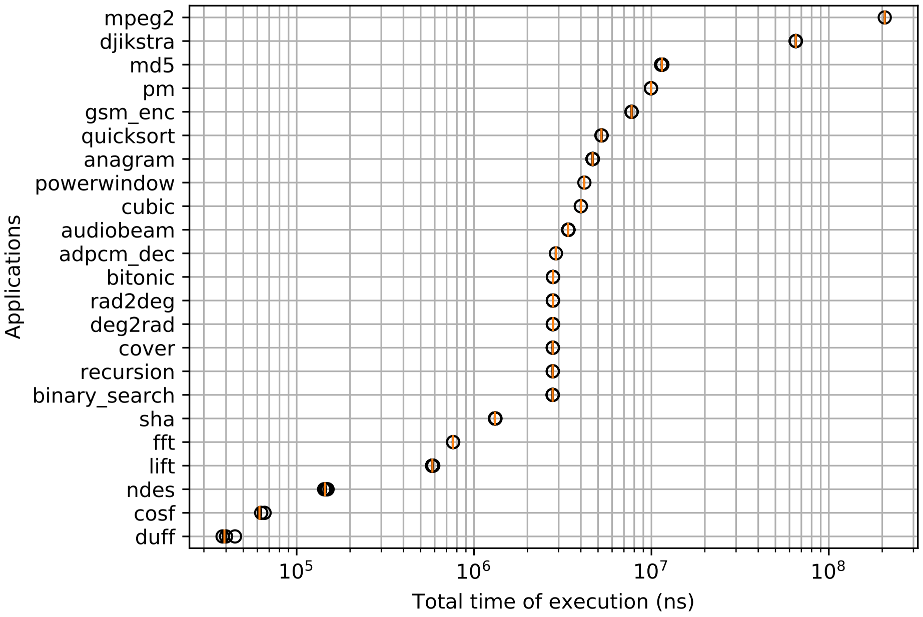


**Figure VI.5:** *Execution time for TACLeBench tested applications (no monitoring)*

#### Evaluation Outline

After defining our evaluation setup in Section VI.4, we study the performance overhead for full runtime CF coverage with trace collection in Section VI.5. In Section VI.6, we constrain our framework to a maximum $r_M^A = 10\%$ of time overhead for trace collection, to then evaluate our ADS service leveraging security metrics from Table VI.1.

## VI.4 Experimental Setup

### VI.4.1 Application Test Set

We define a set of test programs to represent jobs executed by the monitored task $\tau_A$. For our analysis, we consider that $\tau_A$ periodically runs the same application. We reproduce every measurement 100 times to perform static analysis. We evaluate our framework on a panel of single-threaded 23 applications from TACLeBench benchmark [56]. We select these applications since they are self contained (no library or system calls) and they represent typical multimedia applications focusing on computational operations: e.g. audio beam former (`audiobeam`), md5 hash function, MPEG2 motion estimation, and simulations of real embedded embedded controller applications (`lift` and `powerwindow`). The execution time varies in the set of tested applications from $40\mu s$ for `duff` to $200ms$ with `mpeg2` (see Figure VI.5).

In addition to testing representative programs, our goal is to determine the worst-case monitoring situation; i.e. the worst-case monitored application. In the context of CF monitoring, a worst-case program is a program that generates the highest amount of traces. Because the execution time of a monitoring job (trace collection) is constant

**Algorithm 2:** Pseudocode for infbr, inffcall, and infrec programs

```
Function infbr()
    while true do
        /* nothing: infinitie looping */
    return;
Function f()
    return 1;
Function infcall()
    while true do
        f();
    return;
Function rec(int $i$)
    if $i \leq 0$ then
        return 0;
    return rec($i - 1$);
Function infcall()
    while true do
        rec(1000);
    return;
```

for a given buffer size, the worst-case corresponds to the minimum inter-arrival time of CoreSight hardware interrupts (monitoring requests).It is difficult to provide such a worst-case because of non-trivial factors in CoreSight trace formatting process [14]. In work [92], we introduced `infbr` as the worst-case program. However, we observe a higher overhead for some of TACLeBench applications (performance evaluation in Section VI.5.1). Hence, our evaluation considers three pessimistic applications for CF monitoring: `infbr`, `inffcall`, and `infrec`, described in Algorithm 2. These programs generate high amounts of branches (function call, return, direct branch, and conditional branch) inside non-ending loops. The worst-case monitored program for this work corresponds to `inffcall` (Section VI.5.1).

### VI.4.2 Hardware Environment

We use the ARM Juno development platform [77] to implement and test our framework on the cluster of 4 Cortex-A53 CPU cores. According to ARM CoreSight documentation [15], the size of CoreSight trace buffer can be configured from $4B$ to $64KB$. The time for collecting the traces depends on the size of the buffer: the bigger the buffer, the more traces to copy to main memory. Hence, the WCET of the monitor task $C_M$ depends on the buffer size.

### VI.4.3 Embedded Trace Buffer Size Configuration

To detect security threats at runtime, one key factor is responsiveness: the faster the threat is detected, the quicker the system can initiate defense actions (e.g. application abort). In the context of security monitoring based on ARM CoreSight tracing, the responsiveness depends on the time for the data to be accessed by the security monitoring service. Traces can be accessed, after the hardware interrupt is raised when the ETB buffer is full. The shorter the ETB buffer, the faster it is filled. Hence, the shorter the buffer, the better the monitoring responsiveness.

To achieve fine-grained monitoring, we determine in the remainder of this section a suitable buffer size configuration to run the evaluation of the framework (Sections VI.5 and VI.6). Because CoreSight hardware enables trace compression in our setup, we investigate the impact of the size on the ratio between synchronization data and usable CF traces, to then show that even though the trace collection only depends on the buffer size, it is indirectly affected by the monitored program.

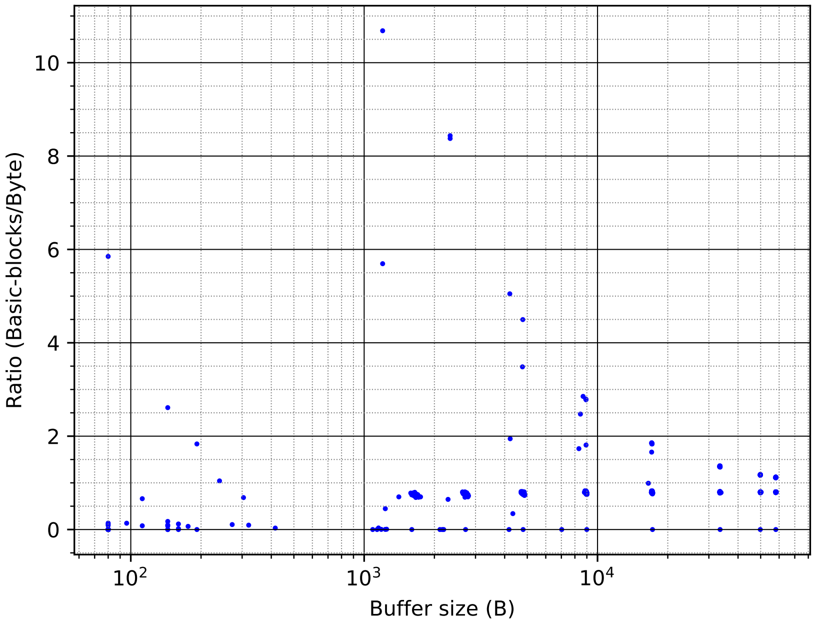


**Figure VI.6:** *Trace buffer usability for different buffer sizes*

#### Trace Buffer Usability

To reduce memory storage, CoreSight hardware enables trace compression. The trace buffer contains actual traces of the monitored core execution, as well as synchronization data. Usable traces include basic-blocks executed by the program as well as trapped exceptions on the monitored core. The scheduler timer is the only exception type trapped in our setup.

Figure VI.6 shows the ratio of basic-blocks per byte of buffer storage, monitoring `powerwindow` program. We observe a decreasing average value and higher variations in measurements when the configured buffer size decreases. We also note a higher amount of null samples for buffer sizes below $1KB$. **Hence, we propose to set the buffer size to $2KB$ for our framework.**

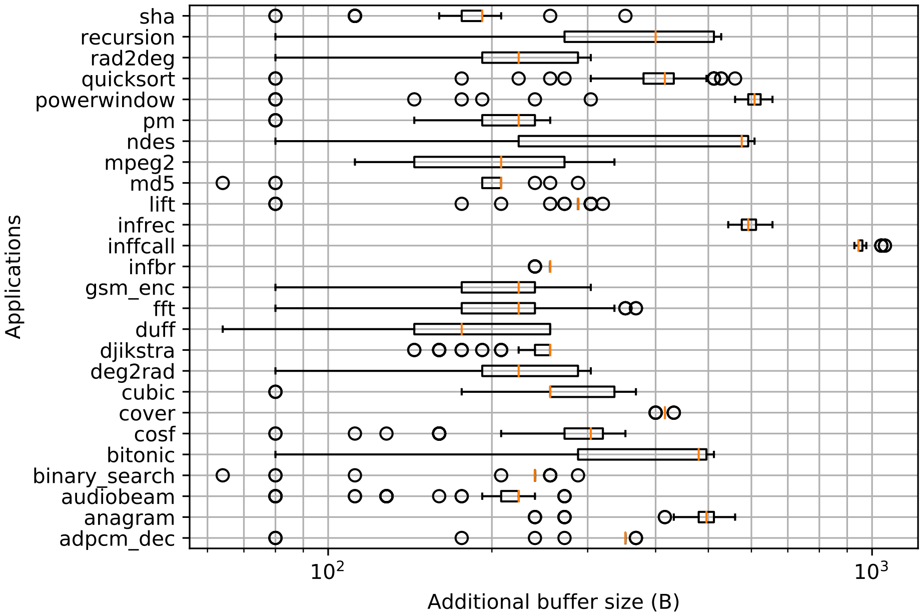


**Figure VI.7:** *Extra buffer size distribution for all applications ($2KB$ buffer)*

### Extra-Buffer Size

After the monitoring request (i.e. CoreSight hardware interrupt) is generated and until it is handled by the monitor for copying the traces back to memory, the monitored program continues to execute (Figure VI.3); during that time, CoreSight hardware possibly stores additional traces in the buffer. The quantity of additional traces depends on the monitored application execution (Figure VI.7): For a buffer of $2KB$ size, our measurements are between $64B$ and $1936B$, and they remain in the great majority below $300B$. We also observe that the buffer size configuration does not significantly impact this quantity as illustrated with `inffcall` program on Figure VI.8.

### Final Buffer Configuration

The trace collection designates the time phase, when the monitored application is preempted by the secure monitor for serving an arriving monitoring request. This operation consists of copying the traces stored in the ETB buffer into main memory. We designate $t_{copy}$ the time for trace collection. The maximum time $t_{copy}$ corresponds to $C_M$ the WCET time for executing trace collection.

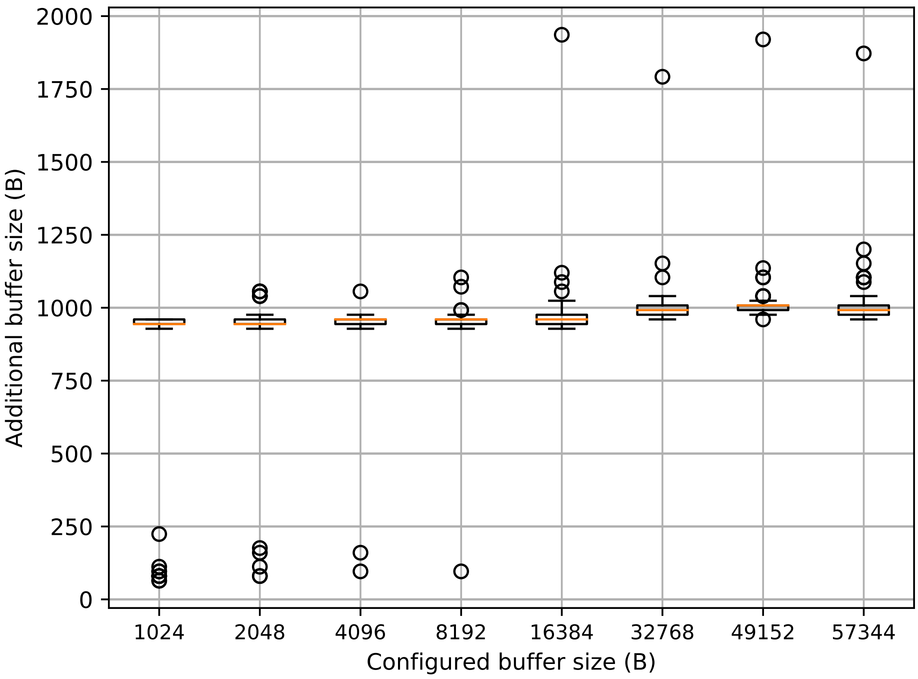


**Figure VI.8:** *Extra buffer size distribution for inffcall application*

The time for collecting traces depends on the quantity of data to copy: $t_{copy}$ depends on the buffer size. By construction, since monitoring jobs are triggered when the ETB buffer is full, $t_{copy}$ value is constant for a fixed buffer size. Figure VI.9 shows indeed a linear relation between the buffer size and the trace collection time for all tested programs and buffer size configurations: $t_{copy} = \alpha_{copy} * s$, with $s$ the actual buffer size and $\alpha_{copy}$ a constant. With buffer sizes between $1KB$ and $63KB$, $\alpha_{copy}$ is comprised between $38ns/B$ and $39ns/B$ for 75% of measurements. More specifically with a buffer size configuration of $2KB$, we note a maximum ratio of $\alpha_{copy} = 44ns/B$.

Excepting pessimistic programs which we consider as anomalies, we measure extra-buffer sizes below $700B$. We can then determine the WCET value $C_M$ for trace collection with our configuration, using the worst-case actual buffer size ($2748B$) and worst-case coefficient ($\alpha_{copy} = 44ns/B$): $C_M = 44 * (2048 + 700) = 121\mu s$.

## VI.5 Control-Flow Monitoring Performance Overhead

The objective in this section is to evaluate the performance overhead for our monitoring framework, following the approach described in Section IV.5.2:

- We evaluate the maximum time overhead $r_M^A$ for the framework, on a set of representative and pessimistic applications. We suppose that the time budget $B_M^A$ is long enough, for the server task $\tau_S$ to accept all arriving monitoring requests.
- We also analyze additional infrastructure costs for integrating the monitoring module: i.e. specific hardware requirements and memory overhead.

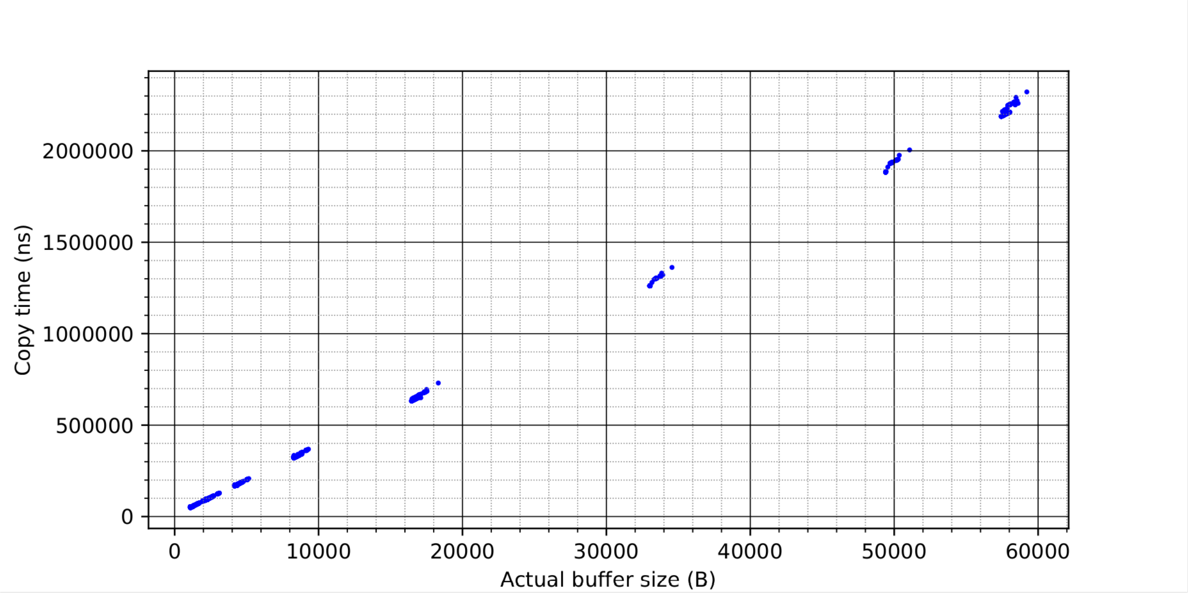


**Figure VI.9:** *Copy time measurements for different buffer sizes and all tested applications*

## VI.5.1 Time Overhead of the Control-Flow Monitoring Framework

In this section, we evaluate the monitoring time overhead for covering the full runtime control-flow of the monitored task (i.e. $r^A_{cov} = 1.0$). We identify three sources of time overhead:

- trace collection ($T_{overhead}$): the delay in the monitored task $\tau_A$'s execution due to the trace collection software framework ($T^A_{copy}$) and ARM CoreSight time overhead to activate hardware CF tracing ($T_{overhead\_internal}$); i.e. $T_{overhead} = T^A_{copy} + T_{overhead\_internal}$.
- trace analysis ($T_{analysis}$): time to analyze a full buffer of collected CF traces; i.e. performing CFG path reconstruction and CFI checking.

### Trace Collection Time Overhead

For this experiment, we measure the time $T^A_{copy}$, which is the total time for trace collection during the execution of the monitored application; i.e. the execution time of the task $\tau_S$, during the execution of the task $\tau_A$). To simplify the evaluation, we set $B^A_M = T^A_{copy}$, to achieve a full coverage of the monitored execution. We then compute $r^A_M = \frac{B^A_M}{C_A}$, the response time slowdown of task $\tau_A$ for the applications test set.

According to Figure VI.10, with a buffer size configured to $2KB$, the ratio $r^A_M$ varies for all programs from 0% execution time to 140% of the execution time. Many of the tested applications' overhead is above 40% overhead. For applications with 0% overhead, the rate of control-flow transitions is too low during the execution to fill CoreSight buffer.

We observe the general worst-case performance overhead for the application `inffcall`: in particular for a buffer size of $2KB$, monitoring requests arrive at an average rate of

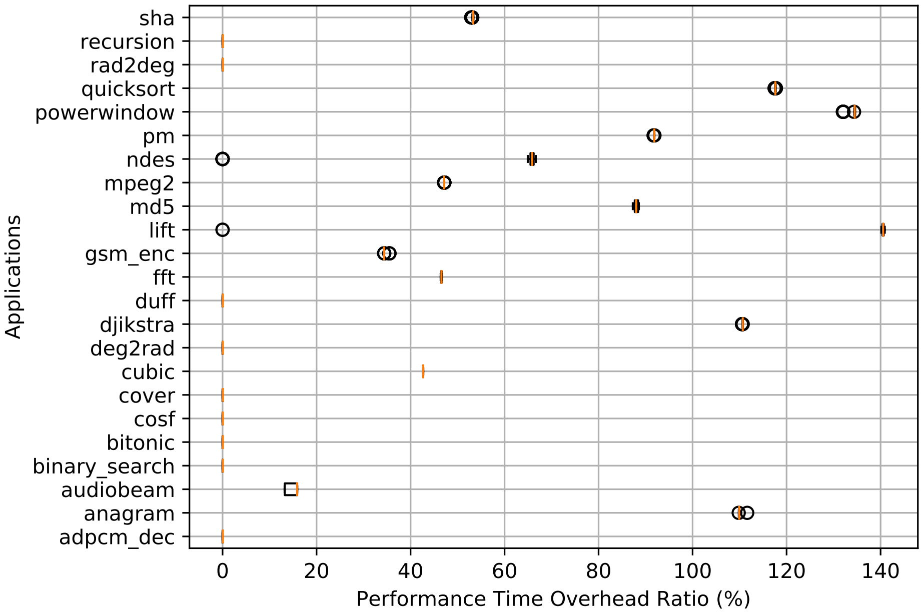


**Figure VI.10:** *Time overhead for tested applications, with trace copy*

$21\mu s$ (with $1\mu s$ standard deviation). Since the program is composed of a very short routine in an infinite loop, we consider constant trace collection copy time ($C_M = 121\mu s$) and inter-arrival time ($T_M = 21\mu s$). $B_M^A = N_M^A * C_M = \frac{C_A}{T_M} * C_M$. Hence, $r_M^A = \frac{C_M}{T_M}$. From our measurements, the worst-case corresponds to $r_M^A = 605\%$ in average, with 53% of standard deviation. Similarly, we compute 128% and 367% average overhead ratios respectively for `infbr` and `infrec` programs (with standard deviations below 18%).

