PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

RAMÃO TIAGO TIBURSKI

# TASK SCHEDULING AND SECURITY FOR EDGE DEVICES IN INTERNET OF THINGS APPLICATIONS

Porto Alegre
2021

PÓS-GRADUAÇÃO - *STRICTO SENSU*


Pontifícia Universidade Católica
do Rio Grande do Sul

**PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL**
**SCHOOL OF TECHNOLOGY**
**COMPUTER SCIENCE GRADUATE PROGRAM**

# TASK SCHEDULING AND SECURITY FOR EDGE DEVICES IN INTERNET OF THINGS APPLICATIONS

**RAMÃO TIAGO TIBURSKI**

Doctoral Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Ph. D. in Computer Science.

Advisor: Prof. Dr. Fabiano Passuelo Hessel

**Porto Alegre**
**2021**

# Ficha Catalográfica

T554t Tiburski, Ramão Tiago

Task Scheduling and Security for Edge Devices in Internet of Things Applications / Ramão Tiago Tiburski. – 2021.
124 p.
Tese (Doutorado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Fabiano Passuelo Hessel.

1. Internet of Things. 2. Edge Computing. 3. Security. 4. Task Assignment and Scheduling. 5. Edge Devices. I. Hessel, Fabiano Passuelo. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

**RAMÃO TIAGO TIBURSKI**

# TASK SCHEDULING AND SECURITY FOR EDGE DEVICES IN INTERNET OF THINGS APPLICATIONS

This Doctoral Thesis has been submitted in partial fulfillment of the requirements for the degree of Ph. D. in Computer Science, of the Computer Science Graduate Program, School of Technology of the Pontifical Catholic University of Rio Grande do Sul

Sanctioned on March 29, 2021.

## COMMITTEE MEMBERS:

Prof. Dr. Tiago Coelho Ferreto (PPGCC/PUCRS)

Prof. Dr. Jorge Luis Victória Barbosa (PPGCA/Unisinos)

Prof. Dr. Leonel Pablo Carvalho Tedesco (PPGPSI/UNISC)

Prof. Dr. Fabiano Passuelo Hessel (PPGCC/PUCRS - Advisor)

Dedico este trabalho à minha família.

## ACKNOWLEDGMENTS

Agradeço primeiramente ao meu pai, Floriano, que muito batalhou para que eu chegasse neste momento e, hoje, infelizmente, me acompanha do céu. Obrigado por tudo, meu pai! Não menos importantes são as outras pessoas da minha família que sempre estiveram ao meu lado, minha mãe, Clarice, minha esposa, Luana, meus filhos, Julia e Lorenzo, e meu irmão, Rimoel. Agredeço a todos pelo apoio incondicional e compreensão durante minha pesquisa. Obrigado por estarem ao meu lado durante todos os períodos importantes da minha vida.

Agradeço ao meu orientador e amigo, Prof. Fabiano Hessel, pelo incentivo e confiança no meu trabalho e pelos valorosos conselhos durante a caminhada do doutorado.

Agradeço aos meus amigos Everton e Willian pelo companherismo e parceria em todas as etapas do curso e também da vida. Um agradecimento especial também aos amigos do GSE e da PUCRS: Amaral, Moratelli, Sergio, Roben e Ramon também foram pessoas importantes nesta jornada. Não poderia deixar de agradecer os amigos de Erebango e colegas do IFSC São Lourenço do Oeste. Obrigado pela parceria!

Thank you Prof. Dr. Jemal Abawajy and Deakin University in Geelong - Australia for receiving me and giving me the opportunity to work with you during my visiting period. Also, special thanks to the Australian Academy of Science for the support in the 2018 Australia-Americas PhD Research Internship Program.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal Nivel Superior – Brasil (CAPES) – Código de Financiamento 001. Agradeço à CAPES pelo apoio financeiro prestado durante a realização do doutorado que originou esta Tese.

# ESCALONAMENTO DE TAREFAS E SEGURANÇA PARA DISPOSITIVOS DE BORDA EM APLICAÇÕES DA INTERNET DAS COISAS


## RESUMO

A evolução da Internet das Coisas, *Internet of Things* (IoT), e a grande quantidade de dados que tem sido trocada entre os dispositivos e a nuvem nos levaram ao paradigma chamado *Edge Computing*, ou computação de borda. Ele permite a migração da computação da nuvem para a borda da rede e pode proporcionar latência baixa e previsível para usuários finais e aplicações, serviços de segurança e de preservação da privacidade, baixo custo de largura de banda, entre outros. Contudo, novos desafios têm surgido nos dispositivos de borda. Primeiro, a descentralização das aplicações da IoT para a borda da rede torna os dispositivos mais visíveis a ataques, principalmente os dispositivos de borda com limitações de recursos que não suportam mecanismos complexos de segurança devido às suas características. Segundo, os dispositivos de borda geralmente constituem aplicações de baixa latência e de computação intensiva da IoT. Dados gerados por esses dispositivos só são úteis se puderem ser processados de acordo com os requisitos de Qualidade de Serviço, *Quality of Service* (QoS), da aplicação. Entretanto, existem vários cenários da Internet das Coisas em que a quantidade de dados ou o tempo de processamento pode ser maior do que o habitual, como durante momentos de pico em aplicações de baixa latência, o que pode resultar na perda de prazos de dados. Nesse sentido, este trabalho apresenta duas principais contribuições. Primeiro, a definição de uma arquitetura de segurança leve para dispositivos de borda com recursos limitados. A arquitetura de segurança é baseada na integração de um hypervisor leve e mecanismos de confiança. Segundo, a definição de um mecanismo de alocação e escalonamento de tarefas para reduzir o número de tarefas que são processadas depois do seu respectivo prazo durante momentos de pico em aplicações de baixa latência da Internet das Coisas.



**Palavras-Chave:** Internet das Coisas, Computação de Borda, Segurança, Alocação e Escalonamento de Tarefas, Dispositivos de Borda.

# TASK SCHEDULING AND SECURITY FOR EDGE DEVICES IN INTERNET OF THINGS APPLICATIONS

## ABSTRACT

The evolution of the Internet of Things (IoT) and the large amount of data that has been exchanged between devices and the Cloud have pushed the horizon to the Edge computing paradigm. It enables the moving of IoT computation from the high-powered central Cloud to the edge of the network. The benefits of Edge computing result from its proximity to data sources and end-users. It allows low and predictable latency for end-users and applications, secure and privacy-preserving services, low bandwidth cost, among others. However, edge computing also brings new challenges to edge devices. First, the decentralization of IoT applications to the edge made the devices more visible to attacks, especially resource-constrained edge devices that do not support complex security mechanisms due to their characteristics. Second, edge devices are usually part of low-latency and compute-intensive applications. Thus, the data generated are only useful if they can be processed following the Quality of Service (QoS) requirements of the application. However, there are several IoT scenarios where the amount of data may be greater or the processing time may take longer than usual, like during peak times, which may result in loss of data deadlines. In this sense, this work presents two main contributions. First, the definition of a lightweight security architecture for resource-constrained edge devices. The security architecture is based on the integration of a lightweight hypervisor and trust mechanisms. Second, the definition of a task assignment and scheduling mechanism to reduce the number of tasks' deadline violations during peak times in low-latency IoT applications.

**Keywords:** Internet of Things, Edge Computing, Security, Task Assignment and Scheduling, Edge Devices.

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF ALGORITHMS

# LIST OF ACRONYMS

ACO – Ant Colony Optimization

API – Application Programming Interface

ARM – Advanced RISC Machine

CIA – Confidentiality, Integrity, Availability

CN – Core Network

COT – Chain of Trust

CP0 – CoProcessor 0

CPU – Central Process Unit

CRC – Cyclic Redundancy Check

DDOS – Distributed Denial of Service

DOS – Denial of Service

DRAM – Dynamic Random Acess Memory

DRTM – Dynamic Root of Trust for Measurement

DTAS – Deadline-aware Task Assignment and Scheduling

ECDSA – Elliptic Curve Digital Signature Algorithm

EDF – Earliest Deadline First

EPC – Exception Program Counter

FCFS – First Come, First Served

FIFO – First In, First Out

FJS – Flexible Job Shop

GCP0 – Guest CP0

GPOS – General Purpose Operating System

GPR – General-Purpose Registers

GSE – Grupo de Sistemas Embarcados

HTTP – Hypertext Transfer Protocol

I/O – Input/Output

IIOT – Industrial Internet of Things

IL – Increase Lifetime

IOT – Internet of Things

IP – Internet Protocol

JS – Job Shop

JSON – JavaScript Object Notation

KVM – Kernel-based Virtual Machine

LAN – Local Area Network

LED – Light Emitting Diode

LLC – Last Level Cache

LOC – Lines Of Code

MCC – Mobile Cloud Computing

MEC – Mobile Edge Computing

MI – Millions of Instructions

MIPS – Millions of Instructions Per Second

MIPS32 – Microprocessor without Interlocked Pipeline Stages

MMU – Memory Management Unit

MCFEDF – Most Critical First with EDF

NIST – National Institute of Standards and Technology

OS – Operating System

OTP – One-Time-Programmable

PA – Physical Address

PAAS – Platform as a Service

PT – Page Table

PTS – Prioritized Task Scheduling

PUCRS – Pontifícia Universidade Católica do Rio Grande do Sul

PUF – Physical Unclonable Function

QOS – Quality of Service

RAM – Random Access Memory

RAN – Radio Access Network

REE – Rich Execution Environment

RISC – Reduced Instruction Set Computer

ROM – Read-Only Memory

ROT – Root of Trust

RR – Round-Robin

RSA – Rivest-Shamir-Adleman

RTOS – Real Time Operating System

SHA – Secure Hash Algorithm

SOC – System-on-a-Chip

SRAM – Static Random Access Memory

TCG – Trusted Computing Group

TCP – Transmission Control Protocol

TEE – Trusted Execution Environment

TLB – Translation Lookaside Buffer

UART – Universal Asynchrounous Receiver/Transmiter

USB – Universal Serial Bus

VA – Virtual Address

VCPU – Virtual CPU

VM – Virtual Machine

VMM – Virtual Machine Monitor

VSOT – Video Surveillance/Object Tracking

WAN – Wide Area Network

WLAN – Wireless Local Area Network

# CONTENTS

# 1. INTRODUCTION

The Internet of Things (IoT) has been recognized as the computing paradigm responsible for integrating heterogeneous applications, systems, and physical devices to provide smart services solutions that facilitate the daily lives of people. Several application scenarios have served as use cases for the IoT, such as e-health, transportation, smart cities, among others [77] [16]. IoT applications are composed of devices that can range from resource-constrained devices with short-range low-rate communication technologies to supercomputers in the cloud [121]. This is what has been called the cloud-to-things continuum [111].

With the explosion of data, devices, and interactions, cloud architecture on its own can not handle the influx of information. While the cloud gives us access to compute, storage, and even connectivity that we can access easily and cost-effectively, these centralized resources can create delays and performance issues for devices and data far from a centralized public cloud or data center source.

Thus, the evolution of the IoT and the significant amount of data that has been exchanged between devices and the Cloud have pushed the horizon to the Edge computing paradigm [174]. It extends the cloud-based infrastructure to the edge of the network by increasing data processing and decision-making on IoT edge devices, allowing more efficient communication with intermediary nodes [111].

The Edge Computing paradigm enables moving the IoT computation from the high-powered central Cloud to the edge of the network [142]. The benefits of Edge Computing result from its proximity to data sources and end-users. It has the potential to address the following challenges: (i) low and predictable latency for end-users and applications; (ii) secure and privacy-preserving services and applications; (iii) long battery life and low bandwidth cost; and (iv) scalability [135].

However, edge computing brings new challenges to the devices [126] [76]. First, the decentralization of IoT applications to the edge made the devices more visible to attacks [7]. While some existing solutions in cloud computing may address many security issues at the edge of the network, edge computing introduces new security concerns due to its distinct characteristics, such as the protection of resource-constrained devices [111]. Second, edge devices are usually part of low-latency and compute-intensive applications. Thus, the data generated are only useful if they can be processed following the Quality of Service (QoS) requirements of the application. However, there are several IoT scenarios where the amount of data may be greater or the processing time may take longer than usual, common during peak times, which may result in loss of data deadlines.

## 1.1 MOTIVATION

Edge Computing makes possible the decentralization of data processing and decreases the dependency of the cloud [62]. However, it introduces new security and data processing challenges to edge devices [7]. In edge computing, devices are more than data sources [126]. They can process data (e.g., filtering, abstraction, aggregation) coming from IoT sensors, making them hot targets for attacks [7]. Once a hacker controls the device, no higher security mechanism can identify such condition, and all system execution and state can be compromised. Depending on the application, a compromised edge device can represent a danger for life and business. For instance, it may involve stealing sensitive information, manipulating information, malfunctioning the device itself, or other devices controlled by it [102]. However, in the last three years, several attacks that compromise human life have been noticed in edge devices [82], such as autonomous cars [166] [167].

Recent software-based solutions have been applied to mitigate security challenges in edge devices. However, some resource-constrained edge devices cannot store a large amount of data or execute a high complexity security algorithm. Thus, security approaches should have a lightweight design and must be extended with security-oriented technologies that promote hardware as the root of trust [117].

Regarding data processing, low-latency IoT applications (e.g., traffic management, autonomous cars, surveillance systems) generate many data, which have to be processed with minimal time latency to be useful to the application [68]. Suppose a specific area monitored by cameras has a few minutes of high movement of people and objects, also known as peak or rush hour. A camera will have more objects/faces/people to detect in that specific time. Thus, the processing time for this data may be longer than at a time of less movement, and, automatically, the data's deadline, kept the same, may not be reached, invalidating its use.

According to [163], to process data in an edge-fog-cloud architecture effectively, three problems must be addressed jointly: (i) offloading decision, deciding which tasks are offloaded for edge/fog/cloud devices, (ii) task assignment, deciding which layer and device is each task assigned to for execution, and (iii) task scheduling, deciding the execution order of the tasks for each device. Existing research surveyed in Wang *et al.* [163] concern only one or two of such problems, leading to sub-optimal solutions for task scheduling.

Alizadeh et al. [6] argue that it is essential to assign and schedule the tasks in the architecture in a way to be executed in a timely fashion and also to make optimal use of available resources. However, an important question is how to determine which tasks should be executed in each architecture layer [50]. The problem under consideration is similar to the flexible job scheduling problem [85] [84], in which a set of tasks are to be executed by

multiple devices given a partial ordering of task execution and the delay associated with each task.

Premises that motivated this thesis are presented next:

- *Increasing of Devices-to-Cloud Communications:* With the constant increase of IoT devices, the amount of data generated and sent to the cloud increases. It can overload the network, expose data to attacks, and delay decisions and notifications for applications.
- *Trending for Edge Computing:* According to investigations [18] [59] [52] [127], the Edge computing paradigm can contribute significantly to increase security, reduce latency, and also be robust to connectivity issues following the next years. Gartner predicts that by 2025, three-quarters of enterprise-generated data will be created and processed at the edge [57].
- *Low-latency Applications:* Current IoT applications, especially in low-latency scenarios, require data to be processed respecting their deadlines to meet individual QoS requirements. Thus, keeping applications in the cloud implies sending the generated data to the cloud, which can delay responses and compromise applications' results.
- *Increasing of Attacks on Devices:* With the migration of computing to the edge, several hackers also migrate attacks from the cloud to the edge. However, most of the edge devices are resource-constrained regarding storage, RAM, and CPU. Thus, they cannot support or implement conventional highly-secured and sophisticated security techniques, impacting the device's security strength.

## 1.2 HYPOTHESES AND RESEARCH QUESTIONS

This Ph.D. thesis aims to investigate two hypotheses: (i) the integration of embedded virtualization and trust mechanisms can provide a lightweight security architecture to improve the security of resource-constrained edge devices, nonetheless, keeping a small memory footprint; and (ii) a deadline-aware task assignment and scheduling mechanism can reduce the number of deadline violations in low-latency IoT applications during peak times.

Hence, the following research questions were established to support the validation of the hypotheses:

1. **Research Question:** *What are the most common security threats that could compromise edge devices, and what requirements should be considered to improve their security?*

2. **Research Question:** *How to define a lightweight security architecture with a high-security level but keeping a small footprint with tens of kilobytes?*
3. **Research Question:** *Where should a task be assigned to have a better chance of being processed meeting its deadline?*
4. **Research Question:** *Which architecture layer should the assignment and scheduling mechanism be deployed to reduce the number of deadline violations?*

## 1.3 OBJECTIVES

The objective of this thesis is two-fold. The first objective is to define a lightweight security architecture for resource-constrained edge devices. The second objective is to define a deadline-aware task assignment and scheduling mechanism that executes on edge devices and can reduce the number of deadline violations in low-latency IoT applications during peak times. To achieve the research goals, the following objectives were defined:

- Studying the existing works for assignment, scheduling, and security in edge computing;
- Definition of a lightweight security architecture for resource-constrained edge devices;
- Definition of a deadline-aware task assignment and scheduling mechanism;
- Evaluation of the security architecture using a real-world resource-constrained edge device;
- Evaluation of the deadline-aware task assignment and scheduling mechanism by simulating its applicability in low-latency IoT applications;
- Documenting and reporting the research results, publishing them in scientific conferences and journal articles.

## 1.4 CONTRIBUTION

This research presents two main contributions: (i) the definition of a lightweight security architecture for resource-constrained edge devices based on lightweight virtualization and trust mechanisms, protecting devices from the hardware until the highest layer of software and (ii) the definition of a deadline-aware mechanism to assign and schedule low-latency IoT applications tasks during peak times, deciding on edge devices. The specific contributions are:

- A review of lightweight virtualization, security for edge devices, and task assignment and scheduling in edge-fog-cloud architectures;
- The evaluation of a lightweight virtualization layer to be used in resource-constrained devices;
- The creation of a taxonomy of security for edge devices;
- The usage of trust mechanisms and lightweight virtualization to provide security for resource-constrained edge devices;
- The validation of the security architecture in a real-world resource-constrained edge device;
- A deadline-aware mechanism that executes on edge devices to decide the best assignment and scheduling for tasks;
- The validation of the deadline-aware mechanism in low-latency IoT applications during peak times.

## 1.5 THESIS OUTLINE

The remainder of this thesis is organized as follows. Chapter 2 presents theoretical references used in this work, such as definitions of IoT, Edge Computing, virtualization, security, and scheduling. Chapter 3 presents the Hellfire hypervisor, the virtualization layer used to define the security architecture. Chapter 4 presents the lightweight security architecture. Chapter 5 presents the deadline-aware task assignment and scheduling mechanism. Finally, Chapter 6 presents the conclusions, author's publications in the last years, and future work.

# 2. THEORETICAL BACKGROUND

This chapter presents definitions regarding this thesis. Section 2.1 presents edge computing concepts, architecture models, and type of edge devices. Section 2.2 presents the lightweight virtualization concepts and a literature review for embedded virtualization. Section 2.3 defines security for resource-constrained edge devices, presenting challenges, attacks, and key requirements. Finally, Section 2.4 defines scheduling and describes an instance of the task assignment and scheduling problem in an edge-fog-cloud architecture.

## 2.1 EDGE COMPUTING: INTEGRATING IOT AND THE CLOUD

The Internet of Things is a computing paradigm that refers to the interaction and communication between billions of devices that produce and exchange data related to real-world objects (i.e., things) [54]. Connected devices can be sensors, actuators, smartphones, computers, buildings and home/work appliances, cars and road infrastructure elements, and any other device or object that can be connected, monitored, or actuated [24]. Beyond Internet-based communication, they can interact with each other and cooperate with neighbor elements and systems to reach common goals, providing smart services solutions to applications [77].

IoT's features, including large-scale of things and network-level heterogeneity, make the development of applications and services a very challenging task [124]. The IoT relies on a set of well-known, established technologies divided into different layers: devices, network, middleware, and applications [16]. However, it must be said that there is no consensus on this issue. Even these layers are sometimes fused, resulting in a three or two-layer model [121]. Moreover, IoT applications generate enormous amounts of data that should be subsequently analyzed to determine reactions to events or extract analytics or statistics [16].

Cloud-oriented systems have been used for years as the bridge to connect heterogeneous IoT devices to higher-level and cloud-based services and applications. However, the cloud computing paradigm applied directly to IoT presents a set of drawbacks regarding latency, bandwidth, and storage because of the huge amount of data that have to be uploaded and processed [121] [68]. Due to these limitations, Fog and Edge computing paradigms have been introduced [25] [135].

Fog and Edge computing are firmly related concepts, but they are not synonyms [147] [26] [112] [111] [101] [106] [79] [70] [21]. According to the OpenFog Reference Architecture [111], Fog computing extends Cloud computing into an intermediate layer close to IoT devices and enables data processing across domains. In contrast, Edge computing

involves the control and management of a standalone endpoint device individually within the Fog domain, typically within close proximity of IoT sensors and actuators [28] [47] [3] [30].

Fog and Edge computing have become key enablers for the future IoT [129]. They promote multi-layer and decentralized systems that facilitate the development and deployment of IoT applications at the edge/fog, which brings data processing closer to IoT devices and can help avoid the bandwidth shortage of the Internet [55].

Literature definitions around Cloud, Fog, and Edge are given next:

- **Cloud**: The National Institute of Standards and Technology (NIST) defines cloud computing as *"a model for enabling ubiquitous and on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction"* [93]. Significant data are collected and uploaded to be processed in cloud devices, where permanent and sufficient processing resources are available [121].

- **Fog**: Bonomi et al. [25] define Fog computing as a *"highly virtualized platform that provides compute, storage, and networking services between end devices and traditional Cloud Computing Data Centers, typically, but not exclusively located at the edge of networks"*. Another concept, the NIST defines fog computing as a *"layered model for enabling ubiquitous access to a shared continuum of scalable computing resources"* [73]. Finally, the OpenFog Consortium [111] defines fog computing as *"an architecture that distributes computation, communication, control, and storage closer to the end-users along the cloud-to-things continuum"*. The fog has many similarities with the edge, which usually leads to confusion or puts them on the same level. The Fog layer is composed of fog devices. The main function is to process, orchestrate, and store data received from the edge. Also, fog devices can send data selectively to the cloud under request or based on specific rules [121]. The fog is usually placed physically close to the edge (i.e., from tens to hundreds of milliseconds). Some examples of fog nodes are small servers and IoT gateways. Even a powerful edge device sometimes may be considered a fog node (e.g., a smart car with tens of sensors/actuators).

- **Edge**: Edge computing refers to allowing computation to be performed at the edge of the network [135]. One of the most fundamental characteristics of the edge is that the processing, networking, and storage are made right in the IoT devices or the sensor nodes, meaning that communication with the fog or cloud is not mandatory [121]. One key difference between fog and edge is that in fog, the processing is carried out in more powerful devices than IoT nodes, such as Internet gateways. According to Portilla et al. [121], the nature of the edge devices' resources allows a secondary division: edge devices and resource-constrained edge devices, being the last ones tiny devices composed of sensors, a low-end microcontroller, among others, also known as the

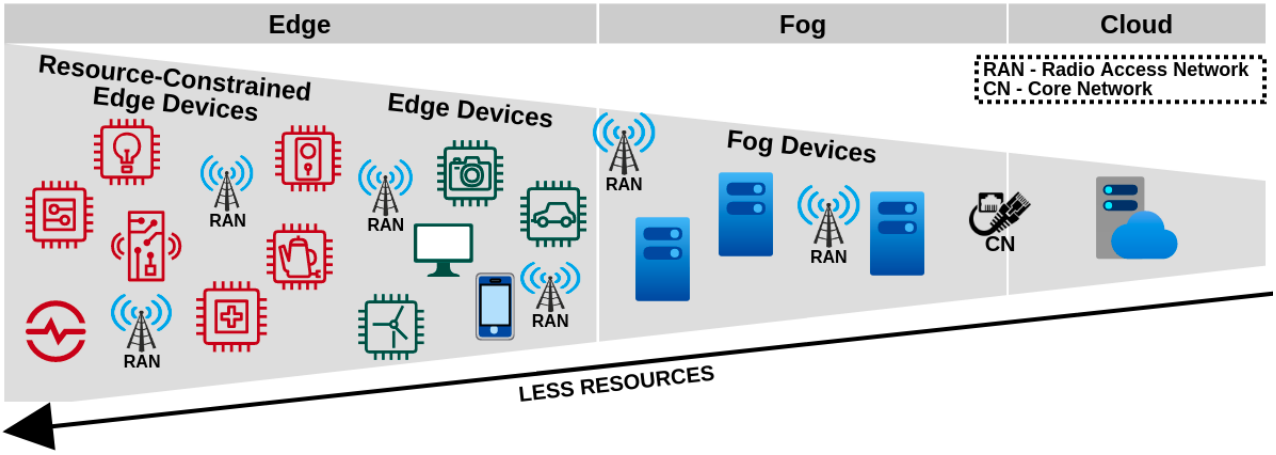



Figure 2.1 – Edge, Fog and Cloud computing.

things. They are very limited in computing and storage capabilities to reduce costs and enlarge their lifetime. Some authors have named this layer as the Mist [122] [20] or the Extreme Edge [46] [121], stating that it pushes processing even further to the network edge, involving the sensor and actuator devices. Some edge devices are IoT platforms (e.g., Raspberry Pi, Arduino), cameras, sensors, actuators, smartphones.

The relationship between Edge, Fog, and Cloud is presented in Figure 2.1. The Cloud works as a management layer, putting almost all the processing at the edge and fog layers. Fog computing focuses its processing efforts at the LAN (i.e., Local Area Network) end of the network [68]. It enables fog devices to process and control data received from multiple edge devices and sends information exactly where it is needed [28]. Fog devices use the Radio Access Network (RAN, e.g., WLAN, cellular networks) to interact with each other and connect to Edge devices [106]. On the other hand, the Core Network (CN) allows interactions between Fog and Cloud, if needed. Edge computing brings data processing closer to or within the data sources [135]. Edge devices can be the simple integration of sensors/actuators with microcontrollers or complex devices such as cars [47]. They use RAN to interact with the Fog layer [106].

### 2.1.1 ARCHITECTURE MODELS

There are several architecture models involving edge, fog, and cloud: Edge-Cloud [158], Fog-Cloud [160], Edge-Fog [148], Edge-Fog-Cloud [97], Mobile Edge Computing (MEC) [129], Mobile Cloud Computing (MCC) [132], among others variations [32]. Since mobility is out of the scope of this work, Figure 2.2 presents the most used architectures considering edge, fog, and cloud.

**Edge-Cloud:** This model maintains data processing and orchestration closer to or within edge devices. The cloud is still responsible for general management and storage.

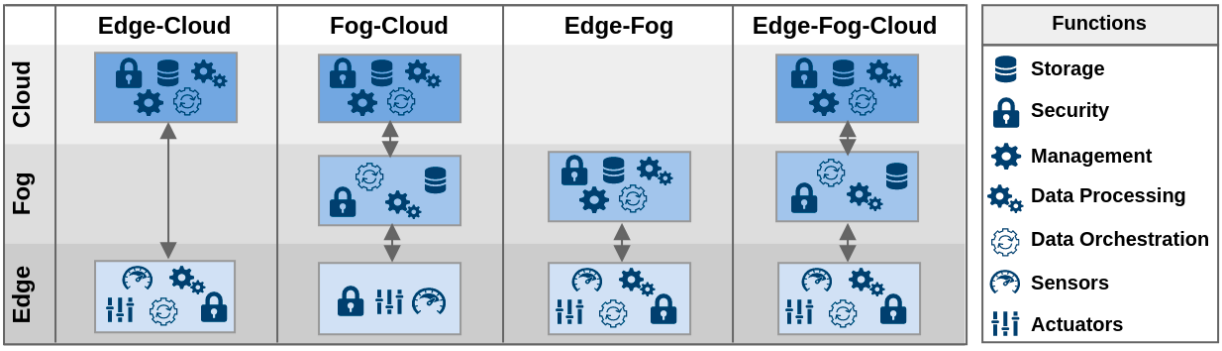



Figure 2.2 – Architecture Models involving Edge, Fog, and Cloud computing.

Also, depending on the application requirements and edge devices capabilities regarding resources, the cloud can process and orchestrate data as the edge [135]. The Edge-Cloud model's main advantage is low-latency to IoT applications at the edge [4]. Some works from literature are: [158] [35] [31] [175] [159].

**Fog-Cloud:** This model instantiates Cloud traditional functions at the Fog layer, which includes storage, data processing and orchestration [28]. Its main advantage is to maintain a computing layer close to data sources devices [25]. Some works from literature are: [160] [23] [106] [145] [53].

**Edge-Fog:** This model avoids the use of cloud computing. All functions are distributed among fog and edge layers. Fog is responsible for management, storage, data processing, and orchestration. The edge can process and orchestrate data as well. The application requirements should define what can be processed at the edge and what should be assigned to the fog. This model has advantages that usually benefit real-time and/or decentralized applications. Some works from literature are: [148] [71] [120] [165] [110].

**Edge-Fog-Cloud:** In this model, the three layers have relevant roles in the architecture model. This model is more suitable for applications that generate many data in short periods. Thus, data can be processed and orchestrated in all layers. A storage function is also an option in the fog. Some works from literature are: [97] [48] [51] [67] [29]

To summarize, the cloud is responsible for general management and storage in most architecture models. Data processing and orchestration are preferred at edge and fog layers, so it is possible to reach the edge/fog computing benefits avoiding communication bottlenecks of wide-area networks [28] [47]. Also, end-to-end security should be ensured for all models, i.e., starting at the edge and going until the cloud.

### 2.1.2 TYPES OF EDGE DEVICES

Edge devices can have different limitations in resources (storage, memory, and CPU). In this work, two terms are used to refer to them: "**edge devices**" to refer to general

devices at the edge layer; and "**resource-constrained edge devices**" to refer to devices starting with a few hundred kilobytes up to some megabytes of storage and memory, typical in IoT applications. Resource-constrained edge devices often have connectivity options, graphic accelerators, and other hardware features that enable more complex applications. Nevertheless, its memory capacity makes challenging the adoption of rich operation systems. For example, the PIC32mz EF family from Microchip has up to 2MB for flash and 512KB RAM with several connectivity options (USB, Ethernet, and Wi-Fi). It has a core MIPS M5150 at 200Mhz with memory management capacity and a cryptography engine. The PicoCore RT1-V1 is another example. It has an ARM Cortex-M7 core with 256MB of flash and 32MB of RAM and connectivity options like USB, Ethernet, and UART.