### CoreSight Internal Time Overhead

The second overhead $T_{overhead\_internal}$ corresponds to the execution time overhead of the monitored job, when the server rejects all monitoring requests (without trace collection). $T_{overhead\_internal}$ originates from CoreSight hardware implementation: $\tau_A$ is never preempted by $\tau_S$, and there is no kernel overhead on the monitored core for handling CoreSight interrupts, since these are routed by the hardware interrupt controller to a different CPU core (core 1 running the server thread on Figure VI.3).

We systematically observe a slightly longer execution time for the application when the tracing framework is up. The overhead is among all tested applications and buffer sizes between $500ns$ and $4\mu s$. Probably because of the high variability in time measurements, the results do not reveal a correlation between $T_{overhead\_internal}$ and the total execution time of the monitored application (Figure VI.11).

### Time Overhead for CFG Path Reconstruction and CFI Checking

As mentioned in the beginning of this section, we consider an additional task for analyzing the collected traces. The trace analysis can be decoupled from trace collection and execute in parallel to the monitored application. We define the corresponding time overhead for analyzing a full buffer of traces $T_{analysis}$. We decompose the trace analysis into two subtasks, following the description in Section VI.1.2: for reconstructing the CFG path of the traced application and CFI checking. We define $T_{cfg}$ and $T_{cfi}$ as the execution time respectively to perform CFG path reconstruction and CFI checking. Hence, since CFI checking can only apply on reconstructed CFG path, both tasks are sequential; i.e. $T_{analysis} = T_{cfg} + T_{cfi}$.

After the monitor copies the batch of traces from the ETB to system memory, these two task are invoked by the SK. In the initial paper [92], we measured the time taken by the tasks to process 64 KiB of trace data to be approx. $T_{analysis} = 1s$ (between $[0.92s, 1.10s]$ with 99% confidence). In our observations, 95% of the overhead is contributed by the CFG path reconstruction ($T_{cfg}$), while the CFI checking only requires 5% ($T_{cfi}$). CFI checking takes approx. between $[1.3ms, 1.5ms]$ for forward edges and $[51.7ms, 61.6ms]$ for backward edges, both with a 99% confidence. This huge overhead for the CFG path reconstruction directly stems from the use of non-optimized libraries and the overhead in the Linux VM that hosts these two tasks. In Section VI.7.2, we describe design strategies to reduce this overhead for a real-world deployment.

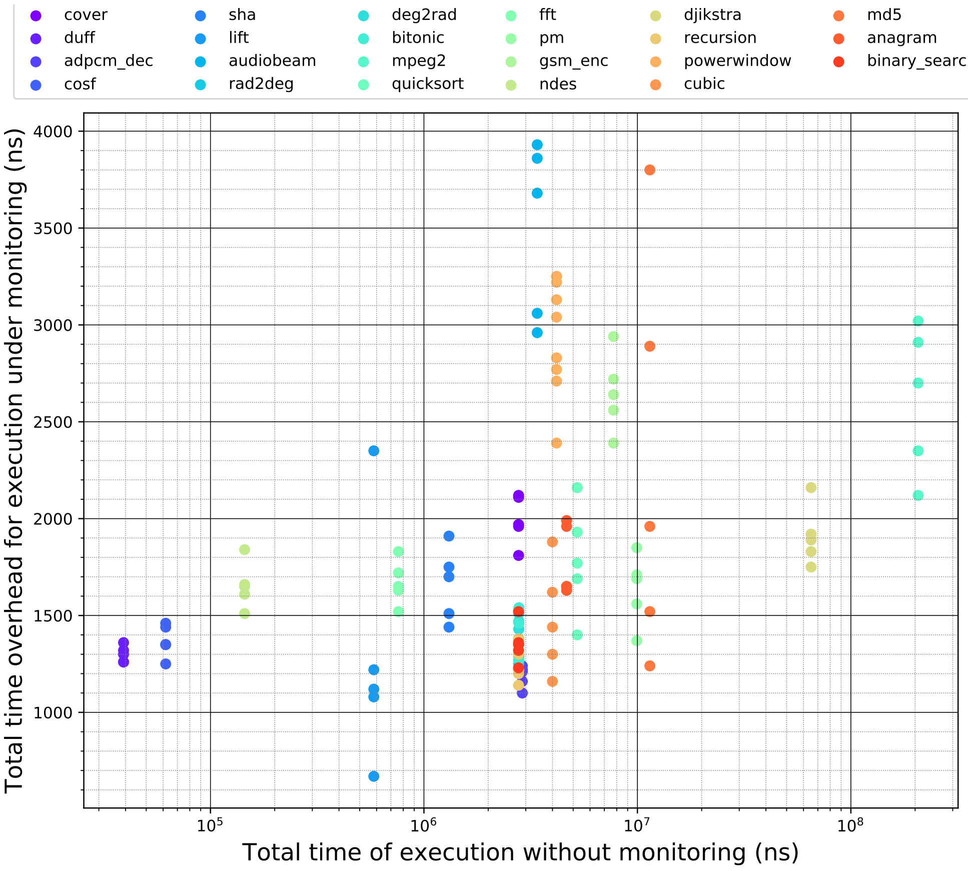


**Figure VI.11:** *Time overhead for tested applications, without trace copy*

### VI.5.2 Additional Hardware Costs for Integrating the Framework

Our framework requires additional hardware to perform monitoring. Of course, ARM CoreSight – or an equivalent processor tracing feature for CPU architectures other than ARM – is a prerequisite for the hardware platform to support hardware-assisted control-flow monitoring deployment. As disclosed in Section VI.1.1 with the system architecture, the trace collection and trace analysis applications require a dedicated set of CPU cores to limit potential interference with the monitored application; in our experiment, we use one CPU core for each application (i.e. two additional cores for monitoring).

In addition to the time overhead, the framework comes with a memory overhead:

- static memory overhead: in the system program binary, corresponding to the two partitions used for trace collection and trace analysis.

- dynamic memory overhead: at runtime with a shared buffer in RAM, to store generated control-flow traces before they are consumed by the trace analyzer.

## VI.6 Framework Security Evaluation

### VI.6.1 Objective

From Figure VI.10, we observe that the time overhead is generally greater than 40% for a buffer of $2KB$. Therefore, in the following, we propose to limit the time overhead to $r_M^A = \frac{B_M^A}{C_A} = 10\%$. We then set the following objectives, following the intrusion detection efficiency evaluation described in methodology (Section IV.5):

- achieve a workable trade-off between CF coverage and detection time. For this, we evaluate the monitoring scope and the monitoring continuity using the metrics described in Table VI.1:
  - $r_{cov}^A$: the coverage of the monitored CF.
  - $w_{uncov}$: the maximum time when CF trace collection is disabled.
  - $w_{detect}$: the maximum time for the ADS to detect an anomalous CF execution.

- evaluate the detection accuracy of the ADS service for application profiling, as well as its ability to detect unknown anomalies (i.e. potential zero-day attacks).

Figure VI.12 shows the distribution of inter-arrival times of monitoring requests for the complete test set. Almost all distributions include common outliers corresponding to $700\mu s, 70ms, 300ms$: these values correspond to initialization and termination of programs, when messages are printed on the console. However, since distribution characteristics vary much across programs, we can expect monitoring inter-arrival time signal to be a relevant input for application profiling.

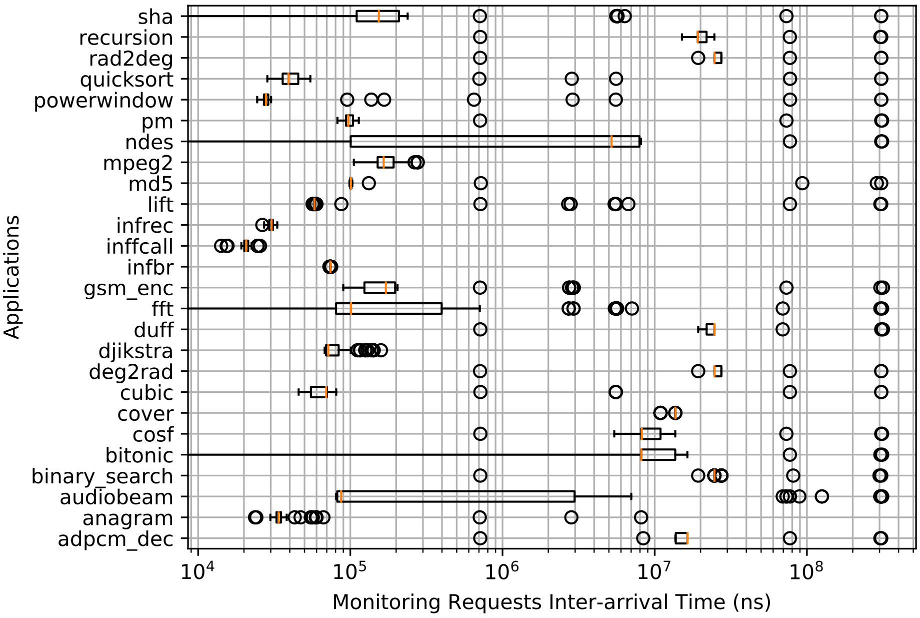


**Figure VI.12:** *Distribution of monitoring requests inter-arrival times for all applications (without trace copy)*

## VI.6.2 Evaluation Setup

We base our security evaluation on the setup described in Section VI.4. For anomaly detection, we need to define an application profile, so that the ADS can identify anomalies as deviations from the profile reference during runtime. To build a profile in our evaluation context, we do not consider applications which generate no monitoring requests (i.e. 0% overhead on Figure VI.11). In our test set, 9 applications correspond to the same 'Null' profile: any monitoring request received by the server is an anomaly, resulting in detecting all applications with a different profile. Programs whose execution time is shorter than $C_M$ are also out of scope, since handling one monitoring request could break the 10% time overhead limit. We also do not consider profiles for the 3 pessimistic programs. Because these are rather simple (low variation in short monitoring inter-arrival times), their profile should be easy to monitor: under an appropriate threshold of received monitoring requests count, the execution is anomalous. In total, we run the evaluation on a profile set of 11 applications. For testing the profiles, we consider our 26 applications (Section VI.4.1), but we use only one of the programs corresponding to the 'Null' profile, as they are all equivalent: hence, our test set is composed of 18 (1 normal, 17 anomalous) different applications.

## VI.6.3 Anomaly Detection Evaluation

### Overview

We evaluate the ADS service following three steps for each application in the profile set: training and validation (detailed in Section VI.3.3), and test. We first build the application profile of the monitored task $\tau_A$, using 70 samples of execution. We then validate the model on 20 samples. No anomalous sample is used during these two first steps. Finally, we evaluate the model on 10 samples of each application from the test set; i.e. 1 normal and 17 anomalous applications. Training, validation, and test sets are composed of unique samples (i.e they do not overlap).

### Training and Validation

We follow the steps described in Section VI.3.3, to configure the system before runtime:

1 We set the WCET for trace collection $C_M = 121\mu s$ (Section VI.4.3). We then select the program to build the reference profile. We measure its maximum execution time $c_{max}$ under monitoring (i.e. $c_{max}$ includes $T_{overhead\_internal}$) across all training samples. We set $C_A = \alpha * c_{max}$, with the coefficient $\alpha = 1.1$ determined empirically, so that test execution times remain lower than $C_A$.

2-3 According to the defined objective in Section VI.6.1, we set $r_M^A = 10\%$ and compute $T_A = C_A * (1 + r_M^A) = 1.1 * C_A$.

4 With $B_M^A = r_M^A * C_A = 0.1 * C_A$, we train the server as explained in Section VI.3.3 to build the application profile and the server configuration. After the profile is generated, we assign corresponding $Q_w$ and $Q_r$ to the server, so that $\tau_S = (C_M, Q_r, Q_w, T_A)$.

5 The framework is ready for the test phase.

### Test

We run each application in the test set with the framework defined in the two previous steps. During runtime, after each monitoring window $T_w$, the ADS compares monitoring requests characteristics from the past window with the application profile. If one of the signals is out of the reference range of values, the ADS detects an anomalous execution.

## VI.6.4 Results

### Detection Accuracy

Table VI.3 shows test results. For each application profile, we observe an accurate detection of execution samples. Our ADS is able to detect anomalous executions; it is potentially able to detect zero-day attacks. We use standard metrics to evaluate the detection accuracy, both in regard of false-positive rate and false-negatives: with $FP$ the count of false-positives, $FN$ false-negatives, and $TP$ true-positives, we define precision as $P = \frac{TP}{TP+FP}$, recall as $R = \frac{TP}{TP+FN}$, and F1-score as $\frac{P*R}{P+R}$.

The detection accuracy is maximum for the great majority of tested cases. We observe false-negative samples only when the ADS monitors the `gsm_enc` program with `audiobeam` profile. The distribution of inter-arrival times for `audiobeam` is broader compared to other applications in the profile set, in particular `gsm_enc` (Figure VI.12):

**Table VI.3:** *Security results with* 10% *performance overhead and buffer size* = $2KB$

| Application | $T_A$ $(ms)$ | $r_{wr}$(%) | $N_w$ | $w_{uncov}$ $(ms)$ | $r^A_{cov}$ (%) | $w_{detect}(ms)$ | | Detection Accuracy (%) | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | mean | stdev | precision | recall | F1-score |
| sha | 1.543 | 100.0 | 1 | 1.422 | 12.5 | 1.543 | 0.000 | 100.0 | 100.0 | 100.0 |
| audiobeam | 4.110 | 100.0 | 4 | 0.907 | 62.0 | 2.539 | 1.458 | 97.1 | 100.0 | 98.5 |
| cubic | 4.765 | 100.0 | 4 | 1.070 | 21.1 | 2.515 | 1.184 | 100.0 | 100.0 | 100.0 |
| powerwindow | 5.058 | 80.0 | 4 | 1.143 | 9.3 | 2.669 | 1.257 | 100.0 | 100.0 | 100.0 |
| anagram | 5.586 | 100.0 | 5 | 0.996 | 8.8 | 2.483 | 1.266 | 100.0 | 100.0 | 100.0 |
| quicksort | 6.300 | 83.3 | 5 | 1.139 | 9.4 | 2.660 | 1.252 | 100.0 | 100.0 | 100.0 |
| gsm_enc | 9.326 | 88.9 | 8 | 1.045 | 26.6 | 2.461 | 1.159 | 100.0 | 100.0 | 100.0 |
| pm | 12.028 | 91.7 | 11 | 0.972 | 10.7 | 1.762 | 2.502 | 100.0 | 100.0 | 100.0 |
| md5 | 13.947 | 92.9 | 13 | 0.952 | 11.3 | 1.132 | 0.246 | 100.0 | 100.0 | 100.0 |
| djikstra | 78.750 | 96.2 | 76 | 0.915 | 9.1 | 2.590 | 1.476 | 100.0 | 100.0 | 100.0 |
| mpeg2 | 250.039 | 2.4 | 6 | 41.552 | 22.6 | 41.673 | 0.000 | 100.0 | 100.0 | 100.0 |

the non-detection of `gsm_enc` is caused by the larger range of authorized values in monitoring windows for `audiobeam` profile. With shorter monitoring windows (i.e. more monitoring budget), we could apply more-fine analysis to improve the detection of dissimilarities between both programs.

### Control-Flow Monitoring Coverage

For the 10% boundary of time overhead, the coverage $r^A_{cov}$ varies among monitored applications, depending on the rate of monitoring request inter-arrival times and the ratio $r_{wr} = \frac{Q_w}{B^A_M}$. Table VI.3 shows $r_{wr}$ average values, since the standard deviation is for all configurations below 0.2% except for `audiobeam` with 6.5%. For most of the applications, we assign more than 80% of $B^A_M$ budget to $Q_w$. `mpeg2` represents the only exception with only 2.4%: because the amount of monitoring requests generated per time window $T_w$ varies much across this application's execution samples, shorter $T_w$ values come with an important degradation of the precision at validation step.

### Anomaly Detection Time

As stated in the previous section, `mpeg2` comes with a lower limit to set $T_w$ value. Except for this use-case, the sum of windows $N_w$ increases with the period $T_A$. The ADS detects in average the anomaly after the second monitoring window. The more monitoring windows, the faster the detection: Table VI.3 shows an average detection time close to $2,5ms$ for 7 of the 11 application profiles, which have different execution periods $T_A$.

# VI.7 Discussion

## VI.7.1 Lessons Learned

### Control-Flow Monitoring Performance Impact

We observed a high performance impact for trace collection based monitoring on the set of tested applications: for more than the half of TACLeBench tested applications, the overhead is above 40% and can reach up to 140%. We assume that this high overhead results from the characteristics of tested programs: TACLeBench applications focus on computational operations, causing a high rate of monitoring requests. In contrast, the framework developed by Kuzhiyelil et al. [92] was tested on a network stack application, with a low performance overhead ($< 1\%$). For that reason, we suggest to combine this test set with related programs for data access and control (e.g. reading sensors and writing to actuators and/or peripherals) to reflect a more holistic picture of the practical applicability of the solution proposed in the work at hand. Instead of monitoring low-critical programs, we could indeed apply the framework on high-critical software which control sensors and actuators and would potentially generate a comparatively reduced performance overhead.

In our evaluation, we computed a maximum time overhead $r_M^A$ corresponding to more than 600% slow-down of the monitored task. This scenario being highly pessimistic (less than 150% across TACLeBench tested applications), it is very unlikely to be observed in realistic deployment. Our framework brings flexibility to control the performance impact, assuring predictability without necessary consideration of the worst-case monitoring overhead. The system integrator can indeed adapt the monitoring coverage, in function of the deployment use-case (i.e monitored task, system timing constraints). More specifically for a given monitored task $\tau_A$, since coverage and performance overhead are correlated so that $r_{cov}^A$ cannot increase when the performance overhead $r_M^A$ decreases, the system integrator can adapt the security coverage by limiting the performance overhead.

### Anomaly Detection Based on Control-Flow Monitoring

Our monitoring solution, except if it is configured to support the worst-case overhead, cannot guarantee a full coverage of CF transitions executed by the monitored task during runtime. The security evaluation in Section VI.6 validates the relevance of the ADS periodic service to detect anomalous behaviors. The observation of monitoring inter-arrival times provides indications to build profiles of applications with few detection errors. Even though our goal for fast detection is to configure short observation time windows, we noticed in some case a size configuration threshold, below which the precision of the detection falls.

### Framework Practicability for Industrial Mixed-Criticality Systems

Our monitoring framework can be deployed in mixed-criticality systems with constraints such as hard real-time deadlines. First to assure the isolation of system tasks, we leverage a partitioned architecture with a separation kernel (Section VI.1.1). Second, Section VI.3.1 provides the schedulability guarantee to integrate the solution into a multi-tasks MCS with one monitored task. The system maximum time overhead of the framework is configurable before deployment by the system integrator (Section VI.3.3).

The overall time overhead considerably depends on the monitored program (Figure VI.10). We propose a trade-off between time overhead and security monitoring coverage. On one hand, our framework limits and makes the worst-case overhead predictable, leveraging a partial CF-traces coverage. The system integrator can configure the worst-case overhead, adjusting the monitored task's characteristics and monitoring constraints, as described in Section VI.3.3. On the other hand, we built an ADS service to detect threats in the whole execution (Section VI.3.2).

Our CFI solution is transparent to the monitored application, since it does not require source code or binary instrumentation. For monitoring an application, in addition to the application binary, we only require CFI metadata that is generated during an offline pre-processing phase by the application developer and stored separately from the application. This makes our solution suitable for use in a multi-supplier product development, as practiced in the automotive, railway, and avionics domains.

For the certification of complex safety and security critical systems built with components from multiple suppliers, a compositional certification methodology [141] is typically used, in which the lower-level component developers (e.g., application developers) perform evaluation for their components and provide the evidences to the higher level developers (e.g., system integrator, operators), who then reuse those artifacts to certify the system as a whole. The practice of reuse of low-level evaluation evidence is also reflected in standards such as Common Criteria [143], ISO 26262 [82], and IEC 62443[133]. As our monitoring solution does not require modification at the source code or binary level, when the system integrator wants to enable monitoring of a component, there is no need to regenerate the evaluation evidences for that component, which eases the reuse.

## VI.7.2 Framework Improvements Towards Industrial Deployment

### Implementation Alternatives

Our evaluation results show a significant performance overhead for trace collection on one hand, a good detection accuracy for the ADS service on the other hand. Hence, we could consider monitoring runtime CF only with the ADS, without CF trace collection (i.e. without CFI checking). Alternatively, we propose to build in future work a monitoring framework derived from our solution (i.e. based on anomaly detection and CFI checking), applying the method introduced by Hasan et al. with Contego framework [66]. With this approach, we define two modes of executions for the system:

- **passive mode:** the ADS runs and controls CF monitoring requests, which are all rejected by the server; i.e. the CFI monitor and more specifically the trace collector are inactive.