## 2.2 VIRTUALIZATION

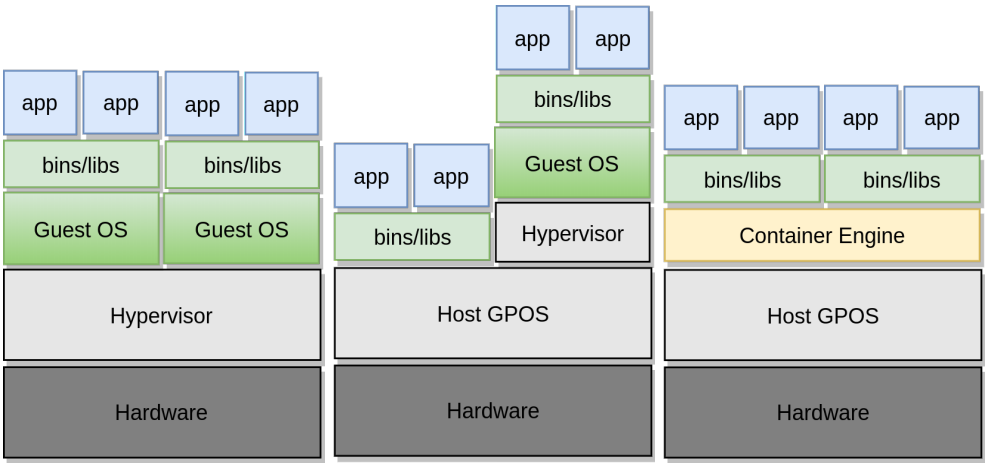
Virtualization means the creation of an environment or a virtual machine (VM) that acts like the real target hardware from the software or user point of view [131] [102] [96]. A VM is implemented as a combination of real hardware and software aiming to execute applications [138]. The virtualization layer is often called a Virtual Machine Monitor (VMM) or simply hypervisor.

Hypervisors are a crucial component of virtualized systems and represent an intermediate software layer between VMs and the hardware [102]. Running in a privileged context, a hypervisor has direct control of the hardware and restricts lateral movement within the system while facilitates high-speed inter-VM communications [130].

Virtualization offers several advantages in terms of technical, environmental, and business solutions. One of them is its ability to support heterogeneous operating-system environments to address conflicting requirements such as high-level APIs, real-time support, or legacy software [115]. Security is another crucial advantage since virtualization can enable a strong spatial separation between the guest OSs [118]. Also, to prevent a complete system failure when an error occurs in one of the subsystems is another advantage. These advantages make the virtualization technology useful also for embedded and real-time systems [102].

### 2.2.1 VIRTUALIZATION FOR EDGE DEVICES

In the last years, virtualization has been the focus of a high amount of industrial and academic research, mainly related to Edge computing [68] [5] [107] [143] [141] [101] [98] [139] [100] [102] [156] [117] [90]. For a long time, the research community believed that hypervisor-based virtualization was an overkill approach for resource-constrained devices




(a) Type-1 Hypervisor. The hypervisor as the first software layer.

(b) Type-2 Hypervisor. The hypervisor running over a GPOS.

(c) Containerization. Dismiss the use of hypervisors.

Figure 2.3 – Comparison between the different virtualization approaches.

due to its inherent overhead [107, 143, 98, 100]. However, the advances in embedded processors that enabled hardware-assisted virtualization and innovative hypervisor software architectures changed this scene. Recent research has shown the benefit of embedded virtualization to meet resource-constrained devices' challenges in edge environments [5, 102, 117]. The essential characteristics for virtualized systems in the context of resource-constrained devices are discussed next:

- **Virtualization Approach:** It can be a type-1 hypervisor, a type-2 one, or a container engine. Type-1 hypervisors (Figure 2.3a) are directly executed on the hardware and are the most adopted approach in server virtualization [19]. Differently, type-2 hypervisors (Figure 2.3b) execute on top of an OS. They are the preferred choice for home and office virtualization since they can run guest OSs side-by-side with user's applications. Containerization (Figure 2.3c) is an operating system-level virtualization method to execute multiple isolated systems (containers) using a single kernel. A user process can check different information about the system, like memory, process trees, files, and directories in a typical OS. OS's containers enforce the process isolation limiting and prioritizing the resources (e.g., CPU, memory, I/O, network, among others) without using virtual machines. A container engine executes on top of a host operating system, responsible for the separation between applications using the OS's features.

- **Underlying GPOS:** It refers to the need for an underlying GPOS (e.g., Linux and Windows). This complex system supports a wide range of applications, like accelerated graphics, artificial intelligence, and graphical user interfaces. Such features make GPOS generally large in footprint and also unpredictable for real-time purposes. A GPOS is designed to perform well on devices with more powerful processing capabilities than we commonly have in resource-constrained edge devices. Although some

GPOS (e.g., Linux) are highly customizable, their minimal footprint is still unacceptable to such devices.

- **Spatial Separation:** It allows the execution of applications inside VMs or even instances of operating systems in separate boxes over the same device [102]. Spatial separation among VMs is provided through a memory management unit (MMU), a hardware block that provides virtual memory abstractions to the system. A hypervisor must keep memory isolation among VMs while the OS (inside the VM) can still maintain isolation between processes. Hardware-assisted virtualization implements the second stage of MMU translation directly. Essentially, the hardware performs the translation without software intervention. The hypervisor still manages its page table, but the software inside the VM can handle the hardware in the same way as in a non-virtualized system.

- **Temporal Separation:** It guarantees the correct distribution of processor's time among VMs according to their execution priorities [102]. Also, system interrupts are a concern not only because of the interference on the correct timing execution but also because it can disrupt the normal operation. Hardware-assisted virtualization can help manage interruptions, allowing them to be redirected to VMs without intervention from the hypervisor. This feature is called interrupt pass-through and it minimizes the overall hypervisor overhead and footprint.

- **Inter-VM communication:** It is a mechanism for communication among VMs. The hypervisor should work as a communication arbiter, copying messages from the sender to the destination application. Also, the hypervisor can check the size, the number of messages and even deny forbidden communication. It can be implemented as paravirtualized services, i.e., using a hypercall API (VM's calls to the hypervisor).

- **Real-time Support:** Proper real-time support is a common feature required by embedded systems. Typically, GPOS focus on performance and present poor real-time results. Otherwise, real-time operating systems (RTOS) can deal with timing constraints, but they do not support software separation. Thereby, separation and real-time capabilities are required to coexist in the same system.

The increasing demand for more complex software stacks makes modularity an essential feature for edge devices. As said before, the containers approach can be used to bring modularity. However, the implicit overhead cost to keep a GPOS limits the reach for resource-constrained edge devices. Otherwise, RTOSs may deliver a certain level of modularity but cannot deal with separation.

Figure 2.4 depicts a lightweight virtualization model that can deal with modularity, security, and computing [102]. In this architecture, the hypervisor is the first layer of software (type-1 virtualization), and it runs in the highest processor's privilege level (supervisor

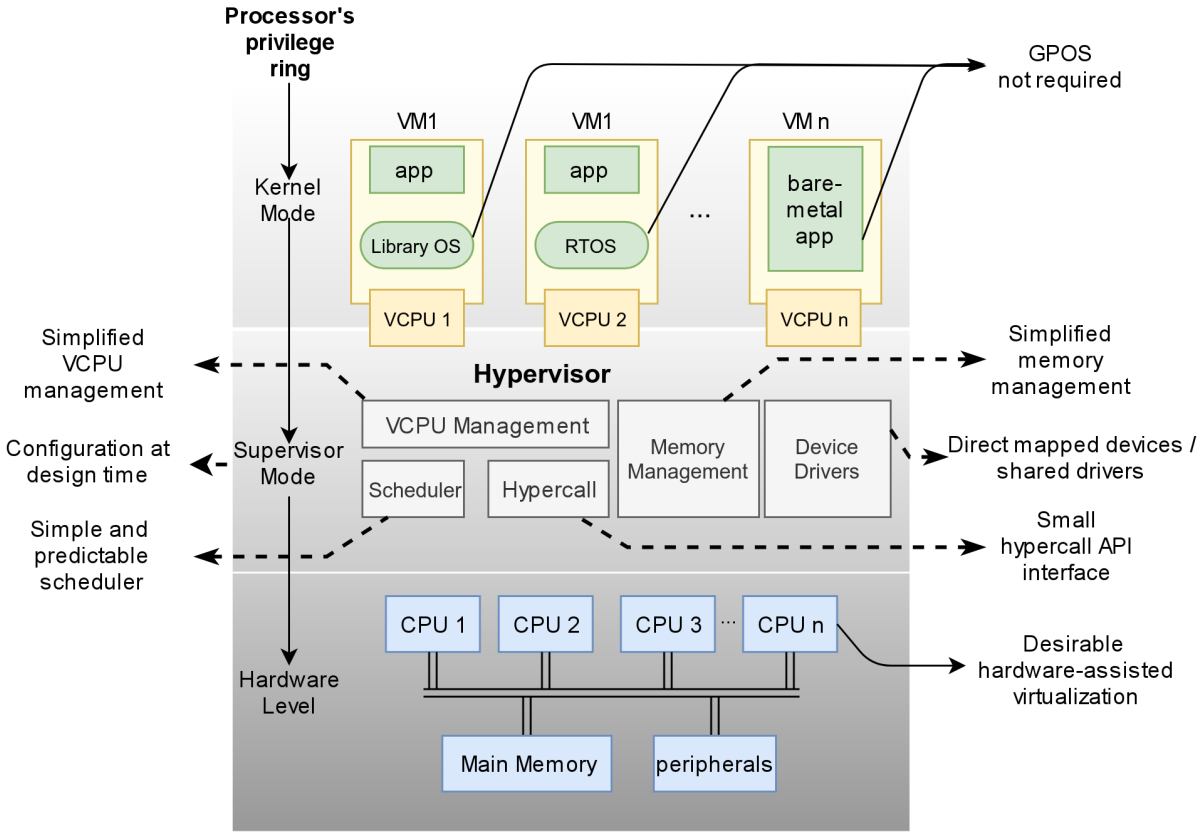



Figure 2.4 – A lightweight hypervisor model. The doted sets indicate the key architecture's characteristics that make possible the building of a lightweight hypervisor.

mode). Hence, it can control all hardware behavior avoiding that guests change the processor or platform configuration. The hypervisor creates the VMs' abstraction by implementing two basic functionalities. First, constructing memory isolation (using the MMU hardware). Second, creating the virtual CPU abstraction (VCPU) (i.e., a data structure that keeps the CPU context during context switches). The next software layer is the guests system. They can be RTOSs, Unikernels, or even bare-metal applications that perform with a lower privilege level (kernel mode). Any unexpected behavior (e.g., access to a not-allowed memory location) traps the hypervisor to execute some programmed action. Thus, the software in a VM views the system as an entirely independent machine allowing the implementation of wholly separated applications and bringing modularity to the system.

While hypervisors for cloud computing require up to tens of megabytes of footprint, lightweight virtualization approaches need tens of kilobytes, keeping virtualization advantages like modularity and security [156]. The simplifications to provide a lightweight virtualization layer to edge devices are presented and discussed next [103] [99] [9] [100]:

- **Simplified memory management**: A lightweight virtualization layer should simplify page tables' implementation, avoiding swapping [137]. The number of VMs is usually known, which can allow the system's partitioning at design time. Thereby, VMs can be allocated contiguously in memory, making the management more straightforward

because only the base address and size are required for its mapping. This scheme saves memory because no page tables are needed. After all, the memory mapping can be directly written to the MMU control registers. This simplification does not affect the memory management implementation at the guest's level because how the virtualization layer manages the memory is transparent to the guest.

- **Static Partitioning**: A lightweight virtualization layer targeting resource-constrained edge devices should not support the management interface since it can determine its setup in compilation time.

- **Directly mapped devices**: Direct access from applications to peripherals may be needed. If the peripheral does not need to be shared, the virtualization layer can allow direct mapping (bypassing) to the applications. This technique avoids the implementation of device drivers at the virtualization layer and improves performance. Otherwise, device drivers for shared peripherals, such as Ethernet, must be implemented and managed by the virtualization layer.

- **Small hypercall API interface**: Hypercalls are calls invoked from guests to the virtualization layer, similar to *syscalls* in a typical OS [102]. The hypercall API allows the implementation of extended services, i.e., services provided by the virtualization layer to its guests, like inter-VM communication or access to shared devices.

- **Simple and predictable scheduler**: The scheduler must implement proven algorithms to maintain predictability, like the round-robin scheduler. Additionally, the interrupt management can be simplified using the *pass-through* technique supported by hardware-assisted virtualization (i.e., interrupts can be redirected to guests without the virtualization layer intervention).

- **Simplified VCPU management**: The trap-and-emulate technique consists of emulating guest's privileged instructions (i.e., instructions that only can be performed on supervisor mode). However, one single instruction may require tens or hundreds of instructions from the virtualization layer side to be emulated, increasing the VCPU management complexity and overhead. Para-virtualization is commonly used to avoid such problems, which require the substitution of privileged instructions on guests by hypercalls. Thus, proper hardware-assisted virtualization can help keep the VCPU management small and straightforward because it allows for eliminating instruction emulation and most of the hypercalls. Nevertheless, hypercalls are still useful for virtualization extended services like inter-VM communication or peripheral sharing.

The lightweight virtualization layer presented in Figure 2.4 follows the microkernel approach. Hence, it can implement only the necessary hardware control and minimal services. Sophisticated network protocols, cryptography, file systems, and other libraries can be

supported at the VM level. For example, a VM with the picoTCP stack [8] can be instantiated as a separate application for network support. The same can be done with cryptographic libraries (e.g., WolfSSL[168]). A VM with an RTOS, like FreeRTOS, can also implement real-time services with a predictable execution in parallel. All these features result in a flexible and lightweight approach.

### 2.2.2 LITERATURE REVIEW FOR EMBEDDED VIRTUALIZATION

This section presents a review of lightweight virtualization approaches that can be adopted in a scenario involving resource-constrained edge devices. The essential characteristics for virtualized systems presented in Section 2.2.1 were considered to select the works discussed next. They are all type-1 hypervisors that do not require underlying GPOS. Table 2.1 presents a comparative analysis of the works regarding lines of code (LOC) and footprint (storage + RAM).

Table 2.1 – Lightweight virtualization approaches for resource-constrained edge devices.

| **Works** | **Ref.** | **Last Release** | **LOC** | **Memory (KB)** | | |
|---|---|---|---|---|---|---|
| | | | | **Storage** | **RAM** | **Footprint** |
| seL4 | [81] [150] | 2020 | 9400 | 138 | 24 | **162** |
| Muen | [27] | 2021 | 2700 | 75 | 16 | **91** |
| Xvisor | [113] | 2020 | 440000 | 1024-2048 | 4096-18432 | **5120-20480** |
| Hellfire | [103] | 2020 | 9800 | 21 | 2 | **23** |
| Bao | [90] | 2020 | 5600 | 41 | 18 | **59** |

Authors in [81] [80] [150] present seL4, a high-assurance, high-performance operating system microkernel. seL4 is the most advanced member of the L4 microkernel family. It supports virtual machines that can run a fully-fledged guest OS. Subject to seL4's enforcement of communication channels, guests and their applications can communicate with each other and native apps. In terms of source-code size, the kernel is about 9400 LOC (ARM and RISC-V). In terms of executable code size, the kernel has about 138 KB. Its RAM size is about 24 KB. Regarding virtualization overhead, the authors do not present numerical results but argue that seL4 adds minimal overhead.

The work in [27] presents Muen Separation Kernel, an open-source microkernel, formally proven to contain no runtime errors at the source code level. It uses Intel's hardware-assisted virtualization technology to provide strong separation and make its implementation simpler. The Muen project makes use of emulation by employing the Bochs IA-32 emulator. The kernel has approximately 2700 lines of code. The work does not present performance results.

Authors in [113] present Xvisor, an open-source type-1 hypervisor, focused on providing a monolithic, lightweight, portable, and flexible virtualization solution. It supports ARM virtualization extensions to provide full-virtualization and para-virtualization through optional VirtIO compatible device drivers. It can map interrupts directly to guests, allowing guest interrupts to be handled without the hypervisor's intervention. Additionally, it provides memory isolation between hypervisor, guests, and guest applications using the third privileged level from ARM's virtualization support. The kernel has approximately 440K lines of code. Experimental results show that Xvisor ARM guest has lower CPU overhead and higher memory bandwidth than KVM ARM guest and Xen ARM DomU.

The Hellfire hypervisor was developed by the GSE/PUCRS research group [2] [103]. It is a type-1 hypervisor and supports isolation, real-time, and inter-VM communications. It is designed to be as small and straightforward as possible; thus, using hardware support to avoid complex software implementation. It follows the microkernel approach and is custom-made during compilation time, i.e., its data structure length is defined during the building process. It is open-source software, mostly written in C language with a few Assembly lines, resulting in 9800 lines of code. It is available online[1].

Authors in [90] present Bao, a lightweight hypervisor implementation that uses a static partitioning architecture, supporting Armv8 and RISC-V platforms. Bao strongly focuses on isolation for fault-containment and real-time behavior. Its kernel has approximately 5600 lines of code. Tests were executed in Xilinx ZCU104, a quad-core Cortex-A53 running at 1.2 GHz per-core, and a shared unified 1MB L2/LLC cache. The hypervisor code and benchmark applications were compiled using the Arm GNU Toolchain version 8.2.1 with O2 optimizations. Results regarding memory show that it needs 23 KB of storage and 17 KB during runtime. To evaluate virtualization performance overhead, authors employed the MiBench Embedded Benchmark Suite [63]. Preliminary evaluation shows Bao generates an average virtualization overhead of 1.25% (one VM) and 32.50% for multiple VMs executions.

In recent years, there has been an increasing interest in containers, which are a vital element of modern cloud computing and play an important role in emerging edge computing applications [100] [101] [9] [89]. According to Morabito *et al.* [100], container-based virtualization provides a different level of abstraction in terms of virtualization and isolation compared to other virtualization solutions. Containers share the same OS kernel with the underlying host machine, making it possible to isolate standalone applications that own independent characteristics: independent virtual network interfaces, independent process space, and separate file systems [101]. However, it is a solution that still requires a GPOS.

After analyzing the possibilities of lightweight virtualization platforms, the Hellfire hypervisor was chosen as the virtualization layer for the proposed security architecture described in Chapter 4. The main motivations are the small footprint and strong isolation,

[1] https://github.com/hellfire-project/hellfire-hypervisor

important features towards security by separation in resource-constrained edge devices. A detailed description of the Hellfire hypervisor is given in Chapter 3.

## 2.3 SECURITY FOR EDGE DEVICES

Security is emerging as one of the most significant challenges for edge computing [126]. The decentralization of data processing and IoT applications to the edge made the devices more susceptible to attacks [7]. Also, edge devices are often deployed in resource-constrained environments without strict monitoring and protection, thereby facing all kinds of security threats [126]. Once an attacker controls the device, it is hard to identify that situation, and all system execution and state can be compromised. According to [174], increasing trust in such devices is the primary challenge. Hence, some aspects must be considered for security improvements: hardware-to-software oriented security and simple architecture building blocks due to resource limitations of edge devices.

Considering widespread published research in the security area [87] [36] [10] [173] [170] [136], and considering particular features of edge devices, such as processing power, storage capacity, network conditions, and heterogeneous applications, a taxonomy of "security for edge devices" was defined. The taxonomy presents challenges to overcome, relevant attacks and how they can violate software, and key security requirements. It is depicted in Figure 2.5. The security challenges and vulnerabilities are presented in Section 2.3.1 while the key security requirements in Section 2.3.2.

### 2.3.1 CHALLENGES AND VULNERABILITIES

In order to provide security for resource-constrained edge devices, there are important challenges that need to be considered:

- **Constrained hardware:** Edge devices can be resource-constrained devices in terms of processing capacity, memory size, storage, and bandwidth communication [12] [77]. It can result in severe security flaws, as only lightweight-based security mechanisms can be applied in order to protect data, applications, and the device itself [77] [66].
- **Simple and low-overhead building blocks:** Edge devices can deal with scenarios involving life support and low-latency applications. Thus, security solutions should follow a simple and low-overhead architecture to meet applications' QoS requirements, which is a challenging task in resource-constrained devices.
- **Data Processing and Scheduling:** An edge device can process data from sensors and make decisions on it. However, some devices do not have sufficient resources to

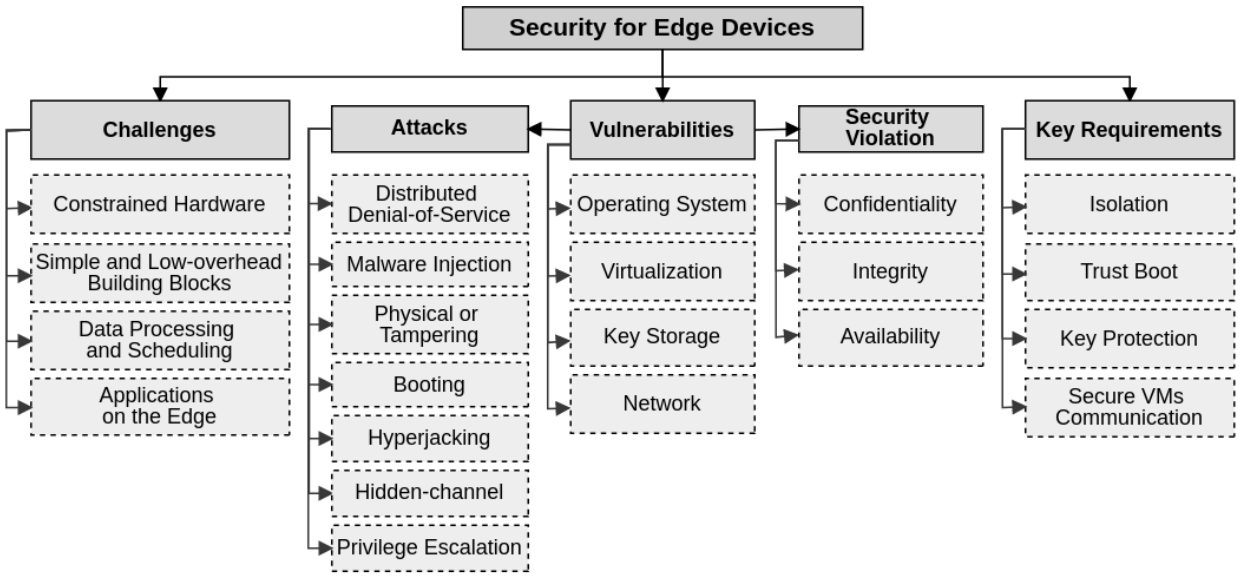



Figure 2.5 – Security taxonomy for edge devices.

do it. Thus, they should send data to be processed in other devices. These scenarios are drawing the attackers' attention to perform new attacks.

- **Applications on the Edge**: Relevant IoT applications are migrating from the cloud to edge devices, making them the primary target of attacks.

These challenges drive the rise of new threats at the edge. Also, old threats from traditional IoT environments are inherited by edge devices. Table 2.2 presents the most relevant attacks against them [126] [170] [1] [10] [65] [149]. Also, it describes vulnerabilities and which security requirements can be violated in each attack.

The security of edge devices can be violated regarding unauthorized access, use, disruption, modification, or even destruction [174] [7]. Most of the attacks aim to disrupt the device's software (e.g., operating system, virtualization, or key storage). However, attacks can also explore the network. In 2016, the Mirai attack [82] exploited basic flaws in IoT devices like hardcoded usernames and passwords for telnet. Once it successfully exploits a device, it converts the device into a bot controlled by the command and control server. The Mirai attack used thousands of IoT devices, such as cameras and video recorders, to cause a DDoS, generating high loads of traffic against the target, making it unavailable. In 2015, hackers tried to break the Jeep Cherokee's security remotely, taking control of the car's critical control system [166]. As a result, they could turn off the engine with the car traveling on a highway. It cost a recall for 1.4 million vehicles for the Chrysler automaker [167]. Many other attacks have caused significant damage, including Imeij [88], Brickerbot [39], Remaiten [161], and Linux.Darlloz [22].

Regarding security violations, attacks can explore confidentiality, integrity, and availability (CIA triad), which forms the basis of any assessment of security [176]:

Table 2.2 – Attacks on Edge Devices.

| Attacks | Vulnerabilities and Security Violation |
|---|---|
| Distributed Denial-of-Service | In a DDoS attack, an attacker first compromises a cluster of edge devices and takes full control of them. Then, it commands each device to launch a denial-of-service attack targeting a fog or cloud device, causing the shutdown of its services. It can happen as a flooding-based attack, which aims to shut down the normal service of a device based on a large amount of flooded malformed/malicious network packets, or as a zero-day attack, which must find an unknown vulnerability in the code running on the target, causing memory corruption and resulting in a service shutdown [82]. It violates availability. |
| Malware Injection | It injects malicious codes into the memory of edge devices. The attacker may force the device to perform some unintended functions or even access the complete system [45]. Thus, enabling adversaries to perform hacking processes, such as bypassing authentication, stealing data, or reporting false data. Backdoor attacks are examples of malware injection. The attacker spreads the malware in the device through unsecured entry points, such as outdated software or a firmware update. If the attacker gains access, it can cause damage to the device, and its applications [157]. It violates confidentiality and integrity. |
| Physical or Tampering | It happens if attackers can access the edge device physically. Valuable and sensitive cryptographic information can be extracted, the circuit can be tampered with, and the software/operating-system can be modified or changed [105]. It violates confidentiality, integrity, and availability. |
| Booting | Edge devices are vulnerable to attacks during the boot process. It happens if the inbuilt security processes are not enabled at that point. The attacker may take advantage of this vulnerability and try to attack the edge device when restarting. It violates integrity. |
| Hyperjacking | An attacker tries to subvert the existing hypervisor or inserting a rogue one. If successful, it can control any virtual machine running on the device. It violates confidentiality and integrity. |
| Hidden-Channel | Exploration of vulnerabilities regarding sharing of hardware components among the device's VMs. Data leakage across the VMs is a consequence of such an attack. It violates confidentiality. |
| Privilege Escalation | A malicious virtual machine can manipulate other VMs or take control of some elements of the device. It violates confidentiality and integrity. |

- **Confidentiality:** It means that the data is only available to authorized parties. When information has been kept confidential, it means that other parties have not compromised it. In edge devices, a breach of confidentiality may occur in different ways, such as backdoor violations.

- **Integrity:** It refers to the certainty that the data is not tampered with or degraded, either intentional or unintentional. In edge devices, integrity could also be compromised when an attacker controls the device, like in booting or physical/tampering attacks, during data transmission inside the device (inter-VM communication, for example) or storage.

- **Availability:** It means that the information is available to authorized parties when needed. An edge device must have adequately functioning computing systems and security controls to demonstrate availability. It must be resilient against DoS or physical/tampering attacks, which might impact the service's availability.

### 2.3.2 KEY SECURITY REQUIREMENTS

Trusted components can be used to build trust in edge devices. However, trust does not entail security. According to the Trusted Computing Group (TCG), "*an entity can be considered trusted if it always behaves expectedly for the intended purpose*". With hardware-assisted trust computing, the device will consistently behave in expected ways, and that behavior should be enforced by computer hardware and software [36]. This can be achieved by loading the hardware with unique encryption keys inaccessible to the rest of the system. Trusted computing is based on the concept of Chain-of-Trust (CoT). A CoT is established by validating each component of hardware and software from the Root-of-Trust (RoT) to the up entity, building a Trusted Execution Environment (TEE), which consists of an area of execution of a device where all code executed and application's data are verified for integrity [130]. A TEE starts during the device's power-up and persists during all system execution.

The key security requirements towards a secure architecture for edge devices involve trust components and lightweight virtualization and are discussed next [174] [126] [130] [87]:

- **Isolation:** It denotes a hardware-based architectural mechanism that can provide access control for software and its data. By placing code and data inside a protected module, no software outside can read or write its runtime state or modify its code. Execution of code inside such a module can only be started from a single predefined location [87]. Such an entry point ensures that attackers cannot reuse the module's code to extract secrets or implement malicious behavior.

- **Trust Boot:** To give strong security guarantees, an architecture should guarantee the integrity of its state. An edge device with a trusted boot can ensure that the device runs an authorized code and not a malicious code, which prevents malware installation. It can be implemented by verifying software components' authenticity during their initialization while preventing later modifications through isolation. It usually relies on digital signatures to protect the code authenticity. Also, the trusted boot needs to be anchored in an inherently trusted component referred to as RoT [130]. To be trusted, an RoT cannot be changed or substituted. It can be implemented in different ways. One approach is entirely in hardware, making it impossible to be replaced. Another possibility is in bootstrap software on a read-only memory (ROM) on-chip. A third way needs hardware capable of performing software verification based on a cryptography key stored in a write-once memory. RoT is typically combined with isolation to protect against attacks where an attacker changes the VM's code after it has been verified [87].

- **Key Protection:** A key protection scheme should be based on specialized hardware to protect the integrity of keys. For example, root public keys can be stored in a write-once memory to avoid key substitution. Alternatively, Physical Unclonable Functions (PUFs) can be used not just for device authentication but also for runtime key generation, avoiding the necessity of key storage.

- **Secure VMs Communication:** There is a need for protection in the interaction between VMs and other software parts of an edge device. The mechanism can be based on a hypercall API, and the hypervisor should work as an intermediate between communication from one VM to another.

## 2.4 TASK ASSIGNMENT AND SCHEDULING

The task scheduling problem refers to allocating resources to tasks, respecting the imposed constraints aiming to optimize one or combining performance measures. In the problem, we must allocate resources, usually limited, to activities to be carried out. Several definitions for scheduling can be found in the literature. A widely known definition is given by Pinedo [116], *"Scheduling concerns the allocation of limited resources to tasks over time. It is a decision-making process that has as a goal the optimization of one or more objectives."* According to Leung [83], resources may be machines in an assembly plant, central processing unit, memory, and input/output devices in a computer system, runways at an airport, and mechanics in an automobile repair shop. The tasks may be operations in a production process, take-offs and landings at an airport, stages in a construction project, executions of computer programs, among others [116]. Each task may have a priority level, starting time, and due date. There are also many different performance measures to optimize. One objective may be to minimize the makespan, i.e., the time difference between the start and finish of a sequence of operations. Another objective may be the minimization of the number of late jobs.

To identify possible problem characteristics that are useful in modeling the task assignment and scheduling problem in an edge-fog-cloud architecture, scheduling components such as machine environment, constraints, and optimization criteria are addressed in Section 2.4.1. The problem is described in Section 2.4.2.

### 2.4.1 SCHEDULING NOTATION

A scheduling problem can be described by a triplet $\alpha \mid \beta \mid \gamma$ [58]. The $\alpha$ field describes the machine environment and contains just one entry. The $\beta$ field provides details

of processing characteristics and constraints and may contain no entry at all, a single entry, or multiple entries. The $\gamma$ field describes the objective to be minimized and often contains a single entry.

According to Pinedo et al. [116], machine environments specified in the $\alpha$ field are presented in Table 2.3. The problem addressed in this work was modeled as a Flexible Job Shop scheduling problem, a generalization of the Job Shop scheduling problem. A more detailed description of its definition is given next.