  This execution mode applies by default in the system, since it does not delay the monitored task.

- **active mode:** the CFI monitor collects CF traces and performs CFI checking, while the ADS keeps executing: depending on the availability of hardware resources, the ADS can run in background on the same set of CPU cores than the CFI monitoring tasks, but with a comparatively lower priority, or concurrently on a separate set of CPU cores.

  This mode is triggered by the system, once the ADS detects an anomaly. as it induces a slowdown of the monitored application, it applies for a limited time period to perform additional security checks: the server accepts full or partial monitoring requests (trace collection) for CFI checking.

We develop this approach further in Chapter VII with the introduction of an analytical multi-mode HIDS.

In our current implementation, we define the time overhead as the slowdown of the monitored application, which corresponds mainly to the trace collection time (Section VI.5.1); i.e. the times when the monitored task is stopped for collecting the traces from the CoreSight local buffer into main memory. CoreSight hardware tracing framework allows concurrent read and write operations to the trace buffer: the application could execute, while the trace collector stores buffer traces to main memory. In such setup, the monitored application's slowdown is limited to the internal overhead, which is much lower than trace collection overhead. However, such configuration could degrade security monitoring properties. We observed in the most pessimistic scenario (`inffcall` with $2KB$ buffer configuration) that the time for collecting one buffer of traces corresponds to more than 6 inter-arrival times of monitoring requests: with concurrent trace collection, the buffer content can, in an extreme case, be overwritten 6 times. Therefore, such monitoring framework would still occasionally require the ability to stop the monitored application to perform security monitoring via trace collection.

### Alternative ARM CoreSight Trace Buffer Configurations

The ETB size threshold configuration impacts the monitoring request rate $T_M$ and the trace collection time $C_M$: the smaller the threshold value, the shorter $T_M$ and $C_M$ times (i.e. the configured buffer is smaller). Thus, for a fixed monitoring budget $B_M^A$, we can observe more monitoring events with smaller threshold values ($B_M^A = C_M * N_M^A$). With such configurations, we could refine the granularity of the ADS service; i.e. reducing the observation time window $T_w$ to build a more precise application profile. This could contribute to reduce the false-negatives rate (i.e. decreasing the probability for two applications to have the same profile) and speed up the detection. Though, an observation of shorter time windows $T_w$ probably has a higher variability because of the execution context (delayed memory access, speculative execution, etc.), which could affect the detection accuracy.

### Framework Deployment in Complex Mixed-Criticality Platforms

Our framework is for now able to monitor a single-threaded task. To extend the monitoring to multiple applications, it is possible to include the application UID in addition to the thread UID in the context of CoreSight traces: i.e. using these identifiers, our solution would be able to distinguish several threads of distinct applications.

With our current architecture, the monitored core is unused while the monitored task is stopped for trace collection; i.e. dummy operations executed by the spinner thread of the monitor (Section VI.1.1). This unused time should be allocated to other tasks running actual work.

### Strategies for Decoupling Trace Collection from Trace Analysis

If the execution time $T_{analysis}$ for analyzing a buffer of trace is shorter than the nominal inter-arrival time $T_M$ of the trace collection task $\tau_M$, we can run both trace collection and trace analysis in parallel to the monitored applications, using an intermediate buffer to store the collected traces (with a size equal to the size of buffer). However, when $T_{analysis} > T_M$ – such as in our case – the trace collector would accumulate more traces than the maximum amount of traces to be analyzed in the available period of time: checking all the traces for an unlimited time would require an unlimited intermediate buffer. To avoid such unrealistic resource requirements, we can apply the following strategies:

- **Accelerating CFI Path Reconstruction and CFI Checking** so that $T_{analysis} = T_{cfg} + T_{cfi} \leq T_M$. This can be achieved by (i) optimizing the implementation, (ii) offloading the tasks to an accelerator such as a FPGA, or (iii) dedicating multiple cores for these tasks.

- **Dropping traces** with limiting the monitoring coverage. This corresponds to the approach described in this chapter, even though we focused our analysis and experiment on trace collection phase only without trace analysis. To consider CFG path reconstruction and CFI checking in our approach, we can refine the monitor task definition in our task model (Section VI.2.2), so that the WCET $C_M$ for monitoring also integrates the time overhead of the monitored application for trace analysis $T_{analysis}$; i.e. for a given buffer of traces, if trace analysis runs sequentially after trace collection, $C_M = t_{copy} + T_{analysis}$. The buffer design depends on the strategy for prioritizing traces to analyze: i.e., recent traces first with FIFO or synchronous analysis via a ring buffer. However, every time traces are dropped, the monitor must reinitialize the current backward-edge and forward-edge CFI status.

# VI.8 Related Work

## VI.8.1 Security Integration in Real-Time Systems

Hasan et al. propose generic methods to integrate security tasks in a real-time systems, which address the trade-off between security monitoring and system impact. The authors first introduce an opportunistic server solution [67], which is later generalized with the adaptive framework Contego [66]. However, this framework cannot apply to our practical use-case context: while their opportunistic approach supposes a deferrable execution of the security tasks, our use-case requires the server ability to preempt the monitored task (as explained in Section VI.2.2).

## VI.8.2 Runtime Threat Detection for Mixed-Criticality Systems

### Timing Deviation Based Threat Detection

Bellec et al. implement a hardware framework [24] for embedded real-time systems, to detect CF hijacking threat such as ROP attacks. Their solution monitors the execution time of pre-defined code regions of the program to protect: when a monitored code region's execution exceeds its WCET, the framework detects a threat. However, the inherent pessimism of the WCET metric could be exploited by a knowledgeable adversary to hide additional malicious execution time. Alternatively, our ADS monitoring approach differs by observing the CF at a higher level, on fixed time windows of execution.

### Control-Flow Integrity for Mixed-Criticality Systems

CFI Monitoring solutions generally affect the execution of the monitored task, especially software implementations. TrackOS [127] is a RTOS designed to integrate CFI monitoring. Before runtime, binary static analysis generates a call graph for each monitored application, to control every function call and return instructions of monitored tasks at runtime. This solution does not allow full CF coverage: it only considers functions, while our framework also monitors additional branch instructions (e.g. loop, conditional statements). RECFISH [156] is another CFI monitoring solution based on FreeRTOS open-source real-time system. Because it requires program instrumentation, this approach is less suitable for deployment in certified legacy systems, in comparison our modular framework.

CFI monitoring is traditionally implemented in software. Hardware implementations usually come with a reduced performance overhead and less interferences in monitored software execution (limited instrumentation). The survey [36] provides a state of the art of hardware based CFI monitoring solutions. Most of the initiatives in the survey are still research projects: from 21 studied CFI implementations, only two run on common available hardware. Several frameworks involve processor-tracing features on common hardware platforms for CFI monitoring: mainly Intel Processor-Tracing [61, 59, 104] and ARM CoreSight [92, 96]. To our knowledge, the framework used for our work [92] is the only hardware-assisted solution targeting the deployment in critical systems with timing constraints. From the observation of a significant performance overhead for a pessimistic monitoring use-case, the authors identify the problem addressed in our work: i.e. assuring a predictable monitoring solution, which achieves a practical trade-off between security monitoring coverage and performance impact. We also extend their analysis based on two use-cases, by monitoring a set of representative and pessimistic applications.

## VI.9 Conclusion

We described and evaluated a first safety-aware method to integrate hardware-assisted CF based security monitoring into a MCS. Our approach leverages a periodic server to predictably collect CF traces for CFI checking, while an ADS service monitors the security of the complete execution of the monitored application. We evaluated the framework on PikeOS industrial real-time OS with ARM CoreSight support, using a set of programs from TACLeBench benchmark. While limiting the time performance overhead to 10% slowdown of the monitored task, our ADS correctly identified almost all samples, with no false-positives and rare false-negatives.

VII

# An Adaptive Host Intrusion Detection System Approach for Embedded Mixed-Criticality Systems

In Chapters V and VI, we investigated several intrusion detection solutions, associated with various intrusiveness levels: using heuristics and specification based methods, monitoring different events such as HPC, system calls, and CF traces. In particular in Chapter VI, we implemented two parallel security modes: a partial hardware tracing based CFI monitoring and CF events based detection. In this chapter, we propose to develop this approach towards adaptive security analysis: i.e. reducing the performance overhead for monitoring, while triggering intrusiveness to apply in-depth security analysis when we suspect the occurrence of a threat.

We consider the following requirements, to discuss the development of an adaptive HIDS:

- The integration of the solution should not compromise the predictability of the system.
- The solution must be able to detect intrusions for different monitored application use-cases. Hence, this chapter introduces a generic multi-mode approach for intrusion detection.

We develop our motivations in Section VII.1.1 and describe the multi-mode based HIDS solution in Section VII.1. In Section VII.2, we propose an evaluation of the approach according to criteria introduced in Section IV.5. Section VII.3 presents our concluding remarks and suggest future work to investigate.

## VII.1 A Multi-Mode Based Host Intrusion Detection System

In this section, we describe the multi-mode based HIDS solution. First, we define our motivations for deploying such method in an industrial embedded MCS (Section VII.1.1), addressing the trade-off between reduced intrusiveness (and time overhead) to limit the impact on system schedulability, performance, and detection coverage. Second, in Section VII.1.2, we present an overview of the multi-mode based solution and we detail the different security modes. Then, we model the system under monitoring in Section VII.1.3.

### VII.1.1 Motivations for combining intrusion detection approaches

Our goal is to develop an adaptive HIDS solution, which combines several levels of security – i.e. security modes of execution – adapting the degree of runtime intrusiveness (and so the performance impact) depending on the likelihood of threat occurence. Hence, we propose to apply light-weight background monitoring: in our case, this corresponds to HPC monitoring (Section V.3) or ARM CoreSight full buffer events (Section VI.3.1). Such heuristics based detection are likely to come with false-positives; thus, upon detection in this default execution mode, we enter the second monitoring mode, a specification based detection to confirm the occurence of a threat.

The main benefit of a multi-mode solution is to apply specfication based monitoring, limiting the significant performance overhead and runtime intrusiveness generally associated with such method (e.g. CFI checking). Of course, this approach makes sense if the two detection modes are able to detect the same types of threats. Possibly, the default monitoring level induces false-positives. We develop the dependencies on the monitoring methods further in Section VII.2.1 for the overall detection accuracy.

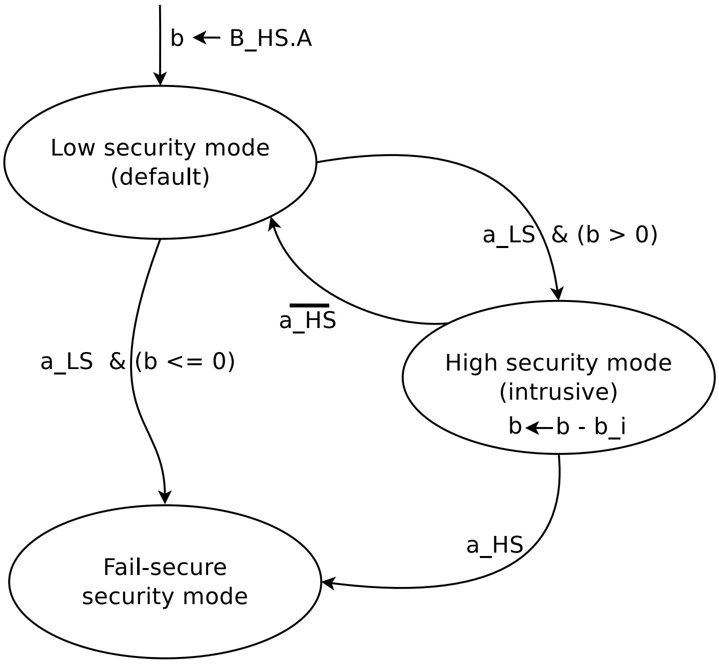

b ← B_HS.A
Low security mode (default)
a_LS & (b > 0)
$\overline{\text{a\_HS}}$
a_LS & (b <= 0)
High security mode (intrusive)
b←b - b_i
a_HS
Fail-secure security mode

**Figure VII.1:** *State machine representing the HIDS execution modes*

### VII.1.2 Overview of the Multi-Mode Host Intrusion Detection Solution

We introduce a multi-mode HIDS solution to combine several intrusion detection mechanisms with the following goals:

- **Low security checks** – apply continuous light-weight and transparent security checks, which may induce low detection accuracy and false-positives.

- **High security checks** – apply sporadic security checks, which come with high detection accuracy and no false-positives, but at the cost of a heavy impact on system performance.

We build our multi-mode HIDS using the following attributes:

- We define three modes of execution:
  - **Low security mode** – execute continuous transparent security checks.
  - **High security mode** – perform sporadic intrusive security checks.
  - **Fail-secure security mode** – apply protective measures once an intrusion has been detected by the multi-mode HIDS.

- For the predictability of the monitoree's execution, we set a maximum time budget to execute intrusive security mechanisms of the high security execution mode. Basing our approach on the problem model defined in Section VI.2 with the server based method to trigger CF trace collection, we introduce $B_{HS}^{A} \in \mathbb{N}$ the time budget to execute monitoring routines in the high security execution mode; this budget corresponds to the maximum time for monitoring, within one period of execution of the monitoree task $\tau_A$.

Figure VII.1 shows a state machine representing transitions between different execution modes of the multi-mode HIDS. We describe the monitoring mechanisms of our solution in the subsections below.

#### Initialization of the Multi-Mode Host Intrusion Detection System

When the monitoree starts to execute, it enters the low security mode, which continuously apply during runtime. The system initializes $b$ the time budget to execute intrusive intrusion detection mechanisms in high security mode: i.e. $b = B_{HS}^{A}$.

We consider two Boolean signals – i.e. positive or negative intrusion detection – for intrusion detection in the low and high security execution modes. We designate the signals for intrusion detection in low and high security modes respectively by the variables $a_{LS} \in \{False, True\}$ and $a_{HS} \in \{False, True\}$, which we both initialize to the value $False$ (i.e. no intrusion is detected).

### Low Security Mode

In this mode, we perform light-weight monitoring. With the investigated solutions of previous chapters, we propose to monitor HPC and ARM CoreSight events periodically, using a fixed time window. By default in fully passive mode, we do not apply CF trace collection; i.e. all CF tracing requests are rejected. However, integrating CF trace collection following the server solution in Section VI.3 could improve the monitoring coverage in high security mode; we develop this approach further in the next subsection.

Depending on the implementation configuration, low security monitoring can continuously run; i.e. also in high security mode in parallel of the additional security checks.

### High Security Mode

The goal of this execution mode is to apply on demand in-depth security checks, which require more system resources and intrusiveness compared to routines of the low security mode. Hence, we enter the high security mode occasionally when an intrusion suspicion has been raised by the low security mode; i.e. $a_{LS} = True$.

When the monitoring budget $b$ has been fully exhausted, even though we cannot apply further high security checks, we can assess that the high amount of alarms triggered in the low security mode corresponds to an intrusion. Depending on the solution certifiability constraints, we could, instead of switching to the fail-secure mode as illustrated on Figure VII.1, generate a passive alarm to be processed offline.

Considering the solutions developed in previous chapters, CFI checking represents a suitable specification based approach to deploy in this mode. We can adapt the server based CF trace collection solution (Section VI.3) to implement the dual security modes approach. On one hand, we can perform CF trace collection in low security mode using the predictable server method introduced in Section VI.3, using stored CF traces only in high security mode when $a_{LS}$ alarm has been raised by low security mode routines. The benefit is to run the CFI analysis on traces generated in the suspicious window, which may be overridden at the time when the HIDS monitor triggers the security mode: with this implementation, we could detect attacks, which would stop before the CF buffer is filled by new benign traces and the window is detected suspicious in low security mode. In this case, we distinguish $B_M^A$ the time budget for collecting CF traces in low security mode from $B_{HS}^A$ the time budget for CFI checking in high security mode. On the other hand, we can trace CF data in high security mode to implement a fully passive low security mode (i.e. without impact on the monitoree's execution time). In this configuration, the budget for trace collection $B_M^A$ must be included in the security mode's time budget $B_{HS}^A$.

In addition to the trace collection method, we also need to define how to distribute $B_{HS}^A$ budget during $T_A$ the period of the monitoree task. We list the following strategies:

- apply high security mode until the budget $B_{HS}^A$ gets exhausted.
- apply high security mode during a limited time period; returning to the low security mode at the end of the period or when the budget $B_{HS}^A$ is exhausted.

- apply the high security mode on a limited set of traced events (e.g. hardware event). For example, checking CFI of the current CF trace buffer only:
  - we systematically analyze the current buffer, even though it is not full.
  - we check the CF trace buffer when it becomes full (ARM CoreSight hardware event).

  Alternatively, we could also run high security monitoring mode on certain statically selected critical code regions of the monitored program.

#### Fail-Secure Security Mode

The execution of the monitoree enters the fail-secure security mode when the multi-mode HIDS has detected an intrusion; the detection corresponds to one of the following scenarios:

- An intrusion has been detected in the high security mode.
- An intrusion has been detected in the low security mode, and the budget $B^{A}_{HS}$ of the high security mode has been completely exhausted.

In this protective execution mode, we can deploy security measures to avoid the adversary to successfully complete the attack. Section IV.4.3, provides insights on recovery strategies.

### VII.1.3 System Model with the Multi-Mode Intrusion Detection Solution

For our analysis, we consider the following configuration. The monitoree starts executing entering the low security mode. In the low security mode, the intrusion detection mechanisms of the low security mode apply continuously (i.e. the system allows concurrent execution of low security and high security tasks. The intrusion detection mechanisms are fully transparent. They can execute without impacting the monitoree's execution, as, according to our methodology (Section IV.4), the HIDS monitor runs on a dedicated hardware as long as the deployed security methods do not require instrumentation of the monitoree.

Upon detection of an intrusion in low security mode, if the high security monitoring time budget is long enough, we enter the high security mode. In this mode, we apply intrusive monitoring with CF trace collection. If the high security mode does not detect an intrusion, we return to low security mode of execution. Upon detection of an intrusion in high security mode, an intrusion is detected by the multi-mode HIDS: we enter in the fail-secure security mode. Once we have reached the fail-secure mode, we remain in this mode until the end of the execution of the monitoree. Finally, if we detect an intrusion in low security mode and the budget is too short to execute high security monitoring routines, the multi-mode detects an intrusion and enters the fail-secure execution mode.

### Real-Time Task Model with Security Monitoring

We use the same task model as in the previous chapters (Section V.3.3 and Section VI.2.2). We consider 3 main tasks in the systems:

$$\begin{cases} \tau_A & = (C_A, T_A, P_A) \\ \tau_{HS} & = (C_{HS}, T_{HS}, P_{HS}) \\ \tau_{LS} & = (C_{LS}, T_{LS}, P_{HS}) \end{cases} \tag{VII.1}$$

which respectively correspond to the monitored task, the high security mode task, and the low security mode task. For simplification, we assume that $\tau_A$ is a periodic task, as for HPC monitoring in Section V.3.3 and for ARM CoreSight hardware events based monitoring in Section VI.2.2. In the low security mode, we use time window based monitoring methods (with HPC, ARM CoreSight hardware events). Hence, we define $\tau_{LS}$ as a periodic task. $\tau_{HS}$ is a sporadic task, since it is triggered after an alarm as been raised by $\tau_{LS}$ task.

We consider the same offset for all tasks ($O = 0$). $C_A, T_A, C_{LS}, C_{HS}$ values depend on the use-case, implementation and deployment platform. We define task priorities and $T_{HS}$ period analytically, while we set $T_{LS}$ according to previous work (CF trace collection rate, HPC monitoring time window), depending on the use-case and security criteria.

$T_{LS}$ configuration depend on $T_A$ value. As we apply low security monitoring on each monitoree job of $\tau_A$: $\exists k \in \mathbb{N}^*, k * T_{LS} = T_A$. As $\tau_A$ and $\tau_{LS}$ have the same offset, we can consistently perform time window based monitoring across different job executions of $\tau_A$. This synchronization is a necessary requirement for CF based ADS developed in Section VI.3, as we measure minimum and maximum values of system events for each window. This is not specifically required for HPC monitoring solution using a ML model to predict traces; though, such setup may complicate the learning phase of the ML model as the training set must include sequences of all possible runtime configurations.

### System Processor Configuration

We define P the set of CPU cores of the system. We can simplify the system with 3 sets: $\mathrm{P}_{LS} \subseteq \mathrm{P}, \mathrm{P}_{HS} \subseteq \mathrm{P}, \mathrm{P}_A \subseteq \mathrm{P}$, which respectively are the set of CPU cores assigned to the low security, high security modes, and the monitoree task. $\mathrm{P}_{HIDS} = \mathrm{P}_{LS} \cup \mathrm{P}_{HS}, \mathrm{P}_{HIDS} \subseteq \mathrm{P}$ is the set of CPU cores assigned to the HIDS monitor.