Table 2.3 – Shop models and their symbols classified according to the machine environment.

| Environment | Symbol | Interpretation |
|---|---|---|
| Single machine | 1 | There is only one machine. This case is a special case of all other more complicated machine environments. |
| Parallel machine | *Pm* | There are *m* machines in parallel. Each job requires a single operation and may be processed on any machine. |
| Open shop | *Om* | Each job needs to be processed exactly once on each of the *m* machines. The route, i.e., order in which the operations are processed, can vary freely. |
| Flow shop | *Fm* | There are *m* machines linearly ordered and all jobs follow the same route, i.e., they have to be processed first on machine 1, then on machine 2, and so on. |
| Flexible flow shop | *FFc* | A flexible flow shop is a generalization of the flow shop and the parallel machine environments. Instead of *m* machines in series, there are *c* stages in series with at each stage a number of identical machines in parallel. Each job has to be processed first at stage 1, then at stage 2, and so on. |
| Job shop | *Jm* | Each job has its own predetermined route to follow. It may visit some of the *m* machines more than once and it may not visit some machines at all. |
| Flexible job shop | *FJc* | Instead of *m* machines in series, there are *c* work centers, where each work center consists of a number of unrelated machines in parallel. When a job on its route through the system arrives at a bank of unrelated machines, it may be processed on any one of the machines, but its processing time now depends on the machine on which it is processed. |

The JS scheduling problem can be stated as follows. Consider a set $V = \{1, 2, ..., o\}$ of operations, and a set $M = \{1, 2, ..., m\}$ of machines. Each job $J_i (i = 1, 2, ..., n)$ where $J_i \subseteq V$ consists of a set of $n_i$ operations that must be processed by machines in order to produce a product. A precedence relation saying that an operation $i$ must be processed before an operation $j$ is represented by an arc $(i, j) \in A$, where $A$ is the set of precedence relations. In the classical JS scheduling problem, precedence relations between operations belonging to the same job are denoted in form of chains (sequence), e.g., given an instance with two jobs $J_1 = \{1, 2, 3, 4\}$ and $J_2 = \{4, 5\}$, we simply have that $A = \{(1, 2), (2, 3), (3, 4), (4, 5)\}$. There is a machine $k_i \in M$ and a processing time $p_i$ associated with each operation. It implies that operation $i$ must be processed for $p_i$ time units on machine $k_i$. Thus, considering $s_i$ as the starting time of operation $i$, we have that the completion time $c_i$ of operation $i$ is equal

to $s_i + p_i$, i.e., $c_i = s_i + p_i$. Moreover, for each $(i, j) \in A$ we have that $c_i \leq s_j$ . No operation may be interrupted and each machine can process only one operation at a time. The problem consists in sequencing all operations on their respective machines, i.e, ordering processing of all operations on all machines to obtain a feasible scheduling solution.

The FJS scheduling problem, first introduced by [108], is a generalization of the JS scheduling problem. It is considered that there may be several machines, not necessarily identical, capable of processing each operation. Additionally to the set $V$ of operations, the set $A$ of precedence relations, and the set $M$ of machines, it is also given a function $F$ that associates a non-empty subset $F(i)$ of $M$ with each operation $i \in V$ . The machines in $F(i)$ are those capable of processing operation $i$. Moreover, for each operation $i$ and each machine $k \in F(i)$, a positive number $p_{ik}$ is given representing the processing time of operation $i$ on machine $k$. The FJS scheduling problem is twofold, assigning one machine to each operation (routing), and sequencing the operations on the machines (scheduling). Firstly, routing is concerned with assigning a machine to each operation of all jobs, i.e., defining the "route" of the jobs through the machines. A machine assignment is a function $\kappa$ that assigns a machine $\kappa(i) \in F(i)$ with each operation $i$. Given a machine assignment $\kappa$, consequently, the processing time of operation $i$ is equal to $p_{i\kappa(i)}$. Secondly, sequencing is concerned with ordering the processing of the operations to obtain a feasible scheduling solution. Commonly, the objective is to minimize the makespan, however, not limited to.

The JS scheduling problem is known to be NP-hard, being considered by many one of the most challenging problems in combinatorial optimization [56]. The scheduling problem in an FJS is at least as hard as the JS scheduling problem because it contains an additional problem that is assigning operations to the machines. The option of alternative resources ensures that the FJS scheduling problem is useful for scheduling in a wider range of applications.

The processing restrictions and constraints specified in the $\beta$ field may include multiple entries [116]. In the scope of this work, the precedence constraint should be defined. It requires that one or more jobs be completed before another job is allowed to start its processing. There are several forms of precedence constraints: if each job has at most one predecessor and at most one successor, the constraints are referred to as *chains*. If each job has at most one successor, the constraints are referred to as an *intree*. If each job has at most one predecessor, the constraints are referred to as an *outtree*.

Due dates are usually not explicitly specified in the $\beta$ field. The type of objective function gives sufficient indication whether or not the jobs have due dates. The $\gamma$ field is related to the objective function to be minimized. The objective to be minimized is usually a function of the completion times of the operations, which, of course, depend on the scheduling solution [116]. The most-used objective functions in the FJS scheduling problem in literature are described and formulated in Table 2.4.

Table 2.4 – Commonly used objective functions in the FJS scheduling problem.

| Objective | Symbol | Formulation | Interpretation |
|---|---|---|---|
| Makespan | $C_{\max}$ | $\max_{\forall i \in 1,\ldots,n}\{C_i\}$ | The maximal completion time between all operations of all jobs. |
| Maximum workload | $W_M$ | $\max_{\forall k \in 1,\ldots,m}\{\sum_{i=1}^{\lvert B_k \rvert} p_{ik}\}$ | The machine with most working time. Appropriate to balance the workload among machines. |
| Total workload | $W_T$ | $\sum_{i=1}^{o} p_{ik(i)}$ | Sum of the working time on all machines. Appropriate to reduce the total amount of processing time taken to produce all operations. |
| Total tardiness | $T$ | $\max_{\forall i \in 1,\ldots,n}\{0, C_i - d_i\}$ | Positive difference between the completion time and the due date of all jobs. |
| Maximum lateness | $L_{\max}$ | $\max_{\forall i \in 1,\ldots,n}\{C_i - d_i\}$ | Maximal difference between the completion time and the due date of all jobs. |

### 2.4.2 TASK ASSIGNMENT AND SCHEDULING PROBLEM IN AN EDGE-FOG-CLOUD ARCHITECTURE

An edge-fog-cloud architecture comprises edge, fog, and cloud devices, with multiple cores and different processing power. Also, they can perform a wide variety of tasks. In smart city applications, tasks are generated by edge devices and processed generally at the fog layer, where devices have more processing power. However, there are some situations in low-latency and compute-intensive applications that may prevent the QoS requirements as the task deadline from being met.

Video monitoring applications are typical in smart cities and are the best example to demonstrate such situations. Smart surveillance has garnered much attention in recent years, particularly by enabling a broad spectrum of interdisciplinary applications in areas like public safety and security, manufacturing, transportation, and healthcare [44] [172]. The main challenge of such a system is handling voluminous data, which are called “tasks” in this work. Cameras continuously send captured video frames for processing, which causes massive traffic, especially when all cameras in a system are taken into account.

In this work, a frame is considered a task requiring processing to detect something (e.g., face, object, among others). Each task must be processed on at most one of the multiple cores that are in the devices. Also, the tasks do not have to be processed in the same order of generation. However, they must respect the order of assignment in a core, i.e., a task that arrived in the device at time $t$ cannot be processed later than one that arrived at time $t + 1$.

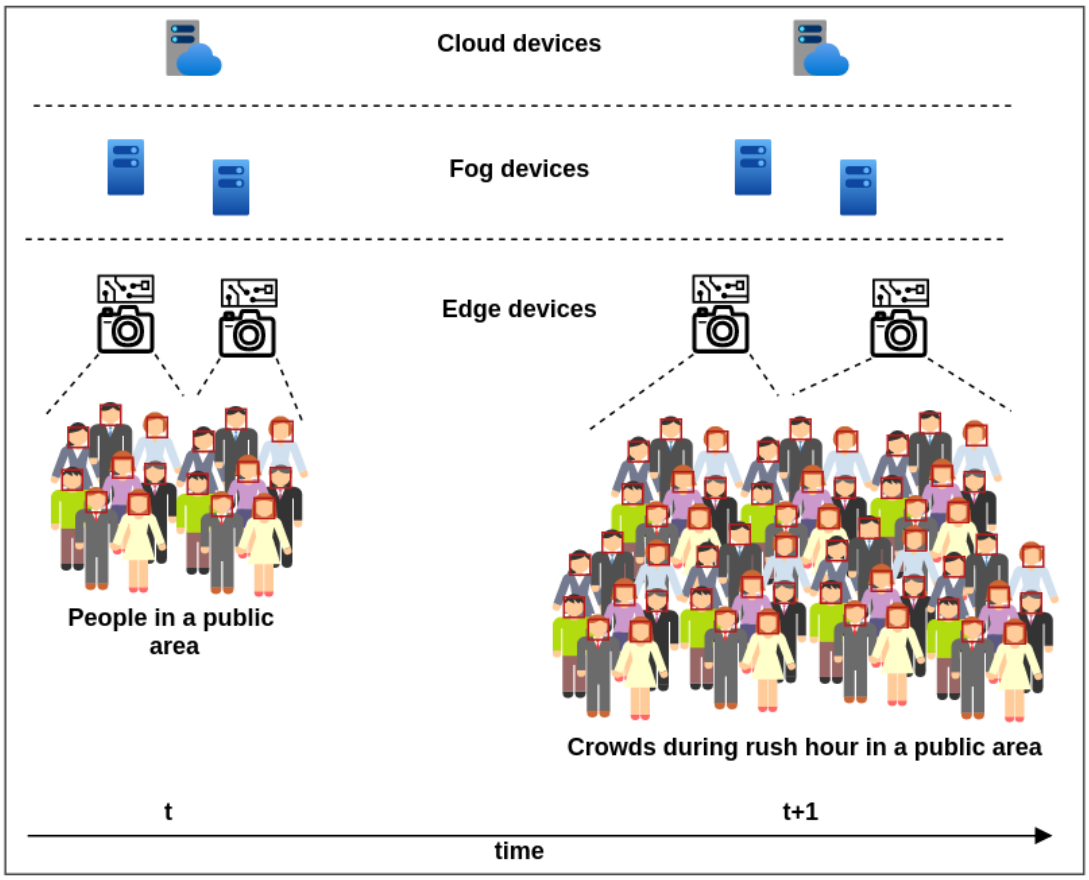



Figure 2.6 – Video monitoring application in a public area of a city.

Typically, videos are segmented into frames or set of images. A single video camera can produce about 25-30 frames/second or more with 4K and 3D video cameras depending on the video's sampling rate. Considering the volume and generation rate of data, algorithms to process the video must offer real-time or near real-time solutions [109]. According to [13], using compute capacities available on the camera itself allows for correspondingly lower provisioning (or usage) in the fog/cloud, enabling promising applications, such as access control in areas of interest, human identity or behavior recognition, detection of anomalous behaviors, interactive surveillance using multiple cameras, crowd flux statistics, among others [69]. However, let us consider the scenario illustrated in Figure 2.6. A video monitoring application performs face recognition of people in a city's public area at time $t$. The number of people increases considerably at time $t + 1$, but the edge-fog-cloud architecture remains the same. Some devices may suffer from overloading when the nearby user population becomes dense at some time of the day. Simultaneously, other devices may be idle most of the time, leading to loss of task deadlines and consequently causing image detection problems for the video monitoring application [164]. Although it is rare for real-time processing applications not to meet the deadline, it can happen [17]. The reasons may be a complete system failure or an inability to work during the system's peak time. If the returned results

exceed the deadline, they become invalid and may cause severe consequences to people depending on the application.

The assignment and scheduling of tasks considering the whole architecture, from the edge to the cloud, becomes fundamental towards meeting QoS requirements. It is a way to achieve the lowest latency for each task, avoiding missing the deadline. Authors in [164] argue that it is unnecessary always to use the closest device to assign a task, as long as a particular QoS requirement is guaranteed. When the edge devices do not have enough computing or storage capacity, the task can be assigned to the fog or the cloud to improve the system's overall performance [76].

Task assignment and scheduling should be addressed jointly in an edge-fog-cloud architecture [163]:

- **Task assignment**: An edge device can choose any core in the system, including itself, to assigned its task. The decision may be based on the propagation latency, the occupied bandwidth, the device processing capacity, the waiting time of a task to be processed, among others.
- **Task scheduling**: When assigning a task to its destination, the device's tasks' processing order needs to be defined.

This thesis addresses minimizing the total tardiness of tasks generated by edge devices during peak times of low-latency IoT applications in an edge-fog-cloud architecture. Note that this problem is addressed in this work as static scheduling, i.e., at each time $t$ an instance of the problem is solved.

## 2.5 SUMMARY

This chapter presented concepts regarding this thesis. The concept of Edge computing was defined and distinguished from Fog and Cloud concepts. Also, the architecture models involving Edge, Fog, and Cloud were presented. Virtualization was defined in the context of edge devices, and a literature review for embedded virtualization was conducted to define a lightweight virtualization hypervisor to be used in the security architecture. The security concept was presented along with challenges, attacks, and security requirements for resource-constrained edge devices. Finally, the task assignment and scheduling problem was presented and discussed in the context of an edge-fog-cloud architecture.

# 3. THE HELLFIRE HYPERVISOR

This chapter presents the features and operation of the Hellfire hypervisor, which is used as virtualization layer in the proposed security architecture (Chapter 4). The description presented in the following sections is based on the implementation of the Hellfire hypervisor on a MIPS32 (Microprocessor without Interlocked Pipeline Stages) architecture, for which it was initially developed. However, it already supports RISC-V for both rv32 and rv64 architectures.

## 3.1 PRIVILEGE-LEVELS AND CONTEXT-SWITCHING

Before understanding the MIPS privilege-ring and the context-switching scheme, it is essential to know that the MIPS has the *coprocessor 0* (CP0), a set of configuration registers accessed by the special instructions *mfc0* and *mtc0* in privileged mode only. In the MIPSVZ specification, a subset of these registers, named guest CP0 (GCP0), are duplicated, and they may be accessed from the VM's privilege level if allowed by the hypervisor.

Figure 3.1 shows the complete privilege-ring and all possible ring transitions to the MIPSVZ core. It implements the root-kernel, root-user, guest-kernel, and guest-user ring levels. A GPOS would use only the root-kernel and root-user ring levels that are backward-compatible to the kernel and user-modes from classic architecture. The hypervisor uses the most privileged-ring (root-kernel) and delegates the VMs to the guest-kernel mode. Keeping the VMs in a less privileged ring allows for the hypervisor to create separation. Additionally, it uses a subset of the ring transitions, shown in Figure 3.1 as dotted arrows. Thus, it is possible to handle interrupts directly in the guest-kernel mode.