Because the HIDS monitor runs on dedicated hardware, we set, when possible: $\mathrm{P}_{HIDS} \cap \mathrm{P}_A = \emptyset$. When this is not achievable – i.e. the high security mode requires intrusive monitoring – we can introduce the additional task $\tau_{TC} = (C_{TC}, T_{TC}, P_{TC} > P_A)$ for trace collection, when tracing requires to stop the monitored task $\tau_A$. We can then assign $\mathrm{P}_A$ to $\tau_{TC}$ so that $\tau_{TC}$ is able to preempt $\tau_A$. Since the low and high security modes can run concurrently:

$$\mathrm{P}_{LS} \cap \mathrm{P}_{HS} = \emptyset \tag{VII.2}$$

#### High Security Mode Configuration

We can define the minimum period of $\tau_{HS}$ as, with $C_{LS}^{(BC)}$ the best-case execution time for the low security task $\tau_{LS}$. It corresponds to the maximum between the WCET for running the high security routine and the minimum inter-arrival time of high security task $\tau_{HS}$:

$$T_{HS} = \max\left(C_{HS}, T_{LS} + C_{LS} - C_{LS}^{(BC)}\right) \tag{VII.3}$$

As we perform trace collection and security analysis in high security mode, we decompose the WCET of $\tau_{HS}$ as:

$$C_{HS} = N_{TC}^{(max)} * (C_{TC} + C_{TA}) \tag{VII.4}$$

with $C_{TC}$ the WCET spent to collect one full trace buffer, $C_{TA}$ the WCET for analyzing the collected traces – e.g. with CFI checking on one full buffer of traces – and $N_{TC}^{(max)} \in \mathbb{N}^*$ the maximum amount of trace collection events, that $\tau_{HS}$ is able to serve in one window of the high security mode. $N_{TC}^{(max)}$ value varies in function of the high security mode configuration. We also set $B_{HS}^{A}$ the overall budget to execute $\tau_{HS}$ within a period of the monitored task $\tau_A$:

$$B_{HS}^{A} = N_{HS}^{A} * C_{HS} \tag{VII.5}$$

with $N_{HS}^{A} \in \mathbb{N}^*$ the maximum amount of high security windows of execution.

## VII.2 Analytical Evaluation of the Multi-Mode Host Intrusion Detection Solution

In this section, we evaluate the multi-mode host intrusion detection solution analytically, following 3 directions: security (Section VII.2.1), system performance (Section VII.2.2), and system schedulability (Section VII.2.3). For this analysis, we consider the multi-mode HIDS framework configuration and system model described in Section VII.1.3; we use the evaluation approach described in Section IV.5.

### VII.2.1 Security Evaluation

Through this thesis, we investigated HIDS solutions, which are able to detect unknown threats. By extension, a multi-mode HIDS combining several of these detection methods is also able to detect the same type of threats.

In our security evaluation, we determine the intrusion detection efficiency of our multi-mode HIDS solution, notably through detection accuracy and latency metrics. We then define the coverage of the monitoring solution.

#### Detection Accuracy Scope

As defined in Section IV.2.1, we introduce for the program of a given monitored task $\tau_A$, $G^{(A)}, B^{(A)}$, respectively the execution graph and the set of all transitions in the execution graph. We consider an abstract representation of $G^{(A)}$ and $B^{(A)}$, because in this chapter the HIDS monitor involves several representations of the monitored program (i.e. counting events in fixed time windows, controlling branches executed at instruction level).

Using a given intrusion detection model $M^{(A)} \subseteq B^{(A)}$, we can determine for any execution path $e^{(A)}(t) \subseteq B^{(A)}, t \in \mathbb{N}^*$, whether it corresponds to an intrusion or benign execution; i.e. $e^{(A)}(t) \subseteq M^{(A)}$. Hence, with $E^{(A)}$ the set of all execution paths of the monitored program, we can split it in 2 distinct sets (VII.6):

$$E^{(A)} = E^{(A)}_{malicious} \cup E^{(A)}_{benign}, E^{(A)}_{malicious} \cap E^{(A)}_{benign} = \emptyset \tag{VII.6}$$

so that $E^{(A)}_{malicious}, E^{(A)}_{benign}$ respectively represent the set of malicious and benign executions.

The intrusion detection model $M^{(A)}$ splits the set of all execution paths of $\tau_A$ in detected intrusions and undetected executions (VII.7):

$$e \notin M^{(A)} \Leftrightarrow e \text{ is a detected intrusion} \tag{VII.7}$$

Our configuration involves two intrusion detection methods for high and low security modes. Therefore, we introduce $M^{(A)}_{HS}, M^{(A)}_{LS}, M^{(A)}_{HIDS}$ the HIDS models, respectively corresponding to the high, low security modes, and the global HIDS monitor for a given high security monitoring budget $B^{A}_{HS}$. Detections based on $M^{(A)}_{HS}$ and $M^{(A)}_{LS}$ models cover executions detected respectively in high and low security checks, which trigger the fail-secure mode; n.b. in the low security mode when $B^{A}_{HS}$ has been fully exhausted. We define the HIDS model $M^{(A)}_{HIDS}$ with (VII.8), so that it detects an intrusion upon detection in the low security mode when the budget has been exhausted ($M^{(A)}_{LS}$) or in the high security mode ($M^{(A)}_{LS}$). We illustrate the application of a given HIDS solution to monitor a generic application on Figure VII.2, dividing the execution paths between actual malicious and benign executions sets and between detected malicious and benign sets.

$$M^{(A)}_{HIDS} = M^{(A)}_{HS} \cap M^{(A)}_{LS} \tag{VII.8}$$

Our objective towards HIDS certification is to reduce the false-positive rate of the HIDS. Consequently, we propose to leverage specification based intrusion detection in high security mode so that it is free from false-detection (VII.9).

$$E^{(A)}_{benign} \subseteq M^{(A)}_{HS} \tag{VII.9}$$

Any threat detected by the HIDS monitor must be detected by the low security mode: it is either confirmed in the high security mode or, if $B^{A}_{HS}$ has been fully exhausted, we can configure the system to automatically detect a threat. We define $E^{(A)}_{missed}$ the set of malicious executions that cannot be detected in the low security mode, even though they are detectable by the high security mode (VII.10). Our goal is to combine high security

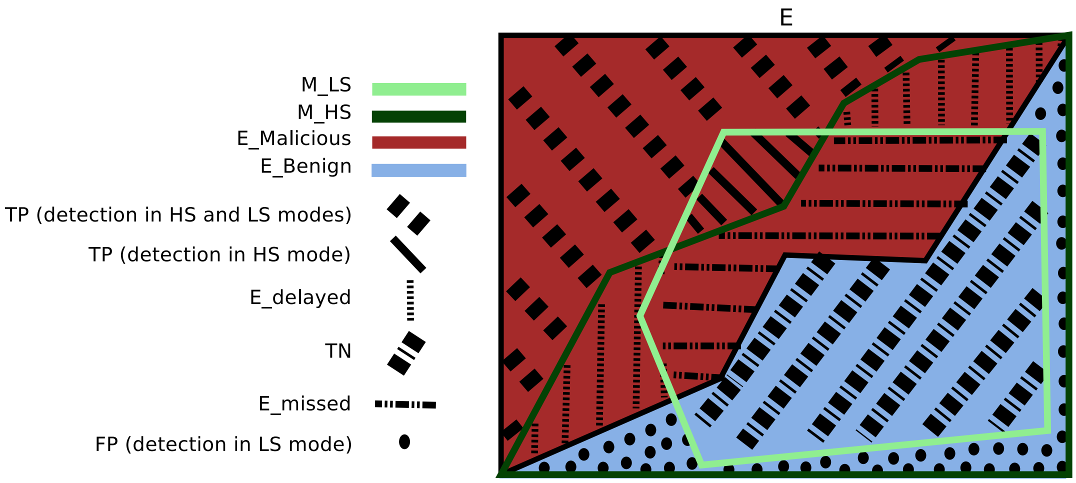

E
M_LS
M_HS
E_Malicious
E_Benign
TP (detection in HS and LS modes)
TP (detection in HS mode)
E_delayed
TN
E_missed
FP (detection in LS mode)

**Figure VII.2:** *An example of execution paths classification for a generic application, using the multi-mode host intrusion detection system*

and low security monitoring models to reduce the amount of elements in the set $E^{(A)}_{missed}$.

$$E^{(A)}_{missed} = M^{(A)}_{LS} \cap E^{(A)}_{malicious} \setminus M^{(A)}_{HS} \qquad \text{(VII.10)}$$

$E^{(A)}_{delayed}$ represents the set of executions which cannot be detected in the high security mode, even though they are detectable in the low security mode (VII.11). In this case, the HIDS can still detect the threat, when too many anomalies are detected in low security mode, so that $B^{A}_{HS}$ budget is exhausted.

$$E^{(A)}_{delayed} = M^{(A)}_{HS} \cap E^{(A)}_{malicious} \setminus M^{(A)}_{LS}, \qquad \text{(VII.11)}$$

For the HIDS approach to be relevant, we assume that it must require a reduced set $E^{(A)}_{missed}$. We consider our case – i.e. using CFI checking and CF related hardware events based heuristics – practicable, since the type of monitored events are directly related to the instructions executed by the monitored program: a threat modifying the CF execution is likely to induce anomalies, both when analyzing traces with heuristics and using CFI checking.

Finally, to improve detection accuracy, we set the following goals:

- increase the detection coverage (i.e. reduce false-negatives) of the high security mode, to reduce the amount of elements of the set $E^{(A)}_{malicious} \cap M^{(A)}_{HS}$.
- increase the detection coverage of the low security mode, to reduce $E^{(A)}_{missed}$.
- reduce or increase the budget $B^{A}_{HS}$ in function of the system performance overhead constraints and more specifically depending on the false-positive rate in low security mode; i.e. the set of delayed detectable threats $E^{(A)}_{delayed}$.

### Detection Latency Scope

Let $\tau_A$ be a task to monitor by a multi-mode HIDS monitor; we define $E^{(A)}$ the set of all possible executions paths of $\tau_A$ and $M_{LS}^{(A)}$, $M_{HS}^{(A)}$ the low and high security models used by the HIDS monitor.

To simplify the analysis, we refine our attack scope to malicious executions, which are continuously detectable by the specified detection path (i.e. in low or high security level). Let consider a malicious execution $e \in E_{malicious}^{(A)}$. The attack starts at a given time of execution, when the remaining budget for the high security mode is $b \in \mathbb{N}^*$, so that $C_{HS} \leq b \leq B_{HS}^{A}$.

Figure VII.3 provides an example of schedule for a given execution $e$ of the application $\tau_A$. The HIDS monitor systematically performs periodic checks in the low security mode. Hence, the maximum time from the attack start until the first low security level check returns is: $W_{wait} = T_{LS} + C_{LS}$. The detection latency varies depending on the two possible detection paths:

- $W_{detect}^{(HS)}$: the maximum latency to detect the attack using the high security mode; i.e. $e \in E_{malicious}^{(A)} \setminus M_{HS}^{(A)}$ and $b \geq C_{HS}$. It corresponds to the sum of $W_{wait}$ and $C_{HS}$, the WCET to execute the high security level check (VII.12).

- $W_{detect}^{(LS)}$: the maximum latency to detect the attack using the low security mode, when the budget is full ($b = B_{HS}^{A}$). In this case, $M_{HS}^{(A)}$ cannot be used to detect the attack (i.e. $e \in E_{delayed}^{(A)}$) or the time to execute high security level check is longer ($C_{(HS)} \leq B_{HS}^{A}$); the example on Figure VII.3 corresponds to the second scenario. $W_{detect}^{(LS)}$ latency corresponds to the sum of $W_{wait}$ and $N_{HS}^{A} * T_{LS}$, where $N_{HS}^{A} = \lfloor \frac{B_{HS}^{A}}{C_{HS}} \rfloor$ is the maximum amount of executed checks in high security mode (VII.13).

  With (VII.13), we derive the general function $w_{detect}^{(LS)}$ to describe the maximum latency in function of the budget $b$ at a given time of the execution with (VII.14).

$$W_{detect}^{(HS)} = T_{LS} + C_{LS} + C_{HS} \tag{VII.12}$$

$$W_{detect}^{(LS)} = W_{wait} + N_{HS}^{A} * T_{LS} = (1 + \lfloor \frac{B_{HS}^{A}}{C_{HS}} \rfloor) * T_{LS} + C_{LS} \tag{VII.13}$$

$$\forall b \in \mathbb{N}, b \leq B_{HS}^{A}, w_{detect}^{(LS)}(b) = (1 + \lfloor \frac{b}{C_{HS}} \rfloor) * T_{LS} + C_{LS} \tag{VII.14}$$

If we consider that the malicious execution $e$ is detectable by both high security and low security models ($e \in E \setminus (M_{LS}^{(A)} \cup M_{HS}^{(A)})$), we can derive the global detection latency of the HIDS monitor $\forall b \in \mathbb{N}, b \leq B_{HS}^{A}, w_{detect}^{(HIDS)}(b)$ in function of $W_{detect}^{(HS)}$ and $w_{detect}^{(LS)}(b)$ with (VII.15). We note an evolution of the rapidity of the security modes to detect an intrusion during the execution time: while the detection latency $W_{detect}^{(HS)}$ remains fixed,

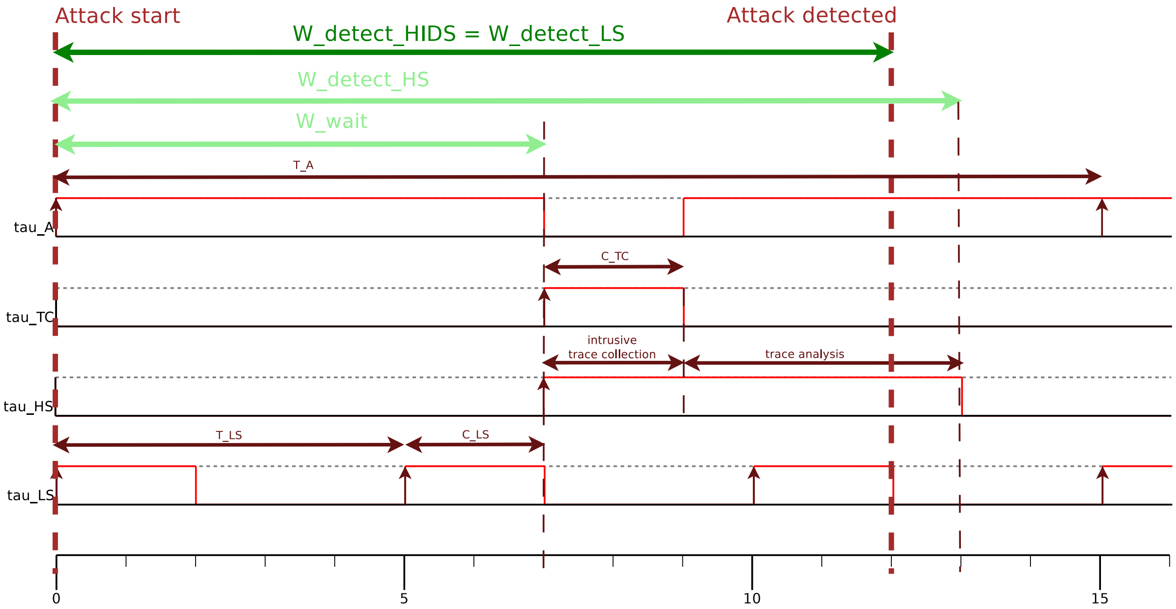


**Figure VII.3:** *An example of schedule for a malicious execution of the periodic task $\tau_A = (C_A = 13, T_A = 15, P_A)$ and a multi-mode intrusion detection system composed of the periodic task $\tau_{LS} = (C_{LS} = 2, T_{LS} = 5, P_{LS})$ and the sporadic task $\tau_{HS} = (C_{HS} = 6, T_{HS} = 6, P_{HS})$. As explained in Section VII.1.3 to model intrusive trace collection (i.e. the HIDS must preempt $\tau_A$ for trace collection), we introduce the additional sporadic task $\tau_{TC} = (C_{TC} = 2, T_{TC} = 6, P_{TC} > P_A)$ to preempt $\tau_A$. The platforms supports 4 CPU cores, where $\tau_{LS}$ runs on CPU core 1, $\tau_{HS}$ runs on CPU core 2, $\tau_A$ and $\tau_{TC}$ run on CPU core 3.*

the detection latency $w_{detect}^{(LS)}(b)$ decreases because the budget $b$ decreases as time goes forward.

$$\forall b \in \mathbb{N}, b \leq B_{HS}^{A}, w_{detect}^{(HIDS)}(b) \begin{aligned} &= \min\left(w_{detect}^{(LS)}(b), W_{detect}^{(HS)}\right) \\ &= T_{LS} + C_{LS} + \min\left(\lfloor \frac{b}{C_{HS}} \rfloor, C_{HS}\right) \end{aligned} \tag{VII.15}$$

For a given implementation of the HIDS monitor, we cannot reduce the WCET of monitoring tasks: $C_{TC}$, $C_{CFI}$, $C_{LS}$. Nevertheless, we could perform the trace collection in low security mode, to reduce the detection latency in the high security mode, by reducing $C_{HS}$ value (VII.4). According to this definition, an alternative would be to reduce the set of analyzed traces ($N_{TC}^{(max)}$). $T_{LS}$ represents a crucial parameter to configure $T_{LS}$ to reduce the latency of the HIDS monitor. Though, it is constrained by $C_{LS}$ and its configuration also has a major impact on the detection accuracy: too short periods of observation are likely to come with a higher false-positive rate, since at a given time of execution the observations of system events may vary in function of the context (e.g. cache access time, processor pipeline state).

### Security Monitoring Coverage

We propose to evaluate the security monitoring coverage of a given HIDS using two metrics for each security mode:

- **Low security mode:** $w_{uncov}^{(LS)} \in \mathbb{N}, r_{cov}^{(LS} \in \mathbb{R}, 0 \leq r_{cov}^{(LS)} \leq 1$ respectively represent the maximum uncovered window and the overall ratio of covered execution in low security monitoring mode.

- **High security mode:** $w_{uncov}^{(LS)} \in \mathbb{N}, r_{cov}^{(HS} \in \mathbb{R}, 0 \leq r_{cov}^{(HS)} \leq 1$ respectively represent the maximum uncovered window and the overall ratio of covered execution in high security monitoring mode.

On one hand, since we assume that low security mode executes continuously, possibly in parallel of the high security mode, this mode fully monitors the execution time of the monitoree. It analyzes the execution at periodic endpoints, between current and previous endpoints. Consequently, the low security mode covers the whole execution of the monitored program:

$$w_{uncov}^{(LS)} = 0, r_{cov}^{(LS)} = 1 \tag{VII.16}$$

On the other hand, the high security mode applies discontinuously on a potentially partial set of the execution. $r_{cov}^{(HS)}$ and $w_{uncov}^{(HS)}$ must be determined by the system integrator, depending on the use-case: i.e. monitored program, deployment platform. We can determine the relative continuity configuration ratio $r_{cont}^{HS,A} \in \mathbb{R}, 0 \leq r_{cont}^{HS,A} \leq 1$ for a given budget $B_{HS}^{A}$, so that:

$$r_{cont}^{HS,A} = \frac{N_{TC}^{(max)}}{N_{HS}^{A}} \tag{VII.17}$$

## VII.2.2 Performance Evaluation

As previously discussed, we reserve dedicated hardware for monitoring tasks that can execute concurrently to the monitored task; though, high security checks may require intrusive trace collection. Hence, the performance overhead to deploy the HIDS monitor depends on the implementation.

### Time Overhead

In low security mode, program monitoring runs passively: it does not interfere with the monitoree's execution. Consequently, the time overhead of the HIDS monitor corresponds to the high security mode; we identify two criteria impacting this overhead:

- $B_{HS}^{A}$: the budget for running the high security mode to control and configure the maximum time overhead of the HIDS monitor on the monitored task.

- $c_{switch}^{A} \in \mathbb{N}$: the time to switch between the low and high security modes during a period of execution of the monitored task $\tau_A$. This depends on the budget $B_{HS}^{A}$, and more precisely $N_{HS}^{A}$, and $T_{LS}$: considering the same time $c \in \mathbb{N}$ for switching from the low to the high security modes and from the high to the low security modes, we derive $c_{switch}^{A} = 2 * c * \lceil \frac{T_A}{T_{LS}} \rceil$. We consider $c_{switch}^{A}$ negligible compared to $W_{HIDS}^{A}$; hence, we do not use it in the detection latency definition (VII.15).

#### Hardware Cost

The hardware cost for integrating the HIDS monitor depends on the platform of deployment and implementation. In our case, to reduce the detection latency, we run the analysis programs of both low and high security modes on dedicate hardware on the platform (i.e. reserved memory area and set of CPU cores). Thus, we propose to have parallel execution environments for the two monitoring modes, including different CPU cores as described in Section VII.1.3.

### VII.2.3 System Schedulability Evaluation

The impact for integrating HIDS on system schedulability depends on the high security mode, when the low security mode runs without interfering with the execution of $\tau_A$. Thus, let consider the execution in high security mode. In this mode, we may need to stop the monitored program to perform trace collection.

For this, we introduce an additional sporadic task for trace collection $\tau_{TC} = (C_{TC}, T_{TC}, P_{TC})$ executing on the same set of CPU cores of $\tau_A$:

- $\tau_{TC}$ must be able to preempt $\tau_A$: $P_{TC} > P_A$.
- The minimum period $T_{TC}$ corresponds to the minimum period of $\tau_{HS}$: $T_{TC} = T_{HS}$.
- $C_{TC}$ represents the WCET for trace collection in high security mode. We define $B^A_{TC}$ the overall budget for intrusive trace collection during one execution of $\tau_A$, so that $B^A_{TC} = N^A_{TC} * C_{TC}$, with $N^A_{TC} = \lceil \frac{T_A}{T_{LS}} \rceil$.

For the schedulability analysis, we derive from $\tau_A$ and $\tau_{TC}$ the new periodic task $\tau_A^{(monitoring)}$, representing the monitored task under intrusive monitoring (VII.18). We can then reiterate the system schedulability analysis replacing the monitored task $\tau_A$ by $\tau_A^{(monitoring)}$.

$$\tau_A^{(monitoring)} = (C_A^{(monitoring)}, T_A^{(monitoring)}, P_A), \text{ where } \begin{cases} C_A^{(monitoring)} & = C_A + B^A_{TC} \\ T_A^{(monitoring)} & = T_A + B^A_{TC} \end{cases} \tag{VII.18}$$

## VII.3 Concluding Remarks

To conclude our analytical approach for multi-mode HIDS in embedded MCS, we propose a set of evaluation criteria to assist future development in Section VII.3.1. We discuss limitations and further topics to investigate in Section VII.3.2 and Section VII.3.3.

### VII.3.1 Evaluation Criteria for Multi-Mode Based Intrusion Detection System Deployment

We consider several aspects for evaluating a multi-mode HIDS approach. Table VII.1 summarizes the impact of these configurations on the characteristics of the system under monitoring. First, we identify two main criteria, which depend on the HIDS use-case, especially on the HIDS monitor implementation and on the technologies involved in low and high security modes:

- **Intrusiveness level** – the trade-off between:
  - detection efficiency as described in Section VII.2.1, through security monitoring coverage and detection latency.
  - time overhead and the impact of the solution on system schedulability (Section VII.2.2 and Section VII.2.3).

  This criteria directly depends on the trace collection implementation: periodically in low security mode or punctually in high security mode. Analyzing the traces corresponding to execution within the anomalous detected window (i.e. in low-security mode), potentially reduces the detection latency by reducing the WCET in high security mode $C_{HS} = C_{TA}$. This configuration may also improve monitoring coverage for threats with short execution time, which end before the next periodic low security check.

- **Security coverage level:** the trade-off between HIDS hardware cost and security efficiency (via detection latency and monitoring coverage). This depends on the separation or the joining of $\tau_{LS}$ and $\tau_{HS}$ tasks on the set of CPU cores; i.e. allowing or disallowing background low security checks in parallel of high security check execution. The parallel execution of low and high security tasks assures a better detection latency. In addition, it can improve the monitoring coverage and detection accuracy in low security mode by measuring the traces at exact period time; i.e. without delay caused by high security checks with higher priority.

Additionally, the set of parameters to configure the HIDS monitor, represent key aspects to influence the characteristics of a given solution:

- $T_{LS}$ – the frequency of periodic checks in low security mode conditions the frequency of high security checks $T_{HS}$ (VII.3): it has a direct impact on the detection latency of the HIDS (VII.15). On one hand, incrementing $T_{LS}$ increases the detection latency. On the other hand, it reduces the system overhead for low security monitoring on the corresponding set of CPU cores $\mathrm{P}_{LS}$, to execute other tasks with lower priority.

  The configuration of $T_{LS}$ potentially influences the detection accuracy in low security mode (i.e. modifying $M_{LS}^{(A)}$). In particular, we empirically observe for HPC monitoring, that while the false-positive rate increases with shorter periodic time window, the false-negative rate increases for longer time windows. We note in this second case (i.e. when increasing $T_{LS}$), that the detection accuracy in high security mode could also be affected, especially when trace collection runs in high security mode. As the delay between two low security checks increases, the attack is more likely to end before malicious traces are collected and analyzed in high security mode, because trace collection is triggered after that low security routines have detected an intrusion.

**Table VII.1:** *Impact of security monitoring configuration on evaluation criteria. The symbols (–, ×, ❍, ●) respectively correspond to (no, negative, mitigated, positive) impact of the configuration parameter on the evaluation criteria.*

| **Evaluation criteria** | | **Parallel exec. of LS/HS modes** | **HS trace collection in LS mode** | **Increasing $T_{LS}$** | **Increasing $N_{HS}^{A}$** | **Increasing $r_{cont}^{(A,HS)}$** |
|---|---|---|---|---|---|---|
| **System integration** | System overhead | × | – | ● | × | – |
| | System schedulability | – | × | – | × | – |
| **Detection efficiency** | Latency | ● | ● | × | – | × |
| | Accuracy | ● | ● | ❍ | ● | ● |
| **Monitoring coverage** | LS mode | ● | – | – | ❍ | ❍ |
| | HS mode (global) | – | – | – | ● | – |
| | HS mode (upon detection) | – | ● | × | – | ● |

- $N_{HS}^{A}, N_{TC}^{(max)}$ – the distribution of the execution time budget of the high security mode $B_{HS}^{A}$ affects the detection latency and the monitoring coverage of the HIDS. These parameters impact system performance overhead and schedulability, as well as security efficiency. More specifically, we identify the following relations:
  - $N_{TC}^{(max)}$ influences the high security check WCET $C_{HS}$ (VII.4) and so the detection latency of the HIDS (VII.15). It also conditions the monitoring coverage: for a given budget with $N_{HS}^{A}$, the greater $N_{TC}^{(max)}$, the more continuous the set of covered traces (VII.17).
  - $N_{HS}^{A}$ influences the overall monitoring coverage of the solution during a period of execution of $\tau_A$ (i.e. $M_{HS}^{(A)}$). Increasing its value improves the coverage: this can also improve the detection accuracy of the HIDS, reducing the false-positives rate due to the false alarms raised in low security mode. Nevertheless, increasing $N_{HS}^{A}$ (i.e. increasing the budget $B_{HS}^{A}$) also induces increased level of intrusiveness and time overhead.

### VII.3.2 Limitations for Using Several Security Modes

One main pitfall of the multi-mode HIDS approach lays first in the detection evaluation for the different modes, i.e. defining $M_{HS}^{(A)}$ and $M_{LS}^{(A)}$ to reduce $E_{delayed}^{(A)}$ and $E_{missed}^{(A)}$. A second challenge is to adapt the configuration to the deployment use-case (type of system, monitored application, and HIDS monitor implementation). For this, we propose to consider the different criteria discussed in Section VII.3.1 and summarized in Table VII.1.

### VII.3.3 Future Perspectives to Develop Multi-Security Modes

We simplified the multi-mode HIDS analysis in this chapter, with the two low and high security modes of execution corresponding to the intrusion detection approaches developed in Chapter V and Chapter VI. In future work, we could extend this analysis by inserting additional security modes of execution (i.e. adding intrusion detection layers).

Alternatively, future development could focus on improving detection efficiency, especially detection accuracy, by combining different modes of execution. Notably, we could use parallel models of execution running concurrently to cover the monitored task's execution. We identify two intrusion detection strategies:

1 - all models detect an intrusion for a given execution path (i.e. a set of traces or time window of observation). This configuration potentially induces low false-positives, but comparatively more false-negatives and long detection accuracy.

2 - any model detects an intrusion. In opposition to the first option, this approach is likely to come with more false-positives, less false-negatives, and shorter detection latency.

Another approach to investigate is to build a single model of execution $M^{(A)}_{HIDS}$ using several data sources, typically analyzing traces with ML. Though, because of the complexity and potential indeterminism in the observed system events (e.g. with HPC traces in time windows), such application would require heavy datasets for training the model of execution.

VIII

# Conclusion

In this chapter, we conclude this thesis following two directions. In Section VIII.1.2, we describe the previous gaps and our contributions in HIDS research for embedded MCS. In Section VIII.2, we discuss the limitations of our HIDS solutions and examine further research perspectives and open questions for HIDS development.

## VIII.1 Contributions to the State of the Art

Security has become an essential need for productizing embedded MCS: the growing openness of these platforms to the outside world, notably due the inclusion of connectivity and user interface features for system low-critical functions, represents an increasing attack surface for adversaries. Securing embedded MCS is challenging because of the potentially strict constraints related to the high-critical functions supported by the platform. The integration of security features must comply with system properties. In particular, security add-ons must induce limited intrusiveness to match the constraints for system certification and acceptable trade-off between security and performance overhead.

In this thesis, we focus our work on the detection of threats in embedded MCS, using HIDS. We highlight the limitations of HIDS related research in Section VIII.1.1: many of the reviewed solutions have limited evaluation, considering system schedulability, performance overhead, and security. We then summarize our contributions in Section VIII.1.2.

### VIII.1.1 Research Gaps in Host Intrusion Detection Systems Literature

In literature, the focus of HIDS research is to improve intrusion detection accuracy. Various solutions are designed to execute in GP computers; in comparison, only few works consider deployment in systems with criticality constraints and embedded constraints. In our literature review, the great majority of papers presents a limited evaluation scope. Notably, for offline HIDS (i.e. the trace analysis runs after the execution of the monitored program) the evaluation does not include detection latency and runtime overhead estimates. None of the reviewed solutions explicitly addresses the problem of compliance with system certification towards industrial deployment: they do not evaluate the impact for collecting traces at system runtime, i.e. analyzing the level of intrusiveness (or transparency) induced by the HIDS. Even though many solutions induce false-positives in the detection, they do not discuss the effects for deployment into an industrial embedded MCS, in terms of certifiability of the HIDS and suitability to protect the execution of high-critical applications. The portability of a HIDS framework among different platforms represents an additional challenge, because of the diversity of embedded MCS; many of the reviewed solutions either require custom-hardware implementation or instrumentation of the monitored software.

We note significant disparities across the range of reviewed papers for the security evaluation of HIDS. For GP computers, many solutions propose an exhaustive evaluation of the detection accuracy, testing the HIDS on a set of representative threat implementations. Other works provide a simple proof of concept of their HIDS, with few simplified attack to demonstrate the soundness of the approach. Alternatively, some papers propose an analytical evaluation without experiment on intrusion implementations.

### VIII.1.2 Contributions Towards Host Intrusion Detection into Embedded Mixed-Criticality Systems

The contribution of this thesis led to several publications [80, 81, 92, 79, 48, 78]. We proposed a survey of HIDS, with a focus on embedded MCS towards industrial deployment [78]. For this, we defined a set of criteria to compare the solutions in terms of system security, certifiability, and performance overhead. To our knowledge, it is the first literature review to present the state of the art of HIDS applying these comparison criteria. We notably discussed main security challenges within the scope of Industry 4.0 use-cases [48]. we identified the gap in HIDS research towards development in industrial embedded MCS as the necessity for the solution to detect threats at system level, leveraging limited intrusiveness and performance overhead. This is the specific problem, which we addressed in this thesis.

We presented a methodology to develop HIDS for embedded MCS. We proposed a generic system architecture based on a mixed-criticality OS to securely and safely integrate HIDS mechanisms into an embedded MCS [81], with a set of methods, criteria, and metrics to evaluate the solution in respect of system schedulability, security, and performance overhead.

We investigated two main intrusion detection approaches:

- **heuristics based** HIDS, using ML: for this, we leveraged low-level system events with system calls [80] and HPC [109] to perform transparent monitoring without instrumentation of the monitored program.

- **specification based** HIDS, with hardware-assisted CFI monitoring: we developed a safety-aware (i.e. predictable and configurable) hardware-assisted CFI monitoring approach [92, 79] for embedded MCS.

We implemented and tested the solutions on an actual industrial embedded MCS platform (with SYSGO's PikeOS hypervisor) [79, 92, 80]. While in literature, evaluation approaches and system architectures vary, we based the HIDS frameworks on our generic system architecture and applied the set of evaluation metrics consistently for the different approaches. Because of the diversity of embedded MCS platforms, there is to our knowledge no widely available threat benchmark for these systems. Therefore, we performed partial analytical security analysis and experimental measurements of the evaluation metrics on a small set of monitored programs. We validated the detection accuracy on several anomalous execution scenarios of the monitored program. This is a limitation compared to related work, particularly in regards to system call based HIDS using heuristics.

Finally, we introduced a multi-mode solution combining several HIDS approaches. The goal is to improve the detection accuracy (reducing the false-positives rate of heuristics based HIDS) while limiting the performance overhead for heavy – in terms of computation load – and intrusive HIDS.

## VIII.2 Future Research Perspectives

We developed several methods for intrusion detection, including ML based monitoring of HPC counters, CFI checking. We proposed a global system architecture and an evaluation approach to integrate such solutions into embedded MCS. We describe the main limitations and possible improvement strategies to deploy our solutions into an industrial context in Section VIII.2.1 and Section VIII.2.2. We finally describe remaining open questions in Section VIII.2.3.

### VIII.2.1 Current Limitations for Intrusion Detection into Embedded Mixed-Criticality Systems

The current HIDS solutions come with several limitations. As mentioned in the previous section, implementing an exhaustive set of actual realistic threats remains an open issue for the security evaluation of embedded MCS, because of our deployment constraints. Building convincing intrusion scenarios is particularly complicated in our context of work as we use SYSGO's PikeOS as RT hypervisor, which is certified to the security evaluation assurance level 3+ [50]; i.e. we assume that system interfaces, such as inter-partition communication, system calls, and OS drivers, are trusted. Consequently, we implemented attack scenarios within the scope of the monitored application. For this, we identified two main options for experimental intrusion detection evaluation:

- developing a generic monitoree program and a set of associated threat scenarios, which are representative of industrial use-cases. This corresponds to building a monitoring testbench, with configurable system interactions and characteristics of the monitored program.

- using a comprehensive set of representative industrial use-cases and threat scenarios. For example, such experimental setup could be a testbench provided by industrials for a given platform.

In this thesis, we implemented simple programs or used benchmarks of computational operations focused programs. We considered a stand-alone monitored application: the experimental setup only involves software bricks, which are involved in HIDS monitoring or required by the monitored program. We excluded complex setups, e.g. which leverage inter-partition communication. We implemented intrusions as deviations of the monitored program's execution. We noted that it is particularly difficult to build attack scenarios to evaluate HIDS based on system call monitoring: this restricts the set of meaningful applications to monitor and the set of attack scenarios. The applications must indeed execute distinct sequences of system calls while the set of system calls depend on the API of the OS.

Because of the lack of representative security threats and the diversity of possible deployment platforms, we provided a set of metrics to evaluate a given HIDS framework, including coverage of the monitored program, slowdown of the monitored execution, detection latency. Though, as we do not know the characteristics of a typical threat – i.e. attack duration, entry points (like cache management, memory usage, network related operations), execution discontinuity – the evaluation needs a comparison of our metrics against the characteristics of representative attacks.

In the context of our heuristics based HIDS approaches, our evaluation highlighted variations in detection accuracy results (false-positives and false-negatives rates), depending on the tested monitored program. Though, we did not investigate further the correlation between the detection accuracy and the characteristics of the program to monitor. For evaluating the correlation, we could identify program characteristics to then develop a generic program with configurable characteristics. A major issue for such HIDS approaches is to collect sufficient trace use-cases to build the reference of execution. Especially for ML based methods, the building phase requires a huge amount of execution samples; Because of embedded MCS constraints, we are not aware of any datasets to assist the generation of the ML model.

### VIII.2.2 Possible Improvements Towards Intrusion Detection in Certified Industrial Systems

We consider four main strategies to pursue our research, for the practical deployment of HIDS into industrial embedded MCS.

**Upgrade our HIDS frameworks** An option is to focus future work on improving security metrics, while reducing the performance overhead for monitoring. Particularly in the context of ML based intrusion detection, we could adapt the building phase of the ML model to reduce detection latency and errors. For the hardware-assisted CFI monitoring framework, we could adapt the implementation, for example using multi-threading or low-level programming language instead of object-level programming language; such enhanced implementation could also involve accelerators like GPU or FPGA to speed-up the detection. Another option to comprehend the scope of detected threats and potential false alerts is to extend the security evaluation with more representative use-cases, threat scenarios, and deployment platforms.

**Extend the multi-mode HIDS approach** An improvement is to extend the generic multi-mode HIDS approach, combining heuristics and specification based intrusion detection methods. On one hand, we can already use the methods developed in this work to implement the multi-mode HIDS; e.g. with HPC based monitoring and CFI checking. On the other hand, we could compare the characteristics of the solutions for a given execution environment, to evaluate the potential performance of a multi-mode HIDS combining the different approaches. In particular, we could evaluate how the HIDS perform, in function of the type of monitored data: CF events, HPC measurements, or executed system calls.

**Improve the HIDS' adaptability** We could work on the adaptability of the HIDS solution in function of the deployment use-case. For this, we would extend our evaluation scope with representative industrial monitored programs and threat scenarios. An beneficial outcome of such evaluation could be, for a given use-case, to highlight a correlation between appropriate system events to monitor and the characteristics of the platform and monitored program. With sufficient information on the deployment use-case, we would be able to assist HIDS setup, by providing a configurable trade-off between security coverage and performance overhead. More particularly with ML based HIDS, we could investigate further how to tune the parameters of the ML model in function of the characteristics of the monitored program.

**Integrate the HIDS to a complex security infrastructure** For deploying HIDS into a software product, the integration of a HIDS module must comply with the software requirements defined in the product development and release lifecycle. In practice, DevSecOps [49] introduces a set of processes to automate the integration of security mechanisms at every phase of the software development lifecycle.

A HIDS represents one element of the security infrastructure, which possibly also includes NIDS, SIEM modules. We could investigate how to connect our HIDS approaches with other security modules of the security infrastructure. For example in the Automotive domain, considering a fleet of embedded MCS such as ECU networks or car systems, HIDS outputs (information on threat detections) could be used by a SIEM system to correlate threat suspicions at different infrastructure levels. We can expect various challenges, depending on the attributes of such global infrastructure. For example, a solution where HIDS outputs are handled by a SIEM running from a Cloud infrastructure, requires appropriate network bandwidth, availability, response time, etc. An alternative option could be to perform runtime adaptive monitoring leveraging the dynamic modification of the HIDS reference model; e.g. by using federated learning and over-the-air update of the HIDS reference model.

### VIII.2.3 Open Questions

In this section, we outline several problems to address by future research for HIDS integration into industrial embedded MCS.

**Runtime adaptive HIDS reference** Following the multi-mode HIDS approach, we could combine different detection solutions in function of the characteristics of the monitored execution and runtime constraints to enhance the properties of the HIDS; i.e. faster detection, less errors, lower performance overhead, etc. Instead of chaining different detection methods, as for the multi-mode HIDS, we could apply several methods (e.g. CF related events monitoring with heuristics) concurrently. As a benefit, we expect that attesting the detection of an intrusion using vote mechanisms would help to improve detection accuracy.

**About the use of ML for HIDS into industrial embedded MCS** A key issue for future work to focus on is to assess the benefit of ML based technologies to detect intrusions, compared to standard heuristics based methods such as sets of rules based on statistical observations of the monitored execution. In our experiments, we used TensorFlow-Lite open-source framework to perform online ML based trace analysis. Even though the framework is adapted to execute in embedded systems (i.e. with limited hardware resources), we could weight the advantages of such implementation, – notably portability, maintenance, and extensive API – compared to custom-hardware based solutions, which we could optimize to reduce implementation costs, e.g. using FPGA or GPU assistance. ML based detection is generally not comprehensive by a human, due to the inherent complexity of the ML model; identifying the cause of a detected intrusion represents a difficult challenge. This diagnostic is even more complicated as the detection accuracy can vary greatly for a given model configuration (i.e. network topology), depending on the monitored program. Therefore, we can question the suitability of ML approaches to detect threats into embedded MCS, especially in the context of deployment into an industrial certified system: notably the use of a ML engine to handle critical tasks represents today an active research problem [86, 146, 7].