The flowchart in Figure 3.2 describes the operations executed during the transitions. On entering the hypervisor exception handler (root-kernel), the GPRs are saved. In the case of a timer interrupt, the CP0 context is saved, the scheduler selects the next VM restoring its CP0 context. Any other interrupt will trigger the corresponding device driver. Other exceptions as hypercalls or guest faults are handled accordingly. Finally, the GPR context is restored, and the control is returned to the guest. Hence, the transition from guest-kernel to root-kernel happens in the following situations: (i) root-kernel interrupts and (ii) guest-kernel exceptions. For example, if an interrupt targeting the root-kernel happens, the processor will jump to the hypervisor's interrupt vector handler. Thus, it will perform the GPR context saving, perform the required operation at the device driver level, restoring the GPR, and jumping back to the guest-kernel mode. During the context restoring, two special operations must be performed: (i) set the Exception Program Counter (EPC) (a CP0 register

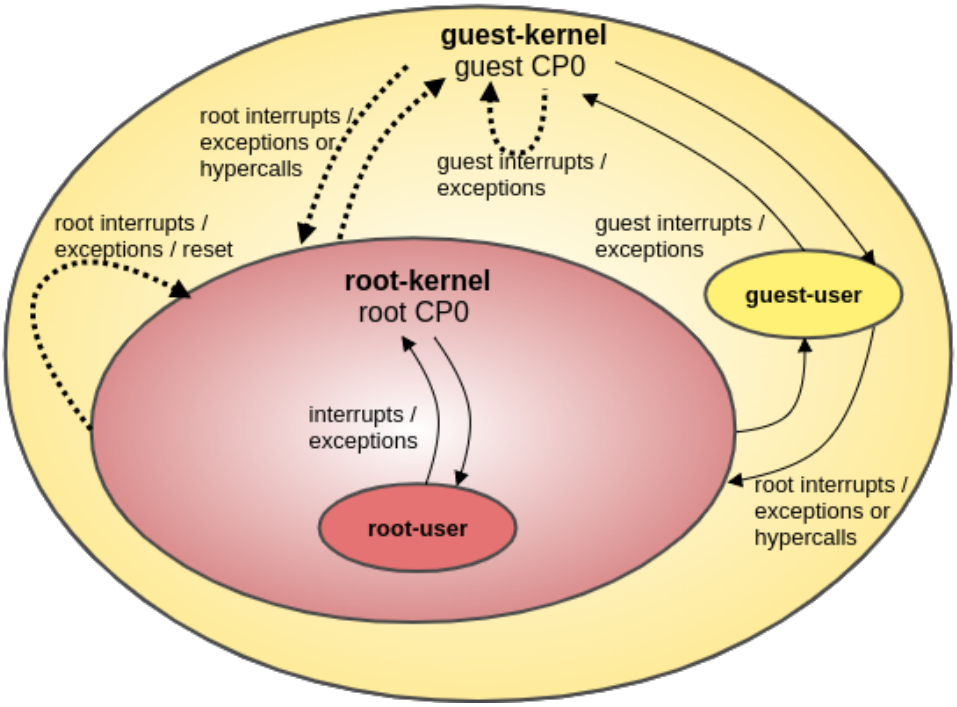



Figure 3.1 – The MIPSVZ privilege-ring model and its possible transitions. Dotted arrows show the transitions used by the hypervisor implementation.

that keeps the VM's program counter); and (ii) set the CP0 guest_id register (used to select the TLB entries and detailed in Section 3.2).

## 3.2 MEMORY VIRTUALIZATION

This subsystem is mainly responsible for the separation, keeping the different domains isolated. Its existence relies on a hardware mechanism present in the processor: the Memory Management Unit (MMU). Memory addresses generated by the processor's core, called virtual addresses (VA), need to be translated to physical addresses (PA) by the MMU controlled by an operating system or a hypervisor. The OS needs to keep a page table (PT) for each process mapping VA to PA. During context-switching, the OS remaps the processor to the corresponding page table. If a process requires an address translation not present in the page table, a trap is issued (page-fault). Thus, malfunctioning or even malicious memory access is detected and stopped. The MIPSVZ module implements hardware-assistance for memory management with an additional translation level, called two-level MMU. In this scheme, the guest OS configures its virtual memory using the guest CP0 in the same way in a standalone environment. Typically, the hypervisor keeps a page table for each VM and configures the MMU accordingly to the guest's needs. During memory translation, the two-level MMU generates IPA (Intermediate Physical Address) from VA based on the guest's MMU. The PA will result from the combination with the hypervisor's MMU configuration. This

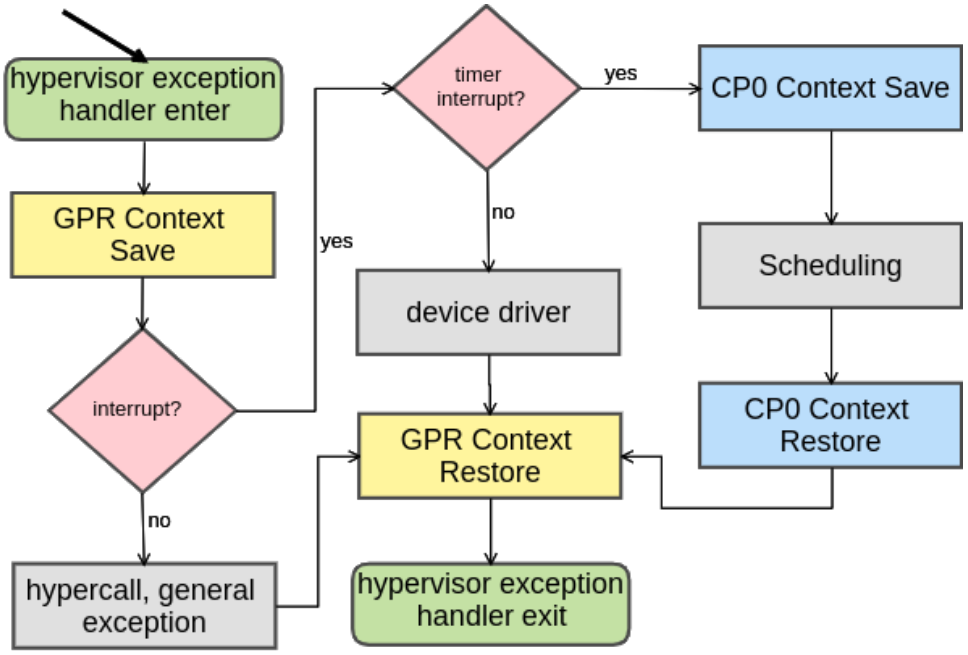



Figure 3.2 – Flowchart to the hypervisor's context-switching and exception handler.

scheme avoids modifying the guest OS while reducing the virtual memory configuration and translation traps. Figure 3.3 describes this mechanism.

Standard hypervisors for cloud computing implement a complete paging mechanism. As stated, the Hellfire hypervisor keeps a page table to map guest OSs to physical memory. In these systems, the guest OS does not need to be entirely loaded into the main memory to be executed. The hypervisor can implement an on-demand paging mechanism (swapping). Such a scheme reduces memory usage since pages that have not been used recently can be stored in the swapping system. Additionally, it avoids the memory external fragmentation problem because the VMs do not need to be allocated contiguously in physical memory. However, this approach has critical drawbacks for resource-constrained edge devices. First of all, swapping systems and on-demand paging mechanisms impact real-time responsiveness. Nevertheless, some resource-constrained edge devices do not support swapping due to storage restrictions. Moreover, a complete virtual memory management mechanism implies a more complex hypervisor and, consequently, a larger footprint and more processing requirements.

A *simplified virtual memory management* mechanism brings some advantages to resource-constrained edge devices. First, it avoids second-stage translation misses keeping the VM whole mapped at the hardware during its execution. Thus, bare-metal applications, RTOSs, or Unikernels that do not implement virtual memory support will not suffer additional delays and jitter due to hypervisor paging management. Additionally, the limited number of virtual machines usually required by resource-constrained edge devices allows for a static configuration. For these systems, memory fragmentation due to contiguous guest OS allo-

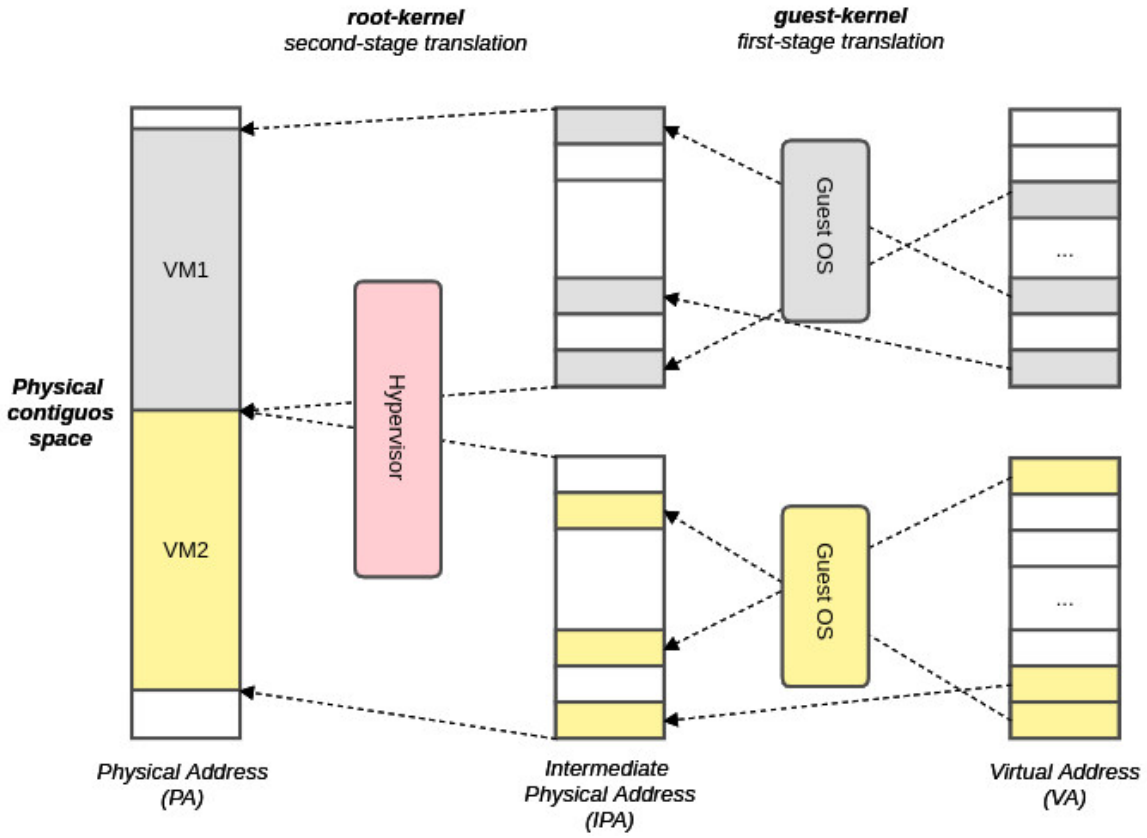



Figure 3.3 – Hellfire Hypervisor memory management strategy using the MIPSVZ two-level MMU hardware support. VMs are contiguously mapped in the physical memory.

cation is not a significant problem. Thereby, the hellfire hypervisor simplifies the memory management by combining two distinct techniques:

- static VM's memory allocation (detailed in Section 3.5);
- avoiding to keep a complete page table scheme in memory. The contiguous memory allocation is represented in Figure 3.3.

## 3.3 I/O VIRTUALIZATION

Some hardware peripherals need to be shared among the VMs, like Ethernet and timers. For example, when a guest tries to read or schedule a timer interrupt, the hypervisor will need to intercept these actions by traps or using para-virtualization to share the device properly. Similarly, the hypervisor may implement a network switch layer to allow guests to access the external world. As a consequence, all I/O may need to be controlled by the hypervisor. Both examples are complex in terms of implementation and lines of code. Also, they may impose performance penalties on the hypervisor. VirtIO [128] surged as an effort to standardize the I/O interfaces for Linux hypervisors consisting of a set of Linux modules.

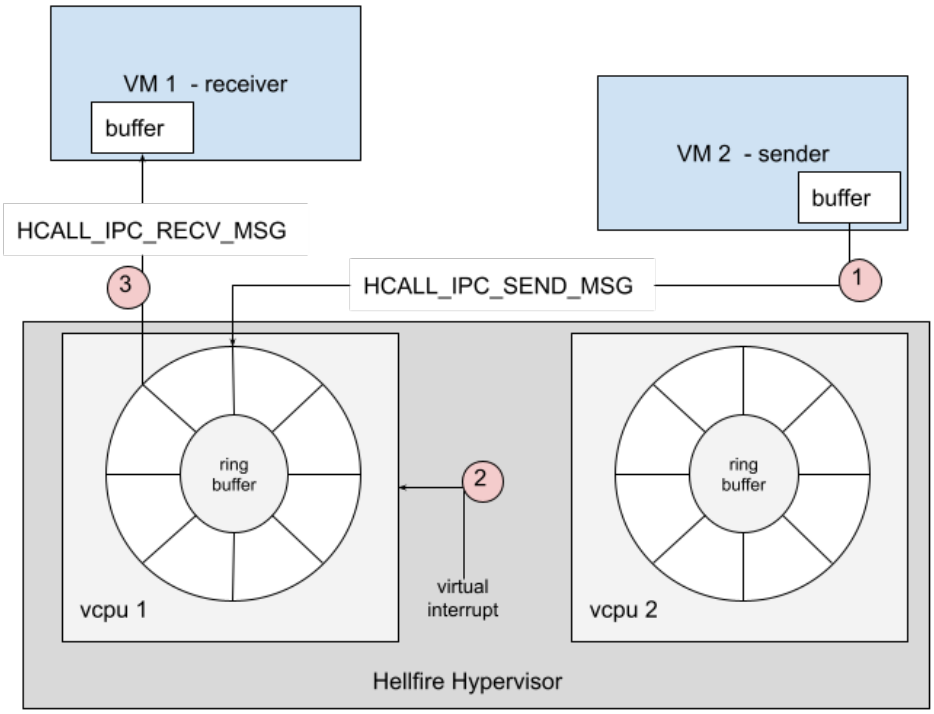



Figure 3.4 – Example of inter-VM communication involving two VMs.

Nonetheless, simplified subsystems are essential for embedded hypervisors. For example, the ability to map a peripheral directly to a VM redirecting its interrupts can save many efforts and diminish the hypervisor attack surface.

The hypervisor implementation supports *directly mapped devices*, which requires mapping non-continuous memory regions to a VM. Usually, I/O devices are mapped to specific physical addresses. For example, a VM may have mapped 32 KB of RAM allocated in the physical memory from 0x1000_0000 to 0x1000_8000. If the same guest requires access to a peripheral at the physical address 0x1F00_0800 the hypervisor must configure a TLB-entry to match it. The static partitioning approach allows for defining all direct-mapped devices in a configuration file, as stated in Section 3.5.

## 3.4 INTER-VM COMMUNICATION

The hypervisor defines a hypercall interface for communication among VMs. This implementation adopts a message passing mechanism based on para-virtualization. The hypervisor routes messages among VMs using the address, size, and ID destination, which are hypercall parameters configured by the guest. Thereby, the hypervisor does not make any assumptions about the message formatting. Suppose a multi-task guest OS needs to demultiplex [146] incoming messages among different tasks. In that case, the hypervisor

may add a header to the message indicating the origin and destination task id. In this case, the communicant guests must agree about the header format.

Each VCPU implements its incoming message queue as a limited circular buffer, statically allocated for performance purposes. A message targeting a determined VCPU will be copied to its queue, and the hypervisor will insert a virtual interrupt to the guest. The next time that the guest is executed, it will handle the virtual interrupt and call a hypercall to retrieve the message. Figure 3.4 describes hypervisor behavior while redirecting messages between guests. The VM 2 invokes the HCALL_IPC_SEND_MSG hypercall (1), causing a message copy from its buffer to the ring buffer of the destination VCPU. After, the hypervisor injects a virtual interrupt (2) in the VCPU 1. In the next execution, VM 1 will handle the interrupt executing the HCALL_IPC_RECV_MSG hypercall. Thus, the hypervisor will copy the ring buffer's message to the target buffer (3), completing the message's sending.

## 3.5 STATIC PARTITIONING

Cloud hypervisors implement management interfaces to allow users to configure all system elements. Cloud computing requires to instantiate or stop VMs without overall system interruption. Migration or reconfiguration should not influence the execution of the other guests. As discussed prior in this section, typical resource-constrained edge device applications restrict the number of VMs, and usually, they must be executed during all device's runtime. Beyond simplifying the memory subsystem, as seen in Section 3.2, static partitioning benefits from these characteristics. Static partitioning consists of determining the system resources allocated to each guest at design time. For example, memory space, scheduling priorities, directly mapped devices, among other resources, are estimated by the developers and defined programmatically before compilation. Despite this method being less flexible than using an underlying GPOS or a hypervisor with a management interface, it brings two advantages: a small attack surface and simplicity.