**HIDS integration into product development and release lifecycle** The use of DevSecOps [49] processes and best practices to automate HIDS development and integration into a software product remains an open problem, especially in the context of industrial MCS platforms which require safety certification. In the scope of productization, we must consider a global security approach covering the overall security project: through product requirements, platform architecture, implementation, test, and maintenance aspects. Thus, we could investigate in future work how to integrate HIDS mechanisms development in software development lifecycle, using standard processes such as DevSecOps practices. For this, we would introduce a set of generic templates for an easy, ideally automated, plugin of HIDS mechanisms into the MCS product lifecycle. In this case, we must evaluate the integration effort.

Appendix A

# Hardware Performance Counters Selection

## A.1 Offline Monitoring Framework

The test system is detailed on Figure A.1. It is composed of two partitions running on top of PikeOS real-time OS:

- The HPC monitor first initializes the counters with the combination of events to be traced. It then notifies the monitored partition to run the test and waits for its notification. It finally reads the counters and prints measurements.
- The monitored application starts the test upon request of the monitor and sends back a notification once the test has ended.

The test iteration is repeated a certain amount of times so that we can analyze results with statistics. As there are only 6 counters on the platform, and 59 distinct events to trace, we run the test for 10 combinations of counters: [1, ... 6], [7, ... 12], .... [55, ... 59].

The initialization phase of the monitored application is not traced: this part of execution is not relevant for application profiling, it depends mainly on the system software which we suppose secure in this work. We instead trace a specific code region encapsulated in a function.

Both monitored partition and HPC monitor run on the same single CPU. This induces some interferences on the counters. These are negligible as the monitor is blocked during the execution of the monitored partition; however synchronization and context switch operations are counted in the HPC.

For this trace collection case, HPC configuration is enabled from user space, to avoid switching to kernel mode before configuring counters.

## A.2 Results Analysis

Figures A.2 show box plots of the measurements distribution, for each HPC event. Many HPC events (25 over 59) present a similar execution pattern for the test set, as seen on Figure A.2b; table A.1 summarizes corresponding main event types. Generally, it is trivial to distinguish tests which modify the application and tests which change significantly the amount of executed loop iterations. Remaining tests (addqp, addtab, and rmprint)

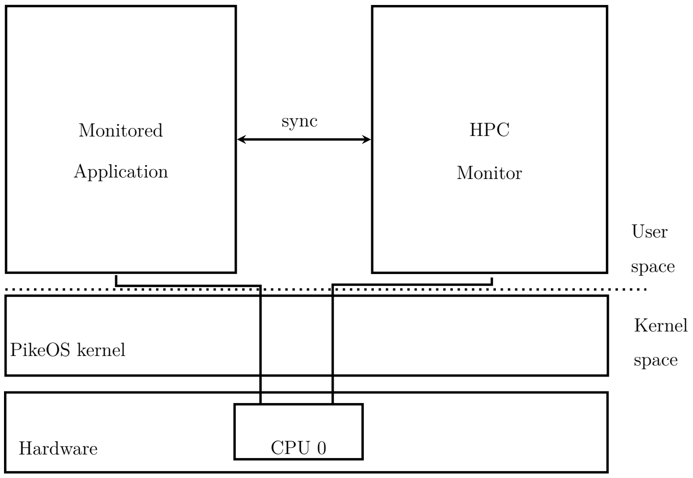


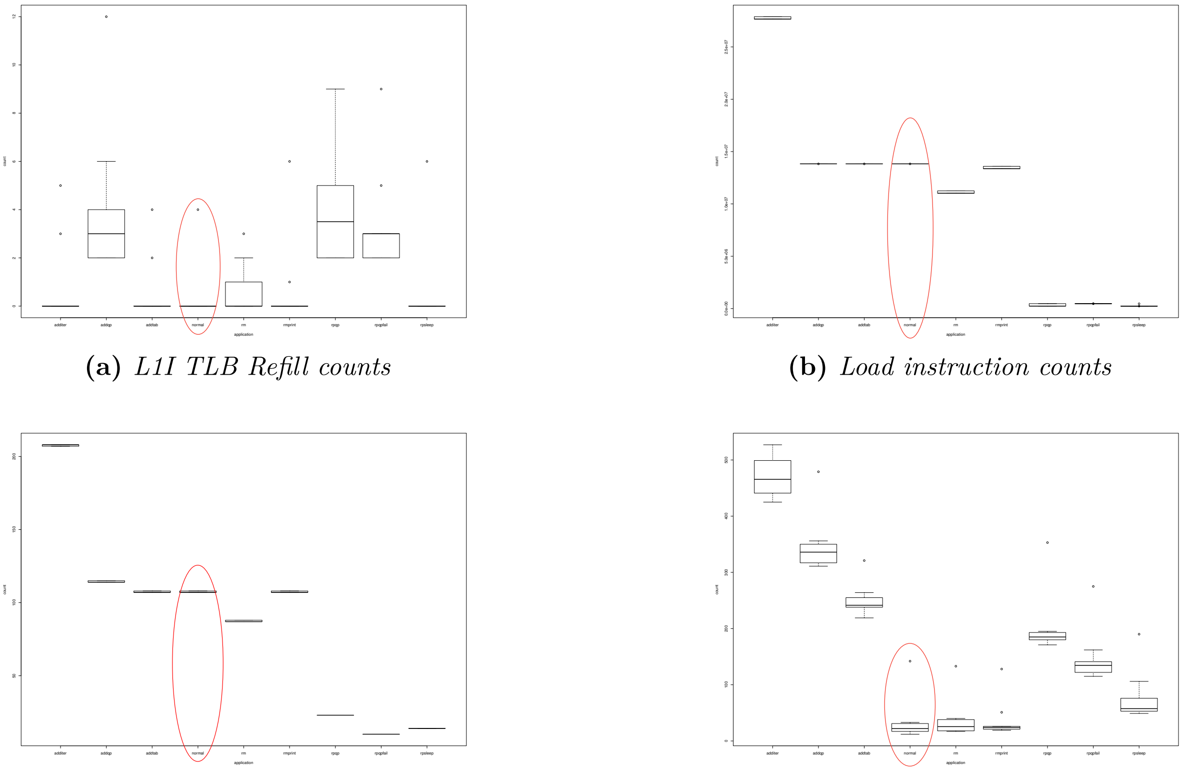
**Figure A.1:** *Offline hardware performance counters monitoring framework*

are more similar to the normal application. Some events allow further separation of this deviation subset. First, exception related events (on Cortex-A53 processor, 7 events) spot more precisely the deviations inducing system calls. Figure A.2c highlights the differentiation between normal and addqp deviation. Refill and write back operations on L1 Data cache distinguish well modifications on the data-flow (Figure A.2d); in total 3 events show different distributions for these cases.

**Table A.1:** *Hardware performance counter events presenting similar results on the test case*

| | |
|---|---|
| **Access** | L1 Instruction, L1 Data caches<br>Memory<br>Bus requests (load/store) |
| **Program** | Load, Store instructions<br>All instructions<br>Branches (conditional, immediat, speculatively executed, etc.) |
| **CPU state** | Pipeline interlocks<br>Writes to program counter |
| **Time** | Bus cycles, CPU cycles |

Figure A.2a provides an example of unsuitable event to trace; L1 Instruction TLB refill events are indeed rare and their occurrence varies significantly on a simple test set. [4] already spotted TLB events fluctuation, and thus they are not adapted for application profiling. On the test case, the lower (below 100 occurrences) the counter value is, the higher variability is. Hence, we do not trace these events, since their value is close to 0. Some events's values depend on each other. For example, counting memory accesses is possible on the cortex-A53 processor with event MEM_ACCESS; we can alternatively



**(a)** *L1I TLB Refill counts*

**(b)** *Load instruction counts*

**(c)** *Exception taken counts*

**(d)** *L1D Cache Refill counts*

**Figure A.2:** *Box-plots of events counts for the set of monitored hardware performance counters on the test set*

trace all memory read and write operations and add them. The table A.2 lists all events to remove from the tracing selection.

**Table A.2:** *Unsuitable hardware performance counter events for application profiling*

| Reason | Total | Filtered HPC events |
|---|---|---|
| unstable | 12 | TLB refill, L2D cache write back/refill, cache maintenance management, interrupts |
| null | 11 | error events, other specific events non-executed in the hello test set (e.g. floating point operations) |
| Software defined | 2 | HPC overflow, software defined event |
| redundant | 3 | memory access, bus accesses, exception return |

## A.3 Final Hardware Performance Counters Selection

From the results we filtered 28events. As the hardware allows at most 6 events to be simultaneously monitored, we then need to select 6 events from the subset defined in section A.2. We choose the counters manually, to maximize the information level on the system hardware state coverage. From state of the art solutions (table III.8) and according to our measurements, the final combination is given in table A.3. Hence, we first select two cache related events to represent system data-flow with accesses to the memory, including missed accesses. Then, we count instructions and the amount of raised exceptions to estimate how much OS services are solicited. Finally, we add information on the CPU branch predictor, to monitor potential misuse of this hardware, and detect suspicious execution flow patterns. We additionally trace CPU cycles for normalizing measurement per time unit.

**Table A.3:** *Hardware performance counters selection*

| Type | HPC | Description |
|---|---|---|
| Data-Flow | L1 Data Cache accesses<br>L2 Data Cache accesses | memory accesses (load/store)<br>and missed accesses |
| Control-Flow | Instructions<br>Exceptions<br>Mispredicted branches<br>Indirect branches spec. executed | amount of instructions<br>privileged operation requests<br>branch predictor trace<br>branch predictor trace |

# Bibliography

[1] F. A. T. Abad, J. V. D. Woude, Y. Lu, S. Bak, M. Caccamo, L. Sha, R. Mancuso, and S. Mohan. On-chip control flow integrity check for real time embedded systems. In *2013 IEEE 1st International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pages 26–31, 2013.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems. `https://www.tensorflow.org/`, 2015. Software available from tensorflow.org.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity: Principles, implementations, and applications. In *CCS*. ACM, 2005.

[4] M. F. B. Abbas, S. P. Kadiyala, A. Prakash, T. Srikanthan, and Y. L. Aung. Hardware performance counters based runtime anomaly detection using SVM. In *2017 TRON Symposium*, 2017.

[5] A. S. Abed, T. C. Clancy, and D. S. Levy. Applying bag of system calls for anomalous behavior detection of applications in linux containers. In *2015 IEEE Globecom Workshops*, 2015.

[6] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik. C-FLAT: Control-flow attestation for embedded systems software. In *CCS*. ACM, 2016.

[7] K. Agrawal, S. Baruah, and A. Burns. The safe and effective use of learning-enabled components in safety-critical systems. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[8] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis. An empirical survey-based study into industry practice in real-time systems. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 3–11, 2020.

[9] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[10] J. Alves-Foss, W. S. Harrison, P. Oman, and C. Taylor. The MILS architecture for high-assurance embedded systems. In *International Journal of Embedded Systems*, 2005.

[11] M. Anandapriya and B. Lakshmanan. Anomaly based host intrusion detection system using semantic based system call patterns. In *2015 IEEE 9th International Conference on Intelligent Systems and Control*, 2015.

[12] ARINC Specification 653P1-2, Avionics Application Software Standard Interface Part 1 -Required Services. `http://www.arinc.com`, 2005.

[13] ARM. ARMv8-A Architecture Reference Manual. `https://static.docs.arm.com/ddi0487/a/DDI0487A_j_armv8_arm.pdf`, 2017.

[14] Arm Holdings. Coresight program flow trace architecture specification. `https://developer.arm.com/documentation/ihi0035/b/`.

[15] Arm Holdings. Coresight trace memory controller technical reference manual. `https://developer.arm.com/documentation/ddi0461/`.

[16] Arm Holdings. *ARM CoreSight SoC-400 Technical Reference Manual*, June 2016.

[17] AUTOSAR. Requirements on health monitoring. `https://www.autosar.org/fileadmin/user_upload/standards/foundation/1-3/AUTOSAR_RS_HealthMonitoring.pdf`.

[18] AUTOSAR. Specification of intrusion detection system protocol. `https://www.autosar.org/fileadmin/user_upload/standards/foundation/20-11/AUTOSAR_PRS_IntrusionDetectionSystem.pdf`, 2020.

[19] AUTOSAR: Automotive open system architecture. `https://www.autosar.org/`, 2021.

[20] I. Avdagic and K. Hajdarevic. Survey on machine learning algorithms as cloud service for cidps. In *2017 25th Telecommunication Forum (TELFOR)*, pages 1–4, 2017.

[21] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. In *IEEE Transactions on Dependable and Secure Computing*, pages 11–33, 2004.

[22] S. Axelsson. Intrusion detection systems: A survey and taxonomy. Technical report, Department of Computer Engineering Chalmers University of Technology Göteborg, Sweden, 2000.

[23] K. Basu, P. Krishnamurthy, F. Khorrami, and R. Karri. A theoretical study of hardware performance counters-based malware detection. In *IEEE Transactions on Information Forensics and Security*, 2020.

[24] N. Bellec, S. Rokicki, and I. Puaut. Attack detection through monitoring of timing deviations in embedded real-time systems. In *32nd Euromicro Conference on Real-Time Systems*, 2020.

[25] J.-P. Blanquart, J.-M. Astruc, P. Baufreton, J.-L. Boulanger, H. Delseny, J. Gassino, G. Ladier, E. Ledinot, M. Leeman, J. Machrouh, et al. Criticality categories across safety standards in different domains. *Embedded Real Time Software and Systems*, 2012.

[26] R. A. Bridges, T. R. Glass-Vanderlan, M. D. Iannacone, M. S. Vincent, and Q. G. Chen. A survey of intrusion detection systems leveraging host data. In *ACM Computing Surveys*. ACM, 2020.

[27] A. Burns and R. I. Davis. Mixed Criticality Systems - A Review. In *ACM Transactions on Embedded Computing Systems*, volume 50, page 81, 2017.

[28] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1), 2017.

[29] N. Burow, X. Zhang, and M. Payer. SoK: Shining light on shadow stacks. In *S&P*. IEEE, 2019.

[30] Capstone. Capstone webpage. `https://www.capstone-engine.org/`, 2021.

[31] N. Carreon, A. Gilbreath, and R. Lysecky. Window-Based Statistical Analysis Of Timing Subcomponents For Efficient Detection Of Malware In Life-Critical Systems. In *2019 Spring Simulation Conference (SpringSim)*, pages 1–12, 2019.

[32] R. Chalapathy and S. Chawla. Deep learning for anomaly detection: A survey. *ArXiv*, abs/1901.03407, 2019.

[33] C.-Y. Chen, S. Mohan, R. Pellizzoni, R. B. Bobba, and N. Kiyavash. A novel side-channel in real-time schedulers. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2019.

[34] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *14th conference on USENIX Security Symposium*, 2005.

[35] M. Cinque, D. Cotroneo, and A. Pecchia. Challenges and directions in security information and event management (siem). In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 95–99, 2018.

[36] R. D. Clercq and I. Verbauwhede. A survey of hardware-based control flow integrity (cfi). unpublished, 2017.

[37] Common vulnerabilities and exposures. https://cve.mitre.org/index.html.

[38] G. Creech and J. Hu. Generation of a new IDS test dataset: Time to retire the KDD collection. In *IEEE Wireless Communications and Networking Conference*, 2013.

[39] G. Creech and J. Hu. A semantic approach to host-based intrusion detection systems using contiguous and discontiguous system call patterns. In *IEEE Transactions on Computers*, 2014.

[40] P. Cronin and C. Yang. Lowering the barrier to online malware detection through low frequency sampling of HPCs. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust*, 2018.

[41] A. Damien. *Sécurité par analyse comportementale de fonctions embarquées sur plateformes avioniques modulaires intégrées.* PhD thesis, INSA Toulouse, 2020.

[42] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *ASIACCS 2015 - Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 555–566, 2015.

[43] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose. Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 20–38, 2019.

[44] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin. HAFIX: Hardware-assisted flow integrity extension. In *DAC*. ACM, 2015.

[45] L. Davi, P. Koeberl, and A.-R. Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *DAC*. ACM, 2014.

[46] R. Davis, S. Altmeyer, and A. Burns. Mixed criticality systems with varying context switch costs. In *24th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2018.

[47] D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, SE-13(2):222–232, 1987.

[48] P. Denzler, J. Ruh, M. Kadar, C. Avasalcai, and W. Kastner. Towards consolidating industrial use cases on a common fog computing platform. In *25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2020.

[49] Development, security, and operations (devsecops) webpage. https://www.devsecops.org/, 2021.

[50] S. Dominic Eschweiler. Security Target PikeOS Separation Kernel v4.2.2. `https://www.atsec.de/wp-content/uploads/2019/01/1041b_pdf.pdf`, 2018.

[51] J. J. Dongarra. The linpack benchmark: An explanation. In *Supercomputing*. Springer Berlin Heidelberg, 1988.

[52] B. Dutertre and V. Stavridou. A model of noninterference for integrating mixed-criticality software components. In *Dependable Computing for Critical Applications 7*, pages 301–316, 1999.

[53] Elinos embedded linux webpage. `https://www.sysgo.com/products/pikeos-hypervisor/`.

[54] M. Elrawy, A. Faisal, I. Awad, and H. F. A. Hamed. Intrusion detection systems for iot-based smart environments: A survey. In *Journal of Cloud Computing*, 2018.

[55] Euro-MILS Project. Whitepaper: Mils architecture. `http://euromils.eu/downloads/2014-EURO-MILS-MILS-Architecture-white-paper.pdf`, 2014.

[56] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener. TACLeBench: A benchmark collection to support worst-case execution time research. In M. Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, OpenAccess Series in Informatics (OASIcs), 2016.

[57] J. Fellmuth, P. Herber, T. F. Pfeffer, and S. Glesner. Securing Real-Time Cyber-Physical Systems Using WCET-Aware Artificial Diversity. In *2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, pages 454–461, 2017.

[58] D. Fiser and W. G. Sanchez. "Detecting attacks that exploit Meltdown and Spectre with performance counters". `https://blog.trendmicro.com/trendlabs-security-intelligence/detecting-attacks-that-exploit-meltdown-and-spectre-with-performance-counters/`, 2018.

[59] X. Ge, W. Cui, and T. Jaeger. GRIFFIN: Guarding control flows using Intel processor trace. In *22nd International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017.

[60] I. Georgiev and I. Georgiev. Some Analysis of the Timing Parameters in Real-time Embedded Systems. In *2020 International Conference on Information Technologies (InfoTech)*, pages 1–4, 2020.

[61] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin. PT-CFI: Transparent backward-edge control flow violation detection using Intel processor trace. In *CODASPY*. ACM, 2017.

[62] W. Haider, J. Hu, and M. Xie. Towards reliable data feature retrieval and decision engine in host-based anomaly detection systems. In *2015 IEEE 10th Conference on Industrial Electronics and Applications*, 2015.

[63] M. Hamad, M. Nolte, and V. Prevelakis. Towards comprehensive threat modeling for vehicles. In *1st Workshop on Security and Dependability of Critical Embedded Real-Time Systems*, page 6, 2016.

[64] S. Han, M. Xie, H.-H. Chen, and Y. Ling. Intrusion detection in cyber-physical systems: Techniques and challenges. In *Systems journal*. IEEE, 2014.

[65] X. Hao, M. Lv, J. Zheng, Z. Zhang, and W. Yi. Integrating Cyber-Attack Defense Techniques into Real-Time Cyber-Physical Systems. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 237–245, 2019.

[66] M. Hasan, S. Mohan, R. B. Bobba, and R. Pellizzoni. Exploring opportunistic execution for integrating security into legacy hard real-time systems. In *IEEE Real-Time Systems Symposium*, 2016.

[67] M. Hasan, S. Mohan, R. Pellizzoni, and R. B. Bobba. Contego: An adaptive framework for integrating security tasks in real-time systems. In *29th Euromicro Conference on Real-Time Systems*, 2017.

[68] S. Heath. *Embedded systems design*. Newnes, 2002.

[69] M. Hill and T. Lake. Non-interference analysis for mixed criticality code in avionics systems. In *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*, pages 257–260, 2000.

[70] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee. Enforcing unique code target property for control-flow integrity. In *CCS*. ACM, 2018.

[71] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy*, 2016.

[72] J. Hu. Afda-ld dataset webpage. `https://www.unsw.adfa.edu.au/unsw-canberra-cyber/cybersecurity/ADFA-IDS-Datasets/`, 2013.

[73] J. Hu, X. Yu, D. Qiu, and H. Chen. A simple and efficient hidden Markov model scheme for host-based anomaly intrusion detection. *IEEE Network*, 2009.

[74] Intel processor tracing webpage. `https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html`.

[75] ISO. ISO 26262 Standard: Road Vehicles - Functional Safety. `https://www.iso.org/standards.html`, 2021.

[76] S. Jose, D. Malathi, B. Reddy, and D. Jayaseeli. A survey on anomaly based host intrusion detection system. In *Journal of Physics: Conference Series*, 2018.

[77] Juno arm development platform webpage. `https://developer.arm.com/tools-and-software/development-boards/juno-development-board`.

[78] M. Kadar, G. Fohler, P. Gorski, and D. Kuzhiyelil. A survey of host intrusion detection for embedded mixed-criticality systems. submitted, 2021.

[79] M. Kadar, G. Fohler, D. Kuzhiyelil, and P. Gorski. Safety-aware integration of hardware-assisted program tracing in mixed-criticality systemsfor security monitoring. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2021.

[80] M. Kadar, S. Tverdyshev, and G. Fohler. System calls instrumentation for intrusion detection in embedded mixed-criticality systems. In *4th International Workshop on Security and Dependability of Critical Embedded Real-Time Systems*, 2019.

[81] M. Kadar, S. Tverdyshev, and G. Fohler. Towards host intrusion detection for embedded industrial systems. In *50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*, 2020.

[82] O. Kath, R. Schreiner, and J. Favaro. Safety, security, and software reuse: A model-based approach. In *RESAFE*. Springer, 2009.

[83] K. Kavanagh, T. Bussa, and J. Collins. Gartner, Magic Quadrant for Security Information and Event Management, 2021.

[84] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman. Survey of intrusion detection systems: Techniques, datasets and challenges. In *Cybersecurity*, 2019.

[85] G. Kim, H. Yi, J. Lee, Y. Paek, and S. Yoon. LSTM-based system-call language modeling and robust ensemble method for designing host-based intrusion detection systems. *arXiv:1611.01726 [cs]*, 2016.

[86] S. Kim and K.-J. Park. A survey on machine-learning based security design for cyber-physical systems. *Applied Sciences*, 11(12), 2021.

[87] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy*, 2019.

[88] Koucham, T. Rachidi, and N. Assem. Host intrusion detection using system call argument-based clustering combined with bayesian classification. In *2015 SAI Intelligent Systems Conference (IntelliSys)*, pages 1010–1016, 2015.

[89] P. Krishnamurthy, R. Karri, and F. Khorrami. Anomaly detection in real-time multi-threaded processes using hardware performance counters. In *IEEE Transactions on Information Forensics and Security*, 2020.

[90] K. Krüger, M. Völp, and G. Fohler. Vulnerability analysis and mitigation of directed timing inference based attacks on time-triggered systems. In *30th Euromicro Conference on Real-Time Systems*, 2018.

[91] S. Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis dissertation, Department of Computer Sciences, Purdue University, 1995.

[92] D. Kuzhiyelil, P. Zieris, M. Kadar, S. Tverdyshev, and G. Fohler. Towards transparent control-flow integrity in safety-critical systems. In *23rd Information Security Conference*, 2020.

[93] J. C. Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *15th International Symposium on Fault-Tolerant Computing (FTSC-15)*, pages 2–11, 1985.

[94] M. Laureano, C. Maziero, and E. Jamhour. Intrusion detection in virtual machine environments. In *30th Euromicro Conference*, pages 520–525, 2004.

[95] Y. Lee, I. Heo, D. Hwang, K. Kim, and Y. Paek. Towards a practical solution to detect code reuse attacks on ARM mobile devices. In *HASP*. ACM, 2015.

[96] Y. Lee, J. Lee, I. Heo, D. Hwang, and Y. Paek. Using CoreSight PTM to integrate CRA monitoring IPs in an ARM-based SoC. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 22(3), 2017.

[97] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Unknown Host Publication Title*, pages 261–270. IEEE, 1987.

[98] H.-J. Liao, C.-H. R. Lin, Y.-C. Lin, and K.-Y. Tung. Intrusion detection system: A comprehensive review. In *Journal of Network and Computer Applications*, 2013.

[99] Linaro. OpenCSD - An open source CoreSight(tm) Trace Decode library. `https://github.com/Linaro/OpenCSD`, 2021.

[100] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium*, 2018.

[101] A. Liu, X. Jiang, J. Jin, F. Mao, and J. X. Chen. Enhancing system-called-based intrusion detection with protocol context. In *Fifth International Conference on Emerging Security Information, Systems and Technologies*, 2011.

[102] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[103] M. Liu, Z. Xue, X. Xu, C. Zhong, and J. Chen. Host-based intrusion detection system with system calls: Review and future trends. In *ACM Computing Surveys*. ACM, 2019.

[104] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan. Transparent and efficient CFI enforcement with Intel processor trace. In *HPCA*. IEEE, 2017.

[105] F. Maggi, M. Matteucci, and S. Zanero. Detecting intrusions through system call sequence and argument analysis. In *IEEE Transactions on Dependable and Secure Computing*, pages 381–395, 2010.

[106] R. Mahfouzi, A. Aminifar, S. Samii, M. Payer, P. Eles, and Z. Peng. Butterfly attack: Adversarial manipulation of temporal properties of cyber-physical systems. In *IEEE Real-Time Systems Symposium*, 2019.

[107] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers*, 37(2):160–174, 1988.

[108] S. A. Maske and T. J. Parvat. Advanced anomaly intrusion detection technique for host based system using system call patterns. In *2016 International Conference on Inventive Computation Technologies (ICICT)*, 2016.

[109] A. S. Mateus. Machine learning based anomaly detection for mixed criticality systems. Master's thesis, Chair of Real-Time Systems, TU Kaiserslautern, 2021.

[110] T. Mehmood and H. B. M. Rais. Machine learning algorithms in context of intrusion detection. In *3rd International Conference on Computer and Information Sciences*, 2016.

[111] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. In *International Conference on Learning Representations ser. ICLR '13*, 2013.

[112] R. Mirzazade Farkhani, S. Jafari, S. Arshad, W. Robertson, E. Kirda, and H. Okhravi. On the effectiveness of type-based control flow integrity. In *Annual Computer Security Applications Conference*, 2018.

[113] R. Mitchell and I.-R. Chen. A survey of intrusion detection techniques for cyber-physical systems. In *ACM Computing Surveys*. ACM, 2014.

[114] S. Mohan, M.-K. Yoon, R. Pellizzoni, and R. B. Bobba. Integrating security constraints into fixed priority real-time schedulers. *Real-Time Systems*, 52(5):644–674, 2016.

[115] R. Moskovitch, S. Pluderman, I. Gus, D. Stopel, C. Feher, Y. Parmet, Y. Shahar, and Y. Elovici. Host based intrusion detection using machine learning. In *IEEE Intelligence and Security Informatics*, 2007.

[116] M. Nasri, T. Chantem, G. Bloom, and R. M. Gerdes. On the pitfalls and vulnerabilities of schedule randomization against schedule-based attacks. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2019.

[117] M. Neukirchner, S. Stein, H. Schrom, J. Schlatow, and R. Ernst. Contract-based dynamic task management for mixed-criticality systems. In *IEEE International Symposium on Industrial Embedded Systems*, 2011.

[118] T. Nyman, J.-E. Ekberg, L. Davi, and N. Asokan. CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers. In *RAID*. Springer, 2017.

[119] Opensource Python Machine-Learning. Tensorflow lite framework. `https://www.tensorflow.org/lite`.

[120] Open source hids security (ossec) host intrusion detection system. `https://www.ossec.net/`, 2021.

[121] S. M. Othman, N. T. Alsohybe, F. M. Ba-Alwi, and A. T. Zahary. Survey on intrusion detection system types. In *International Journal of Cyber-Security and Digital Forensics*, 2018.

[122] S. Owicki and L. Lamport. Proving Liveness Properties of Concurrent Programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, 1982.

[123] PaX Team. RIP ROP. `https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf`, 2015.

[124] P. Chen et al. What you see is not what you get! thwarting just-in-time rop with chameleon. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2017.

[125] M. Pendleton and S. Xu. A dataset generator for next generation system call host intrusion detection systems. In *MILCOM 2017 - 2017 IEEE Military Communications Conference (MILCOM)*, 2017.

[126] D. V. Pham, A. Syed, A. Mohammad, and M. N. Halgamuge. Threat analysis of portable hack tools from usb storage devices and protection solutions. In *2010 International Conference on Information and Emerging Technologies*, pages 1–5, 2010.

[127] L. Pike, P. Hickey, T. Elliott, E. Mertens, and A. Tomb. Trackos: A security-aware real-time operating system. In *International Conference on Runtime Verification*, 2017.

[128] Pikeos hypervisor webpage. `https://www.sysgo.com/products/pikeos-hypervisor/`, 2021.

[129] P. Pop, B. Zarrin, M. Barzegaran, S. Schulte, S. Punnekkat, J. Ruh, and W. Steiner. The fora fog computing platform for industrial iot. `https://www.sciencedirect.com/science/article/pii/S0306437921000053`, 2021.

[130] J.-J. Quisquater and D. Samyde. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security*, pages 200–210, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[131] A. V. Raja, J. Lee, and D. Gao. On return oriented programming threats in android runtime. In *Conference on Privacy, Security and Trust*, 2017.

[132] ARM Cortex-A53 MPCore Processor Technical Reference Manual. `https://developer.arm.com/docs/ddi0500/g`, 2018.

[133] J. E. Rico, M. Bañón, A. Ortega, R. Hametner, H. Blasum, and M. Hager. Compositional security certification methodology. Zenodo, 2018.

[134] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. In *CM Transactions on Information and System Security - TISSEC*, volume 15, pages 1–34, 2012.

[135] I. RTCA. Software considerations in airborne systems and equipment certification. `www.rtca.org`, 2011.

[136] E. M. Rudd, A. Rozsa, M. Günther, and T. E. Boult. A survey of stealth malware attacks, mitigation measures, and steps toward autonomous open world solutions. In *IEEE Communications Surveys Tutorials*. IEEE, 2017.

[137] J. Rushby. The design and verification of secure systems. In *Eighth ACM Symposium on Operating System Principles*, 1981.

[138] S. I. T. C. (SAE-ITC). Arinc specification 653p1-4 avionics application software standard interface part 1 -required services. `http://www.arinc.com`, 2015.

[139] U. A. Sandhu, S. Haider, S. Naseer, and O. U. Ateeb. A survey of intrusion detection & prevention techniques. In *International Conference on Information Communication and Management*, 2011.

[140] S. Sarwar, H. Marco-Gisbert, I. Ripoll, and M. Birch. Control-flow integrity: Attacks and protections. In *Applied Sciences*, 2019.

[141] T. Schulz, C. Gries, F. Golatowski, and D. Timmermann. Strategy for security certification of high assurance industrial automation and control systems. In *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*, pages 1–4, 2018.

[142] W. Shi, H. Zhou, J. Yuan, and B. Liang. DCFI-Checker: Checking kernel dynamic control flow integrity with performance monitoring counter. *China Communications*, 2014.

[143] A. Sinnhofer, W. Raschke, C. Steger, and C. Kreiner. Evaluation paradigm selection according to common criteria for an incremental product development. In *MILS Workshop 2015*, pages 1–5, 2015.

[144] Spunk Advanced Security analytics at scale. `https://www.splunk.com/en_us/cyber-security/siem.html`, 2021.

[145] B. Sprunt, J. P. Lehoczky, and L. Sha. Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm. In *Real-Time Systems Symposium*, 1988.

[146] D. Stojcsics, D. Boursinos, N. Mahadevan, X. Koutsoukos, and G. Karsai. Fault-adaptive autonomy in systems with learning-enabled components. *Sensors*, 21(18):6089, 2021.

[147] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. In *IEEE Transactions on Computers*, 1995.

[148] B. Subba, S. Biswas, and S. Karmakar. Host based intrusion detection system using frequency analysis of n-gram terms. In *2017 IEEE Region 10 Conference*, 2017.

[149] X. Sun, H. Khedr, and Y. Shoukry. Formal verification of neural network controlled autonomous systems. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, page 147–156, New York, NY, USA, 2019. Association for Computing Machinery.

[150] The LLVM Foundation. The LLVM compiler infrastructure. `llvm.org`, 2021.

[151] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *USENIX Security*. USENIX Association, 2014.

[152] S. Tverdyshev, H. Blasum, B. Langenstein, J. Maebe, B. De Sutter, B. Leconte, B. Triquet, K. Müller, M. Paulitsch, A. Söding-Freiherr von Blomberg, and et al. *MILS Architecture*. Zenodo, 2013.

[153] University of California. 1999 darpa intrusion detection evaluation dataset webpage. `https://www.ll.mit.edu/r-d/datasets/1999-darpa-intrusion-detection-evaluation-dataset`, 1999.

[154] University of California. KDD Cup 1999 Data Webpage. `https://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`, 1999.

[155] V. van der Veen, E. Göktaş, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *S&P*. IEEE, 2016.

[156] R. J. Walls, N. F. Brown, T. L. Baron, C. A. Shue, H. Okhravi, and B. C. Ward. Control-flow integrity for real-time embedded systems. In *Euromicro Conference on Real-Time Systems*, 2019.

[157] X. Wang and R. Karri. Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters. In *2013 50th ACM/EDAC/IEEE Design Automation Conference*, 2013.

[158] X. Wang, C. Konstantinou, M. Maniatakos, R. Karri, S. Lee, P. Robison, P. Stergiou, and S. Kim. Malicious firmware detection with hardware performance counters. *IEEE Transactions on Multi-Scale Computing Systems*, 2016.

[159] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: alternative data models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 1999.

[160] C. B. Watkins and R. Walter. Transitioning from federated avionics architectures to integrated modular avionics. In *2007 IEEE/AIAA 26th Digital Avionics Systems Conference*, pages 2.A.1–1–2.A.1–10, 2007.

[161] J. Wolf, B. Fechner, S. Uhrig, and T. Ungerer. Fine-grained timing and control flow error checking for hard real-time task execution. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, pages 257–266, 2012. ISSN: 2150-3117.

[162] S. Wunderlich, M. Ring, D. Landes, and A. Hotho. Comparison of system call representations for intrusion detection. In *Martínez Álvarez F., Troncoso Lora A., Sáez Muñoz J., Quintián H., Corchado E. (eds) International Joint Conference: 12th International Conference on Computational Intelligence in Security for Information Systems and 10th International Conference on EUropean Transnational Education*, 2019.

[163] M.-K. Yoon, S. Mohan, J. Choi, M. Christodorescu, and L. Sha. Learning Execution Contexts from System Call Distribution for Anomaly Detection in Smart Embedded System. In *International Conference on Internet-of-Things Design and Implementation*, pages 191–196. ACM, 2017.

[164] A. Young. Developing a Safety Element out of Context according to ISO 26262. `https://lorit-consultancy.com/en/2017/11/developing-a-safety-element-out-of-context-according-to-iso-26262/`, 2021.

[165] B. Zhou, A. Gupta, R. Jahanshahi, M. Egele, and A. Joshi. Hardware performance counters can detect malware: Myth or fact? In *Asia Conference on Computer and Communications Security*, 2018.

[166] C. Zimmer, B. Bhat, F. Mueller, and S. Mohan. Time-based intrusion detection in cyber-physical systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, page 109. ACM Press, 2010.

[167] R. Zuech, T. M. Khoshgoftaar, and R. Wald. Intrusion detection and big heterogeneous data: A survey. In *Journal of Big Data*, 2015.

# Acronyms

**API** Application Programming Interface.

**ARINC 653** Avionics Application Standard Software Interface.

**AUTOSAR** AUTomotive Open System ARchitecture.

**CF** Control-Flow.

**CFI** Control-Flow Integrity.

**DF** Data-Flow.

**DFI** Data-Flow Integrity.

**FNR** False-Negative Rate.

**FPR** False-Positive Rate.

**GP** General-Purpose.

**HIDS** Host Intrusion Detection System.

**HPC** Hardware Performance Counters.

**IT** Information Technology.

**LSTM** Long Short Term Memory.

**MCS** Mixed-Criticality System.

**MILS** Multiple Independent Layers of Security.

**ML** Machine Learning.

**Monitoree** Monitored application.

**NLP** Natural Language Processing.

**OS** Operating System.

**RT** Real-Time.

**RTS** Real-Time Systems.

**SC** System Call.

**SK** Separation Kernel.

**SoC** System on Chip.

# Summary
# Integration Methods for Host Intrusion Detection into Embedded Mixed-Criticality Systems

In Mixed-Criticality System (MCS), low-critical applications have a larger attack surface compared to high-critical applications. Even though MCS isolate criticality domains of execution by design, a threat affecting a low-critical service can alter the execution of a high-critical task: e.g by degrading the system availability or user-experience. Such attack on low-critical tasks can even introduce an entry point for propagating the attack further to high-critical tasks. In this context, detecting such threats in applications during runtime is a major issue for MCS security. This thesis tackles the problem of Host Intrusion Detection System (HIDS) deployment into embedded MCS.

### Chapter I – Introduction

In this chapter, we outline our motivation to address the problem of integrating intrusion detection into embedded MCS. We identify the gaps in state-of-the-art intrusion detection solutions and present the contributions of this work. We conclude by a description of the chapters composing this dissertation.

Embedded MCS can support multiple applications corresponding to different criticality levels on a single hardware platform. On one hand, low-critical applications represent attractive and likely entry points for intrusion in the system. On the other hand, the failure of a high-critical software can lead to disastrous consequences. Hence, the early detection of threats in the execution of a monitored application contributes to improve the overall security of the system. Nevertheless, the deployment of intrusion detection into embedded MCS must comply with system criticality constraints. For this reason, the great majority of intrusion detection solutions, which are developed in the context of General-Purpose (GP) computers, cannot directly be ported to embedded MCS platforms.

In this dissertation, we propose a methodology to deploy HIDS into embedded MCS and introduce two main HIDS approaches:

- a heuristics based solution, deploying machine-learning based analysis for anomaly detection.
- a specification based solution, with hardware-assisted CFI monitoring.

### Chapter II – Background

This chapter introduces basic concepts related to HIDS deployment into embedded MCS. It also defines our context of work, the problem statement, and our goals for this thesis.

For this work, we consider industrial embedded MCS, which induce several constraints for the integration of HIDS mechanisms in the system. Industrial embedded MCS, and more particularly the monitored application, are subject to certification. This means that HIDS software must comply with the certification process. In addition, the HIDS mechanisms must be transparent for the monitored program's execution.

The integrity of the execution of a given user-level application to monitor is our security asset, which we propose to protect with the deployment of HIDS. We aim at introducing methods to integrate HIDS mechanisms into an industrial embedded MCS and evaluate the ability of a given solution to protect a monitored application in such system, more particularly in the context of certification.

### Chapter III – State of the Art on Host Intrusion Detection

This chapter provides an overview of state-of-the-art HIDS solutions. It covers HIDS research in GP computers and we discuss the applicability of existing approaches to run into an embedded MCS. We are not aware of any previous extensive literature review, which addresses the problem of HIDS integration into embedded MCS, with a focus on deployment into industrial systems.

We introduce a set of key criteria related to deployment constraints and system performance in terms of system overhead, security, and safety; we leverage these criteria to evaluate and classify the HIDS frameworks discussed in this chapter. The results show a great diversity in intrusion detection approaches and evaluation methods; though, the set of solutions suitable to run into embedded MCS is much smaller than the set of solutions for GP computers.

### Chapter IV – A Methodology for Runtime Anomaly Detection in Embedded Mixed-Criticality Systems

In this chapter, we describe an approach for integrating HIDS into embedded MCS and define a generic system architecture to integrate HIDS mechanisms in an embedded MCS. The system design is based on a MILS [10] architecture; the intrusion detection mechanisms execute in a secure execution environment isolated from the monitored application and all other user-level applications, which we consider by default as non-trusted. We also introduce a set of metrics to evaluate the HIDS solution with respect to security, system schedulability, and performance overhead.

The HIDS approaches presented in the following chapters are based on the generic HIDS framework proposed in this chapter. In addition, we evaluate these approaches using the set of introduced criteria.

### Chapter V – Machine-Learning Based Anomaly Detection Solutions for Embedded Mixed-Criticality Systems

In this chapter, we present Machine Learning (ML) based methods to detect intrusions in the execution of a monitored application. The goal is to limit intrusiveness in monitoring, by tracing low-level system events with system calls at OS level and Hardware Performance Counters (HPC) at hardware level.

We present a HIDS composed of:

- an online safety-aware system call and HPC tracing infrastructure.
- an offline ML engine to detect anomalies in the traces of execution of the monitored application.

Our experimental results show a good detection accuracy on the set of tested use-cases.

Alternatively, we propose a transparent online HIDS based on HPC tracing. We evaluate the feasibility of online ML based detection, leveraging an open-source ML framework, to perform detection locally on the deployment platform. For the first time to our knowledge, we presented an online ML based HIDS integrated to an embedded MCS. On one hand, as for the previous system call and HPC based solution, our experimental results show a good detection accuracy to detect anomalous executions for our set of tested use-cases. On the other hand, our analytical and experimental evaluation highlights the main constraints and pitfalls for the deployment of ML-assisted HIDS into such platforms. We showed that for one ML engine configuration, the detection accuracy varies depending on the monitored application. Additionally, according to our experiments, we cannot derive a correlation between ML model complexity and detection accuracy.