To make the definition of the system's setup easier, there is a scheme involving a configuration file and a tool to process it. Thereby, the partitioning is written in a structured file parsed by the *libconfig*, a C/C++ library for processing configuration files. This library has a compact and readable content format that is similar to JSON schemes. In the configuration file, the user gives details about the system to be built. An example is given in Figure 3.5. The figure's left side shows a configuration file example with a section called *system* where it is defined debug flags, serial speed, and the hypervisor scheduler's quantum. A section called *virtual_machines* allows for creating a list of VMs specifying their scheduling priority, memory size, storage, mapped devices, and interrupts.

During the building time, the configuration file is read by a tool called *gentool*. It helps the developers to configure the system from a higher abstraction level view. *Gentool*

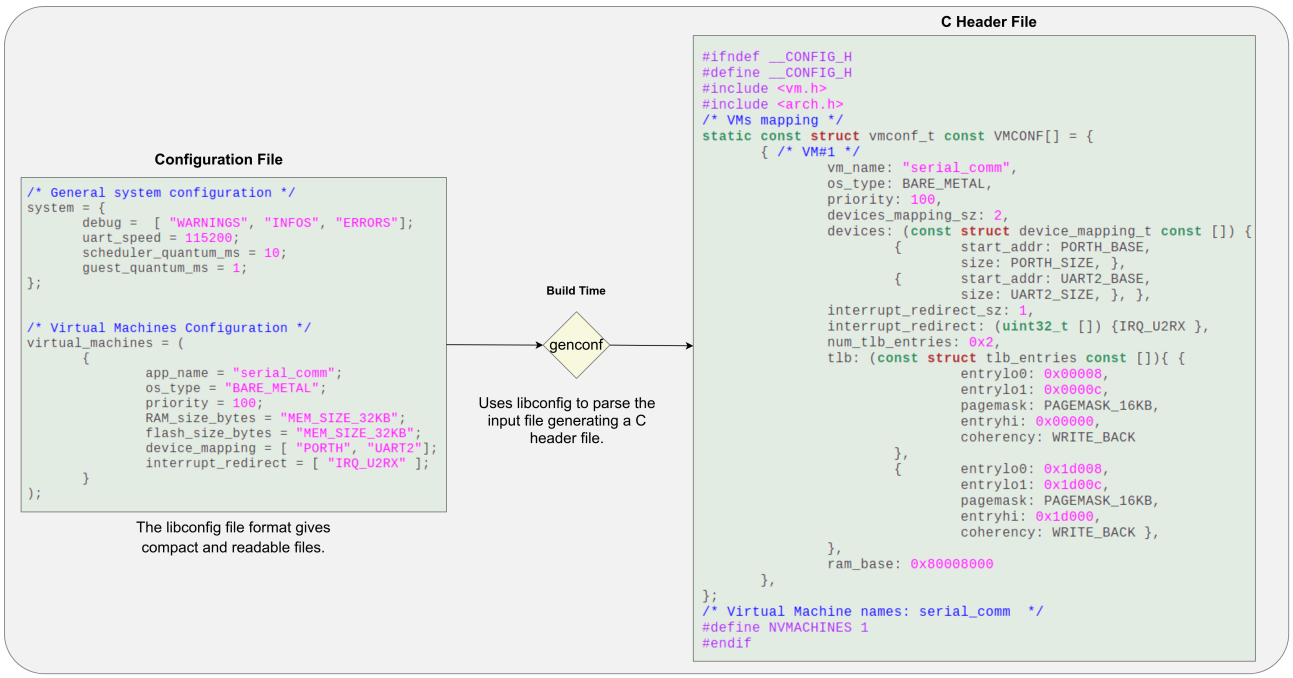



Figure 3.5 – Configuration file versus the C header generated by the genconf tool.

knows the platform architecture details and creates the partitioning based on the number of VMs and the required size for each one. It outputs a C header used in the rest of the compilation process. The VM's configuration is grouped by a data structure, called *struct vmconf_t*, a fixed-size array that keeps the meta-data processed from the configuration file (see the right side of Figure 3.5). Note that the *gentool*'s output is substantially more complex than the input file, especially memory partitioning. The tool considers the memory and storage sizes to optimize the allocation scheme for a given VM's set. The resulting allocation is stored in a fixed size array of *struct tlb_entry* elements. This information is read by the hypervisor during booting time and used to configure the processor's TLB.

Furthermore, the *gentool* keeps details about hardware devices and interrupts. The *device_mapping* property in the configuration file gives an array of device names to be mapped to the guest. The tool creates a structure called *struct device_mapping_t* that keeps the memory addresses and size of the memory-mapped devices allowed to the guest. Similarly, the *interrupt_redirect* is an array that keeps interruptions that must be redirected to the guest. Finally, *gentool* gives a convenient way to configure the system and promote the hypervisor and the VM build.

## 3.6 REAL-TIME SUPPORT

Several aspects may impact real-time responsiveness, such as paging and swapping schemes or scheduling policies. Techniques as on-demand paging [137] or swapping bring execution unpredictability because when a required page is not present in memory,

the process or VM is blocked until the data is loaded. The loading time may vary depending on the system load and the kind of storage involved. RTOSs overcome these problems by simplifying their implementation. For example, no memory management and a more straightforward software stack with predictable scheduling algorithms.

The Hellfire hypervisor finds a trade-off between memory management's advantages, as isolation, and the drawbacks in responsiveness. Thereby, it implements only the memory management features needed to provide separation. No additional schemes like on-demand paging or swapping are provided. A predictable round-robin scheduling algorithm with priority is implemented. Another feature of the hypervisor is the ability to support directly mapped devices, bypassing the hypervisor to access certain devices, avoiding additional overhead, and improving responsiveness. This feature is associated with the interrupt pass-through, where interrupts can be redirected to a VM without hypervisor intervention.

## 3.7 EVALUATION

This section presents the evaluation tests performed in the Hellfire Hypervisor. The main goal is to evaluate if it is possible to use it in resource-constrained edge devices. Thus, the next subsections present the hypervisor footprint, the virtualization performance impact using a well-known benchmark, the inter-VM communication delay for a set of two applications, and real-time results. The Microchip PIC32mz was adopted as the hardware platform. It comprises an M5150 processor core (MIPS32 architecture) with 512KB of SRAM and 2MB of flash. It is depicted in Figure 3.6. The M5150 core implements the MIPS Virtualization Extension (MIPSVZ), making it a perfect testbed.

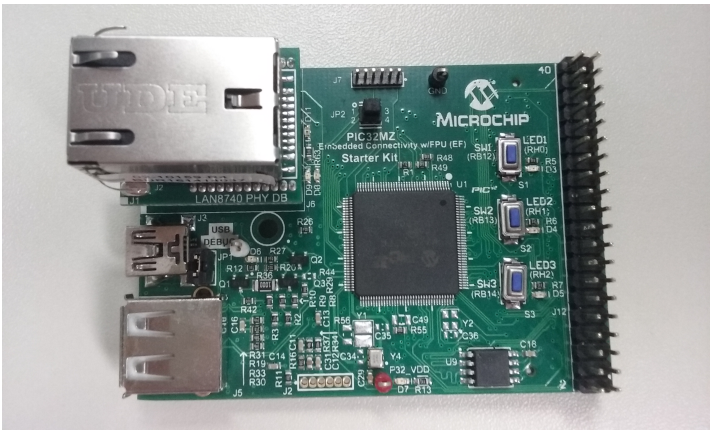



Figure 3.6 – Microchip PIC32mz.

### 3.7.1 FOOTPRINT ANALYSIS

The footprint aspects are essential for resource-constrained edge devices. The tests measured the hypervisor footprint for configurations with one, two, three, and four VMs, with the inter-VM communication, Ethernet, USB, and interrupt redirection drivers enabled. The one VM system consists of a simple blink-led application. The two VMs implement the ping/pong application and use the inter-VM communication mechanism to exchange messages. The three VMs system consists of a combination of the blink-led and the ping/pong applications. To the four VMs system, a VM that performs the Coremark benchmark was included (see Section 3.7.2 for benchmark results).

The source code was compiled using GCC 4.9.2 (Codescape GNU Tools 2016.05-03 for MIPS MTI Bare Metal) with Binutils 2.24.90. Four optimization levels were used: O0 (no optimization); O2 (most of the supported optimizations that do not involve a space/speed trade-off), O3 (all O2 optimizations more optimizations for speed that may increase the footprint), and Os (optimize for space usage). Additionally, the compiler flag *-micromips* was used. MicroMIPS is a code compression instruction set architecture that offers 32-bit performance with 16-bit code size for most instructions and is supported by the M5150, which allows for significant code reduction. Table 3.1 shows the results. All numbers are given in bytes, and only the hypervisor footprint is considered (VMs size is not included). The column *text+ro* means the size of the instructions and the read-only segments (kept in flash storage), while *data+bss* is the sum of the global initialized data and non-initialized data (loaded to RAM during boot). As the optimization levels do not affect the *data+bss* size, Table 3.1 shows only a column for all results.

Table 3.1 – Footprint results for the Hypervisor (bytes).

| | **GCC Optimization Level** | | | | |
|---|---|---|---|---|---|
| | **data+bss** | **text+ro** | | | |
| **#VMs** | **all opt.** | **O0** | **O2** | **O3** | **Os** |
| **1** | 2016 | 32632 | 21328 | 25496 | 20156 |
| **2** | 2028 | 34952 | 21548 | 25716 | 20344 |
| **3** | 2048 | 37620 | 21684 | 25852 | 20468 |
| **4** | 2068 | 40104 | 21788 | 25984 | 20584 |

It is seen that the compiler optimization level plays an important role. For example, the one VM system has a total footprint of 34648 bytes (*text+ro* plus *data+bss*) for O0 optimization and 22172 bytes when optimized for Os, a reduction of 36%. In all optimization levels, a small increase in *text+ro* and *data+bss* sections with additional VMs can be noted. This happens because it is allocated a *struct vmconf_t* (see Section 3.5) in the read-only section for each new VM. Additionally, a *struct vcpu_t* (a data structure that keeps the

execution status of a VCPU) is allocated in the data section for each VM. Finally, based on footprint results, using optimization levels O2 or Os makes it possible to keep the total footprint around 23KB, which is very optimistic for a hypervisor.

### 3.7.2 PERFORMANCE ANALYSIS

Coremark[2] is a benchmark used to measure embedded processors' performance. It was designed to run on microprocessors from 8 to 64-bit. It implements algorithms like list processing, matrix manipulation, state machine, and CRC (cyclic redundancy check). All are everyday operations in embedded applications. The Coremark result is a score number that can be used to compare performance among different processor families. The goal is to determine the virtualization impact by comparing the native Coremark score with the hypervisor under different system configurations. Five different setups were used: native, one VM, two VMs, three VMs, and four VMs systems. For the native setup, the Coremark as a standalone application was performed, i.e., without the hypervisor. The remaining setups consist of a different number of parallel VMs running the Coremark application. Thereby, the native result score was compared to the score of different system setups to find the hypervisor overhead. In all setups, the optimization level used was O2, and the hypervisor scheduler quantum was configured for 5ms, i.e., it performs context-switching every 5ms.

Figure 3.7 shows the results. The native execution resulted in a Coremark score of 589.93, while the one VM system was 588.32, giving a performance penalty of 0.27%. For the remaining setups, the CPU time was equally shared among all VMs, i.e., in the four VM systems, each VM had only 25% of the CPU time. Thus, the resulting score was divided among the VMs. Observe that, for the two systems VM, Figure 3.7 shows the resulting score for each VM and a column bar with the sum. In this case, VMs' scores were 293.63 and 293.72, resulting in a total of 587.36, which gives an overhead of 0.43%. Using the same technique, overheads of 0.68% and 0.76% were found for the three and four VMs systems. These are very optimistic numbers that result from two main reasons: (i) the low hypervisor code complexity and (ii) the MIPSVZ hardware features, especially the TLB and the GPR shadows that keep the context-switching lightweight.

The MIPS 5150 implements performance counter registers [72] that can be programmed to count different kinds of hardware events, e.g., number of executed instructions or invoked hypercalls. Thereby, the register counters were used to determine the impact of the different setups over the cache. For this, the counters were programmed to issue the number of data and instruction cache misses. Figure 3.8 depicts the cache impact when adding more VMs to the system. As expected, the cache misses for data and instructions increases exponentially with the addition of VMs. The M5150 processor core has only 16

[2]https://www.eembc.org/coremark/

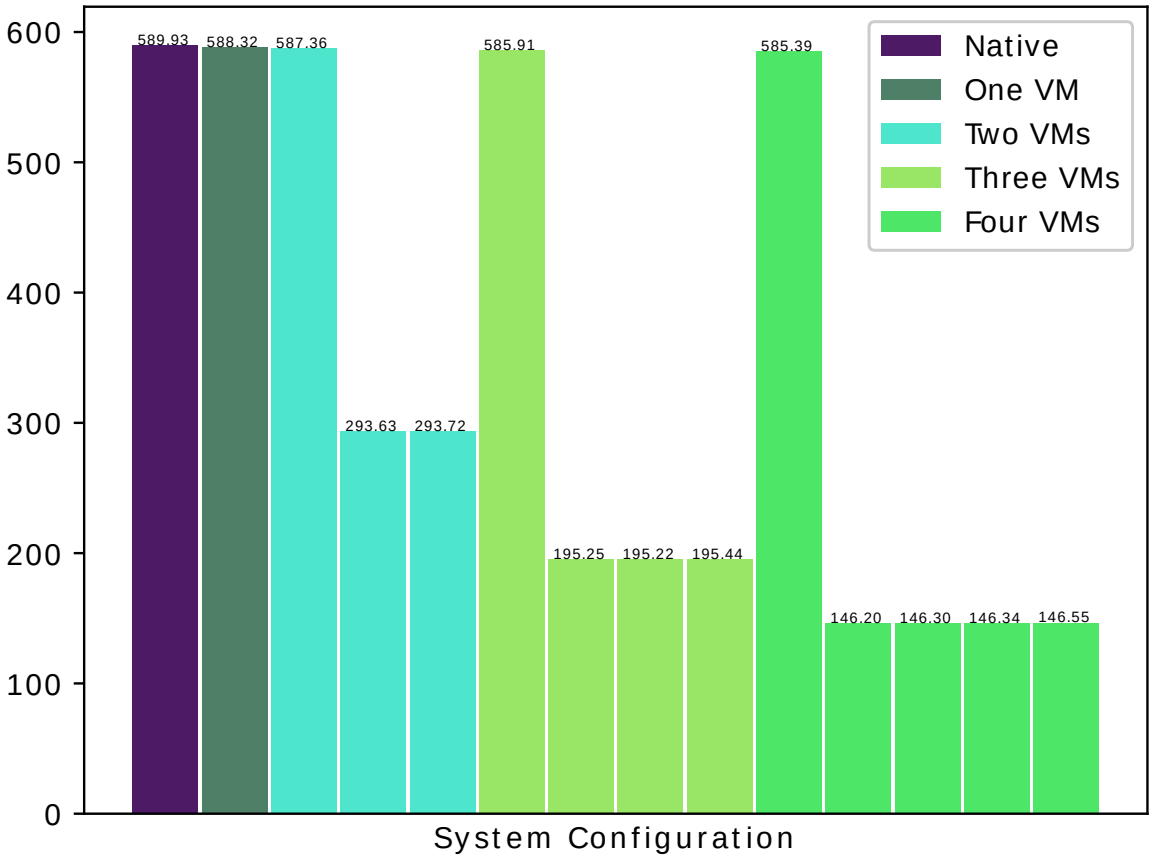



Figure 3.7 – Coremark's score for an increasing number of VMs.