### Chapter VI – A Safety-Aware Control-Flow Integrity Framework for Embedded Mixed-Criticality Systems

In this chapter, we describe a safety-aware method to integrate hardware-assisted CF monitoring into a MCS. We introduce a configurable and predictable control-flow integrity (CFI) monitoring framework. This specification based approach leverages a hardware feature of the processor to transparently trace the monitored application.We also introduce a CF events monitoring service using statistics to be combined with sporadic CFI checking, to address the trade-off issue between performance overhead and security coverage of the monitored application's execution.

For our experiment, we implement the CF based framework on PikeOS industrial real-time OS with ARM CoreSight support. We use a set of programs from TACLeBench benchmark. While limiting the time performance overhead to 10% slowdown of the monitored task, our ADS correctly identified almost all samples, with no false-positives and rare false-negatives.

### Chapter VII – An Adaptive Host Intrusion Detection System Approach for Embedded Mixed-Criticality Systems

In this chapter, we propose an analytical adaptive HIDS approach to combine several monitoring solutions. We define a set of requirements and evaluation considerations to guide future implementations.

On one hand, the goal is to improve the detection of intrusions in the monitored application, combining various ways of detection through diverse system signals to reduce the rate of detection errors. On the other hand, such adaptive HIDS approach can support a configurable trade-off between security and performance impact; i.e. reducing the heavy performance overhead induced by some intrusion detection methods like our safety-aware hardware-assisted CFI monitoring approach. We introduce HIDS parameters to configure the solution and analyze the impact of these parameters on a set of security evaluation criteria.

### Chapter VIII – Conclusion

The chapter concludes this dissertation by comparing our contributions to the state of the art. We introduce several safety-aware HIDS approaches to be deployed in embedded MCS. We base our intrusion detection approaches on heuristics and specification techniques. Both types of approach come with limited intrusiveness on the monitored application to facilitate the integration of HIDS mechanisms into industrial embedded MCS, which can be subject to certification.

We discuss main limitations of our implementations and paths for future HIDS development in the context of industrial embedded MCS. The ability of heuristics based HIDS to detect anomalous application executions varies with the characteristics of the monitored application. Future work could notably focus on the evaluation of representative industrial use-cases and threat scenarios; e.g. monitoring a generic representative application or an exhaustive set of applications. Also, it would be valuable to provide a comprehensive approach to configure a heuristics based detection method in function of application characteristics, to improve detection accuracy of the HIDS solution.

### Appendix A

The appendix describes our process to select HPC for intrusion detection, monitoring a given application. It presents our practical evaluation framework and the experiment results. From these experiment results, we identify suitable hardware events for application profiling and select a set of six HPC events to trace for the experiments in Chapter V.

# Zusammenfassung Integrationsmethoden für Host-Basierte Angriffserkennung in Eingebetteten Mixed-Criticality Systemen

Ein Mixed-Criticality System (MCS) integriert und isoliert Anwendungen unterschiedlicher Kritikalitäten hinsichtlich ihrer Anforderungen im Bezug auf die funktionale sowie informationstechnische Sicherheit. Hierbei weisen Anwendungen mit niedriger Kritikalität eine größere Angriffsfläche als jene mit hoher Kritikalität auf. Im Rahmen der Integration von MCS für eingebettete Systeme und Anwendungen ist dies durch die starken sicherheitstechnischen Rahmenbedingungen sowie die intensivere Nutzung geteilter Ressourcen besonders herausfordernd.

Auch wenn das MCS die Kritikalitätsdomänen der Anwendungen bereits im Entwurf isolieren, kann eine Bedrohung, die einen Dienst mit niedriger Kritikalität betrifft, die Ausführung einer hochkritischen Anwendungen über potentielle Seitenkanäle oder erlaubte Mechanismen verändern: z. B. durch Verschlechterung der Systemverfügbarkeit oder der Benutzbarkeit. Ein Angriff über Anwendungen niedriger Kritikalität kann sogar einen Einstiegspunkt für die Ausbreitung des Angriffs auf hochkritische Anwendungen darstellen. In diesem Zusammenhang ist die Erkennung solcher Bedrohungen und potentieller Angriffe in Anwendungen zur Laufzeit ein essentielles Thema für die MCS-Sicherheit. Die vorliegende Arbeit befasst sich mit Lösungen für die Bereitstellung sowie Integration von Host Intrusion Detection System (HIDS) zur Angriffserkennung in eingebetteten MCS.

### Kapitel I – Einleitung

Dieses Kapitel skizziert die Motivation um die Problematik der Integration von Mechanismen zur Angriffserkennung (Intrusion Detection) in eingebetteten MCS zu adressieren. Es werden die Lücken moderner Lösungen zur Angriffserkennung identifiziert und zudem ein Überblick über die Beiträge dieser Arbeit sowie der einzelnen Kapitel dargestellt.

Ein eingebettetes MCS kann mehrere Anwendungen integrieren und somit unterschiedliche Kritikalitätsstufen auf einer einzigen Hardwareplattform zusammenführen. Hierbei stellen Anwendungen mit niedriger Kritikalität attraktive und wahrscheinliche Einstiegspunkte für das ungewollte Eindringen in das System dar. Sollte darüber hinaus der Ausfall einer hochkritischen Software herbeigeführt werden können, kann das katastrophale Folgen haben. Daher trägt die frühzeitige Erkennung von Bedrohungen und Angriffen bei der Ausführung einer überwachten Anwendung zur Verbesserung der Gesamtsicherheit des Systems bei. Aufgrund der anspruchsvollen sicherheitstechnischen Anforderungen ist der Einsatz von Intrusion Detection in eingebettetem MCS anderen Rahmenbedingungen sowie Abhängigkeiten unterworfen als sie die meisten allgemeinen Lösungen zur Intrusion Detection, welche im Kontext von General-Purpose (GP) Computern entwickelt werden, erfüllen können und daher ist die direkte Portierungen der allgemeinen Lösungen oft nicht möglich. In der vorliegenden Arbeit schlägt die Autorin eine Methodik zum Einsatz von HIDS in eingebetteten MCS und stellt zwei Hauptansätze zur Integration des HIDS vor:

- Eine heuristikbasierte Lösung, die auf maschinellem Lernen basierende Analysen zur Anomalieerkennung ansetzt.
- Eine spezifikationsbasierte Lösung mit hardwareunterstützter Überwachung des Kontrollflusses.

Diese Lösungen werden dann auch im Kontext der Zertifizierbarkeit hinsichtlich der sicherheitstechnischen Anforderungen für den Einsatz industriellen eingebettetem MCS diskutiert.

### Kapitel II – Hintergrund

In diesem Kapitel werden grundlegende Konzepte und Anforderungen im Zusammenhang mit der Bereitstellung von HIDS Lösungen in eingebettetem MCS eingeführt. Es definiert damit den Arbeitskontext, die Problemstellung, und die Ziele für die vorliegende Arbeit.

Im Speziellen werden industrielle eingebettete MCS betrachtet, welche mehrere Anforderungen sowie Einschränkungen für die Integration von HIDS-Lösungen in das System mit sich bringen. Industrielle Embedded MCS und insbesondere die überwachte Anwendung sind zertifizierungspflichtig. Das bedeutet, dass HIDS-Lösungen den Anforderungen und Rahmenbedingungen im Zertifizierungsprozess entsprechen müssen. Außerdem müssen die HIDS-Lösungen für die Ausführung des überwachten Programms transparent sein. Die Integrität der Ausführung einer zu überwachenden Anwendung auf Benutzerebene repräsentiert das Gut (Asset), das durch die Bereitstellung und Operationen der HIDS Lösung geschützt werden soll. Das Ziel ist die Einführung von Methoden zur Integration von HIDS Mechanismen in ein industriell eingebettetes MCS und die Bewertung der Fähigkeit einer gegebenen Lösung, eine überwachte Anwendung in einem solchen System zu schützen und dabei den Rahmen der Zertifizierbarkeit des Systems zu wahren.

### Kapitel III – Stand der Technik zu host-basierten Intrusion Detection Systemen

Dieses Kapitel bietet einen Überblick über den aktuellen Stand der Technik im Bereich der HIDS Lösungen. Es behandelt die Forschungsergebnisse im Kontext der allgemeinen HIDS

Lösungen für General-Purpose Computer und deren Anwendbarkeit bzw. Übertragbarkeit für die Zieldomäne der eingebetteten industriellen MCS.

Dazu werden zuerst eine Reihe von Schlüsselkriterien in Bezug auf Bereitstellung, Integration und Nutzbarkeit, Leistungseffizienz und -verbrauch, sowie Sicherheitsmerkmale eingeführt. Diese Kriterien werden anschließend genutzt, um die in diesem Kapitel besprochenen HIDS-Frameworks zu bewerten und zu klassifizieren. Die Ergebnisse zeigen eine große Vielfalt an Intrusion Detection Lösungen und Integrationsmethoden. Allerdings zeigen sie ebenfalls, dass die Menge an Lösungen, die für die Ausführung in eingebettetem MCS geeignet sind, viel kleiner ist als die Menge an allgemeinen Lösungen für General-Purpose Computer.

### Kapitel IV – Eine Methode zur Erkennung von Laufzeitanomalien in eingebetteten Mixed-Criticality Systemen

In diesem Kapitel werden ein Ansatz zur Integration von HIDS in ein eingebettetes MCS sowie die dazugehörige generische Systemarchitektur zur Integration von HIDS-Mechanismen in ein eingebettetes MCS beschrieben. Das Systemdesign basiert dabei auf einer MILS [10] Architektur. Hierbei arbeiten die Intrusion Detection Mechanismen in einer sicheren Ausführungsumgebung, die sowohl von der überwachten als auch allen anderen Anwendungen auf Benutzerebene isoliert ist, damit die Überwachung von den Komponenten getrennt wird, die standardmäßig als nicht vertrauenswürdig betrachtet werden müssen.

Zudem werden Metriken und Kriterien eingeführt, um die HIDS-Lösung in Bezug auf Sicherheit, Planbarkeit des Systems und Leistungseffizienz zu bewerten. Die in den nachfolgenden Kapiteln vorgestellten HIDS-Ansätze basieren auf dem in diesem Kapitel vorgeschlagenen generischen HIDS-Rahmen. Darüber hinaus bewerten wir diese Ansätze anhand der vorgestellten Kriterien.

### Kapitel V – Machine-Learning-basierte Anomalieerkennung für eingebettete Mixed-Criticality Systeme

Dieses Kapitel stellt die auf maschinellem Lernen (ML) basierenden heuristischen Intrusion Detection Lösungen vor, um so Angriffe auf die Ausführung einer überwachten Anwendung zu erkennen. Dabei besteht das Ziel darin, den Einfluss der Überwachung selbst auf das Anwendungsverhalten zu minimieren, indem lediglich ausgewählte Systemereignisse (abgebildet durch die verfügbaren Hardware Performance Counter) sowie Aufrufe des Betriebssystems in Verbindung mit der überwachten Anwendung erfasst werden (Tracing).

Die erste Lösung resultierend aus diesem Ansatz besteht aus:

- einer Infrastruktur zur laufzeitbasierten Erfassung von Systemereignissen und Betriebssystemaufrufen, welche sich kompatibel zu den Anforderungen an die funktionale Sicherheit integrieren lässt.
- eine Offline-ML-Engine, um Anomalien des Verhaltens in den erfassten Laufzeitereignissen der überwachten Anwendung zu erkennen.

Die hierzu vorgestellten experimentellen Ergebnisse zeigen hierbei eine gute Erkennungsgenauigkeit bei den getesteten Anwendungsfällen.

Darüber hinaus wird eine transparente Online-HIDS Lösung, basierend auf der Reduktion der Anwendungsüberwachung mittels des Performance-Counter-Tracing, vorgestellt und die Machbarkeit einer Online-ML-basierten Laufzeiterkennung von Anomalien unter Nutzung eines Open-Source ML-Frameworks evaluiert. Die Anomalieerkennung erfolgt somit lokal auf dem MCS wo auch die Anwendung integriert ist. Damit wird zum ersten Mal ein Online-ML-basiertes HIDS vorgestellt, das in ein eingebettetes MCS integriert ist. Die hierzu vorgestellten experimentellen Ergebnisse zeigen, wie bei der ersten Lösung, eine gute Erkennungsgenauigkeit für anomale Ausführungen im Rahmen der getesteten Anwendungsfälle. Zudem diskutiert die analytische und experimentelle Bewertung die wichtigsten Einschränkungen und Fallstricke für den Einsatz von ML-unterstütztem HIDS auf eingebetteten MCS Plattformen auf. Es wird gezeigt, dass bei einer ML-Engine-Konfiguration die Erkennungsgenauigkeit je nach überwachter Anwendung variiert. Darüber hinaus konnte gemäß der durchgeführten Experimenten keine Korrelation zwischen der Komplexität des ML-Modells und der Erkennungsgenauigkeit hergeleitet werden.

### Kapitel VI – Lösungen zur sicheren Überwachung der Kontrollflussintegrität für eingebettete Mixed-Criticality Systeme

Dieses Kapitel beschreibt eine Methode zur Integration einer hardwareunterstützten Überwachung, auf der Basis von Hardware Performance Countern, der Kontrollflussintegrität (Control-Flow Integrity - CFI) in ein MCS, wobei die Integration transparent und kompatibel zu den einzuhaltenden Rahmenbedingungen der funktionalen Sicherheit bleibt.

Als Resultat ergibt sich ein konfigurierbares Framework zur Überwachung der Kontrollflussintegrität (CFI). Dieser spezifikationsbasierte Ansatz nutzt über die Hardware Performance Counter entsprechende Hardwaremerkmale des Prozessors, um die überwachte Anwendung transparent zu überwachen und diese Informationen unter voraussagbarem Aufwand zur Verfügung zu stellen. Die Überprüfung auf Anomalien im Verhalten der überwachten Anwendung erfolgt auf der Basis der gemessenen Systemereignisse über einen konfigurierbaren Zeitraum in Kombination mit der Auswertung durch statistische Modelle des erwarteten Verhaltens. Damit bietet die vorliegende Lösung zudem einen Ansatz die nötige Feinabstimmungen im Kompromiss zwischen Leistungseinbußen und Sicherheitsabdeckung der überwachten Anwendung zu adressieren.

Für die experimentelle Evaluierung wurde das Framework auf dem industriellen Echtzeit-Betriebssystem PikeOS in Kombination ARM-basierter Hardware inklusive CoreSight-Unterstützung umgesetzt. Die überwachten Anwendungen sind eine Auswahl von Programmen aus der TACLeBench-Benchmark um einen repräsentativen Querschnitt typischer industrieller Anwendungen bewerten zu können. Während die gemessenen Leistungseinbußen bei der Berechnungszeit unter Einsatz der vorgestellten Lösung zur Überwachung der Kontrollflussintegrität auf 10identifizierte die Lösung fast alle fehlerhaften Ereignisse im Kontrollfluss korrekt (ohne False-Positives und nur seltene False-Negatives).

### Kapitel VII – Ein adaptiver Ansatz für host-basierte Intrusion Detection in eingebetteten Mixed-Criticality Systemen

In diesem Kapitel wird ein Ansatz zur adaptiven host-basierten Intrusion Detection vorgestellt und ausgearbeitet. Übergeordnetes Ziel ist die Kombination mehrerer Überwachungslösungen um die Vorteile der verschiedenen Lösungen aus der vorliegenden Arbeit effizient nutzen zu können. Hierzu werden eine Reihe von Anforderungen und Bewertungsüberlegungen definiert, um zukünftige Implementierungen zu leiten.

Ziel ist es die Erkennung von Angriffen und Anomalien in der überwachten Anwendung adaptiv zu gestalten, indem verschiedene Erkennungsansätze kombiniert werden und wechselseitig aktiv werden können. Die beschriebene Fähigkeit zur Adaption erlaubt es die resultierende Fehlerrate bei der Erkennung sowie den zusätzlichen Aufwand bzw. Verbrauch an Systemressourcen als konfigurierbaren Kompromiss zu gestalten und dabei die Kompatibilität zu den Rahmenbedingungen der funktionalen Sicherheit im MCS zu bewahren. Um die Lösung konfigurierbar zu machen werden verschiedene HIDS Parameter eingeführt und die Auswirkungen dieser hinsichtlich einer Reihe von Sicherheitsbewertungskriterien zu analysiert.

### Kapitel VIII – Fazit

Dieses Kapitel schließt vorliegende Arbeit durch einen Vergleich der vorgestellten Ansätze und Lösungen zum Stand der Technik ab. Die beiden vorgestellten Ansätze (heuristisch und spezifikationsbasiert) sind mit begrenztem Eingriff in die überwachte Anwendung verbunden, um die Integration von HIDS-Mechanismen in industrielle eingebettete MCS zu ermöglichen. Diese industriellen eingebetteten MCS unterliegen üblicherweise einer Zertifizierung und haben damit eine Vielzahl von Rahmenbedingungen hinsichtlich der funktionalen Sicherheit im operativen Einsatz nachweislich zu erfüllen.

Zudem werden die wichtigsten Randbedingungen der Implementierungen und Wege für die zukünftige HIDS-Entwicklung im Kontext von industriellen eingebetteten MCS diskutiert. Die Fähigkeit von heuristikbasierten HIDS anomale Anwendungsausführungen zu erkennen, hängt stark von den Eigenschaften der überwachten Anwendung ab. Zukünftige Arbeiten sollten sich insbesondere auf die Bewertung repräsentativer industrieller Anwendungsfälle und Bedrohungsszenarien konzentrieren. Außerdem wäre es wertvoll, einen umfassenden Ansatz bereitzustellen, um ein auf Heuristiken basierendes Erkennungsverfahren in Abhängigkeit von Anwendungseigenschaften zu konfigurieren, damit die Erkennungsgenauigkeit der HIDS-Lösung spezifisch im Kontext der überwachten Anwendung durch entsprechende Feinabstimmung verbessert werden kann.

### Anhang A

Anhang A beschreibt den Prozess zur Auswahl von verfügbaren Hardware Performance Countern bzw. deren zugrunde liegende Systemereignisse für die Intrusion Detection und Überwachung einer bestimmten Anwendung. Es werden der praktische Bewertungsrahmen und die Versuchsergebnisse vorgestellt. Aus diesen experimentellen Ergebnissen werden dann geeignete Systemereignisse für die Anwendungsprofilerstellung identifiziert und einen Satz von sechs zugehörigen Hardware Performance Countern ausgewählt, die für die Experimente in Kapitel V verfolgt werden.

# Marine Kadar

Safety & Security – Embedded – Real-Time – Intrusion detection

## Work Experience

Since November 2021

SYSGO SAS – Klein-Winternheim, Germany

### Embedded Software Engineer

**Management** and technical execution of an industrial research project to deploy **intrusion detection** into an **avionics system**.

*Project management, application of research work to an industrial use case*

2017 - 2021 (4 years)

SYSGO GmbH – Klein-Winternheim, Germany

### Doctorate researcher in embedded security

- Develop an anomaly detection framework: **machine learning, hardware performance counters**.
- Develop a hardware-assisted control-flow integrity infrastructure: **ARM CoreSight, LLVM compiler**.
- **Publish** scientific papers and present research work to academic conferences and industrial events.
- **Supervision** of internships and Masters thesis.

*Expertise in cybersecurity for embedded mixed-criticality systems, academic research,*

2017 (6 mois)

Thales Services – Grenoble, France

### Embedded security internship

- Implementation of hardware-based security mechanisms using ARM **TrustZone** technology: secure boot, secure storage, and isolation of trusted critical applications.
- **Deploymeny and test** of use-case scenario on an embedded platform.

*Embedded software development*

## Education

2017 – 2021 (4 years)

### External PhD student – Real-Time systems chair of TU Kaiserslautern (Germany)

In the scope of the European project **FORA** - Fog Computing for Robotics and Industrial Automation (European Training Network).

2017

### Diplôme d'Ingénieur Ensimag, with honours (France)

Equivalent to a Master's degree in engineering
**Course : Embedded systems**

## Computer Skills

### Programming languages

- ● C, C++, Python, Bash
- ◉ Assembly, TensorFlow, Matlab, Scilab, R, VHDL

### Software

| | |
|---|---|
| OS | Linux, PikeOS, FreeRTOS |
| Compiler | GCC, LLVM |

### Hardware architectures

- ● ARMv8/v7-A – CoreSight, TrustZone, hardware performance counters
- ◉ Intel (X86), PowerPC

## Languages

- ● French, English
- ◉ German

## References

Available upon request.