KB for instruction cache and 4 KB for data cache. The context-switching between VMs changes the spatial memory location abruptly, forcing new cache lines to be loaded. Additional VMs mean different spatial locations being accessed, and the amount of cache has not been enough. This problem may be minimized, increasing the scheduler quantum to 10ms, causing two times less context-switching.

### 3.7.3 INTER-VM COMMUNICATION DELAY

The inter-VM communication was evaluated regarding the latency of message exchanges between VMs. Thus, the test involves two VMs. The first one works as an echo server that replies to all received messages. The second one sends messages of 256 bytes repeatedly, calculating its round-trip time. It is called the ping-pong application. As a result, an average round-trip time of 199.97 $\mu$s was obtained after 1000 messages. Thereby, the inter-VM mechanism can be considered an efficient way to implement communication on the virtualized platform.

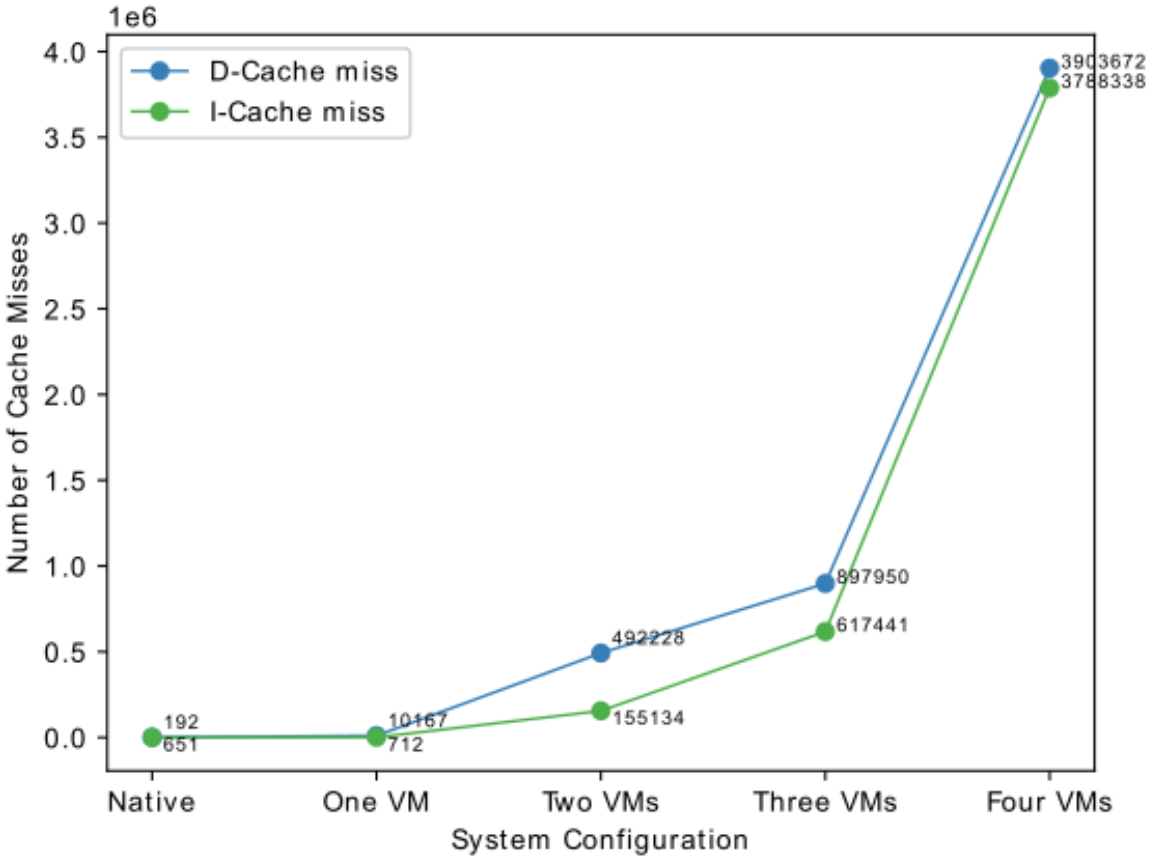
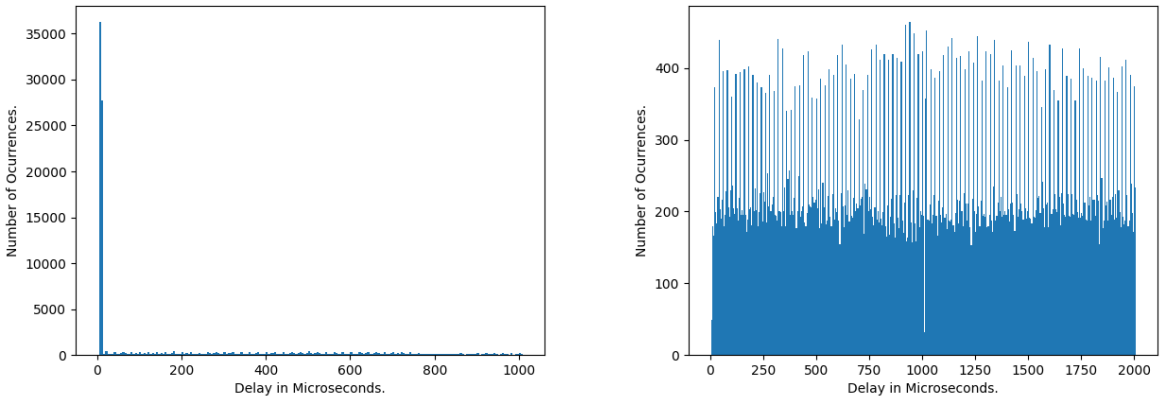



Figure 3.8 – Cache impact for increasing number of VMs.

### 3.7.4 REAL-TIME ANALYSIS

For real-time applications, it is essential to understand the behavior of the system regarding the response delay. The response time for interrupt handling was tested to measure a VM's responsiveness in the Hellfire hypervisor. For this, a VM capable of receiving interrupts from an I/O pin (external source) and generate outputs to another I/O pin was implemented. Hence, a function generator to issue interrupts every ten milliseconds was used. The tests measured the instants of the generated interrupt and the response in the output pin for each interrupt. The time difference is the total delay to the system to react to an external event. The responsiveness was tested in three situations:

1. The system idle, composed of 1 VM: the VM for the latency test;
2. The system under moderate load, composed of 2 VMs: the VM for latency test and the blink-LED application;
3. The system under heavy load, composed of 4 VMs: the VM for latency test, the blink-LED application, and the ping-pong application.




(a) Response delay for the system with a moderate load (2 VMs).

(b) Response delay for the system with a heavy load (4 VMs).

Figure 3.9 – Histograms for interrupt responsiveness for system under moderate and heavy loads.

The ping-pong application generates a heavy load on the system since it changes messages exhaustively. For each situation, 100000 interrupts were generated, obtaining the response delay for each one. For the system idle test, the average response time was 8$\mu$s (microseconds). Since there is only one VM, the interrupts are directly sent to the VM (interrupt pass-through), resulting in a fast response. In the moderate and heavy load tests, interrupts may happen when the target VM is not in execution, requiring rescheduling. The rescheduling may happen when the interrupt arrives, or it is postponed if the hypervisor needs to attend to other VMs. Figure 3.9 presents the response delay histograms showing the time distribution. For the system under moderate load (2 VMs), see Figure 3.9(a), the average response time was 173.23$\mu$s with a minimal of 8$\mu$s and a maximum of 1008$\mu$s. For the system under heavy load (4 VMs), see Figure 3.9(b), the average response time was 1010.48$\mu$s with a minimum of 8$\mu$s and a maximum of 2008$\mu$s. A variation can be seen depending on the system load, but it is possible to determine the worst-case response time making the resulting system behavior predictable.

### 3.7.5 SMART CITY APPLICATION - AIR QUALITY MONITORING

The hellfire hypervisor was also evaluated in a scenario that highlights its capabilities for edge computing deployment. The scenario, presented in Figure 3.10, depicts a typical smart city application that monitors the air quality in urban areas. Air quality is usually monitored by networks of fixed stations strategically placed in the city, where each station can measure a wide range of pollutants (Figure 3.10A). Each station has a resource-constrained edge device in this scenario, which was implemented in a MIPS32 processor core running at 200 Mhz, with 2 MB of flash memory and a 512 KB SRAM. The device was

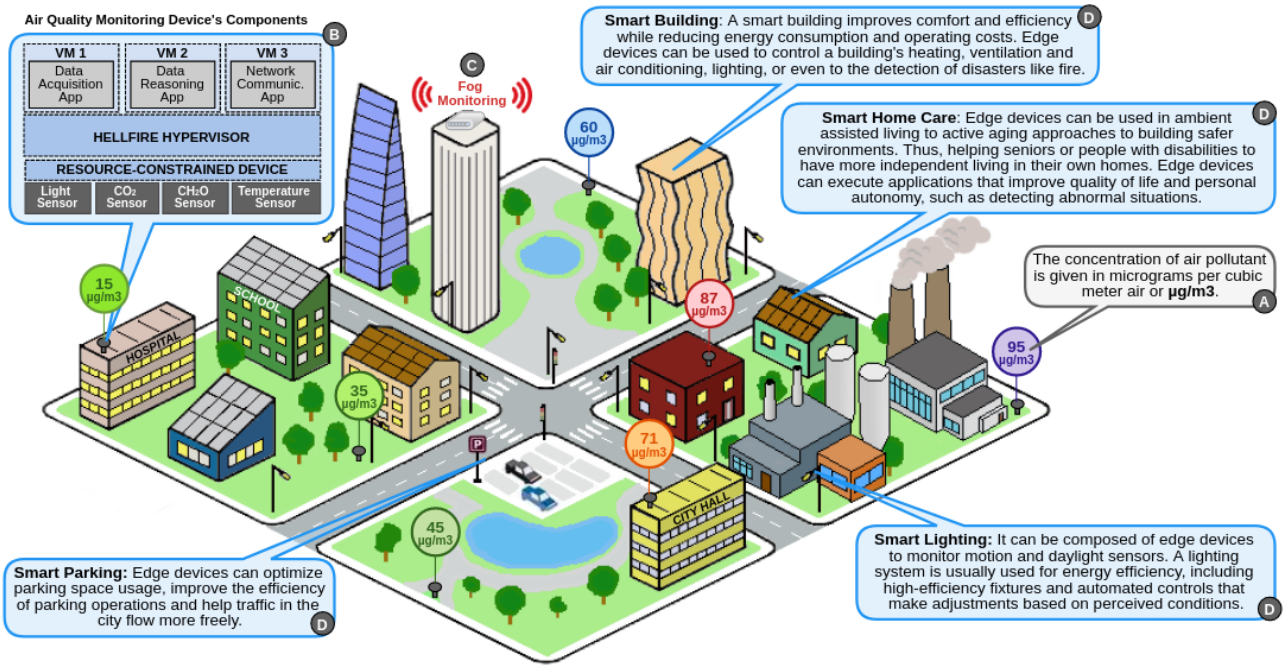



Figure 3.10 – Air quality monitoring scenario in urban areas. Additionally, smart cities applications that can benefit from the lightweight virtualization layer.

connected to some sensors simulated by software to monitor the environment: light, $CO_2$ (carbon dioxide), $CH_2O$ (methanal), and temperature. The devices' software components were implemented and are shown in Figure 3.10B. The air quality monitoring application was divided into three VMs: *data acquisition*, *data reasoning*, and *network communication*. They execute into each edge device to monitor the air quality in different parts of the city. The *data acquisition* VM receives raw data from sensors and sends them to the *data reasoning* VM through inter-VM communication. The *data reasoning* performs data filtering and aggregation to make decisions. For example, it can generate alarms for pollution peaks or aggregate and compact relevant data to reduce communication. Finally, the *network communication* VM implements the required network stack to send data to a fog/cloud monitoring device (Figure 3.10C).

Table 3.2 – Footprint results for the hypervisor in an Air Quality Monitoring Application (KB).

| Software | Storage | SRAM | Footprint |
|---|---|---|---|
| Hypervisor | 21 | 2 | **23** |
| Data Acquisition VM | 32 | 16 | **48** |
| Data Reasoning VM | 32 | 16 | **48** |
| Network Comm. VM | 128 | 64 | **192** |

The results for footprint are presented in Table 3.2. Sizes of 32 KB of flash and 16 KB of SRAM are required to communicate the *data acquisition* VM with sensors. The *data reasoning* application had the same values since it does not implement complex software stacks. *Network communication* was implemented using the picoTCP stack [8]. It required

128 KB for storage and 64 KB of SRAM to support TCP/IP and HTTP protocols. Thus, resulting in a total footprint of 311 KB (storage and SRAM) for the hypervisor and the three VMs execution.

Based on the results, the Hellfire hypervisor can be used in various Smart City applications, such as smart home care, smart building, smart lighting, and smart parking, as highlighted in Figure 3.10D. The use of resource-constrained edge devices in the described scenarios means the reduction of cost and power consumption.

## 3.8 SUMMARY

This chapter presented Hellfire, a lightweight virtualization hypervisor designed to deliver virtualization for small embedded devices. It is a type-1 hypervisor and supports isolation, real-time, and inter-VM communications. It is custom-made during compilation time, i.e., its data structure length is defined during the building process. The hardware support is used to avoid complex software implementation. The results show that it can be used in resource-constrained edge devices due to its small footprint, low virtualization overhead and inter-VM communication delay, and real-time support while enforcing security by separation. Thus, it was chosen to be used as the virtualization layer in the proposed security architecture.

# 4. SECURITY FOR EDGE DEVICES

This chapter presents the proposed security architecture for resource-constrained edge devices. It is described in Section 4.1. Also, Section 4.2 presents the evaluation, while Section 4.3 presents the related work.

## 4.1 SECURITY ARCHITECTURE DEFINITION

The proposed security architecture comprises two main mechanisms: the Chain-of-Trust Protection and the Virtualization Protection. They ensure the authenticity of the executed code, the integrity of the runtime states, and the confidentiality of elements stored in the persistent memory. The mechanisms are detailed in Sections 4.1.1 and 4.1.2.

### 4.1.1 CHAIN-OF-TRUST PROTECTION

A CoT is established after various secure boot stages, starting on the hardware and going to the highest level of software. An edge device must be designed to boot up only if the first piece of software to execute is cryptographically signed by a trusted entity (e.g., device vendor), and its signature matches with a root public key which is stored into the device. Figure 4.1 illustrates the device verification processes. First, the developer compiles the source code generating a device's binary image or firmware (1). Second, the hash digest of the binary image (a numeric code capable of identifying it uniquely) is calculated (2). Third, the hash digest is encrypted by a private key (3) from a public key-pair. The result is a signature that is placed in the device along with the firmware. Also, the developer must write the corresponding public key to the device's write-once memory. During startup, the hardware or the bootstrap software will recalculate the hash digest from the firmware on storage (4). The last step is to decrypt the signature using the public key (5) and compare it to the firmware's digest (6). Both must be the same since different digests indicate that the firmware was changed and the boot process is stopped. It is assumed that the attacker has no access to the developer's private key since he can use it to generate a compatible firmware.

Without the integrity of the initial boot program, any further integrity verification becomes pointless. Thus, a tamper-evident hardware module must protect the initial boot program. Specialized hardware can be used to store cryptographic keys and perform signature verification. The minimum requirements in terms of hardware are (see Figure 4.2-A): a One-Time-Programmable (OTP) memory to store the root public key and a primary boot-

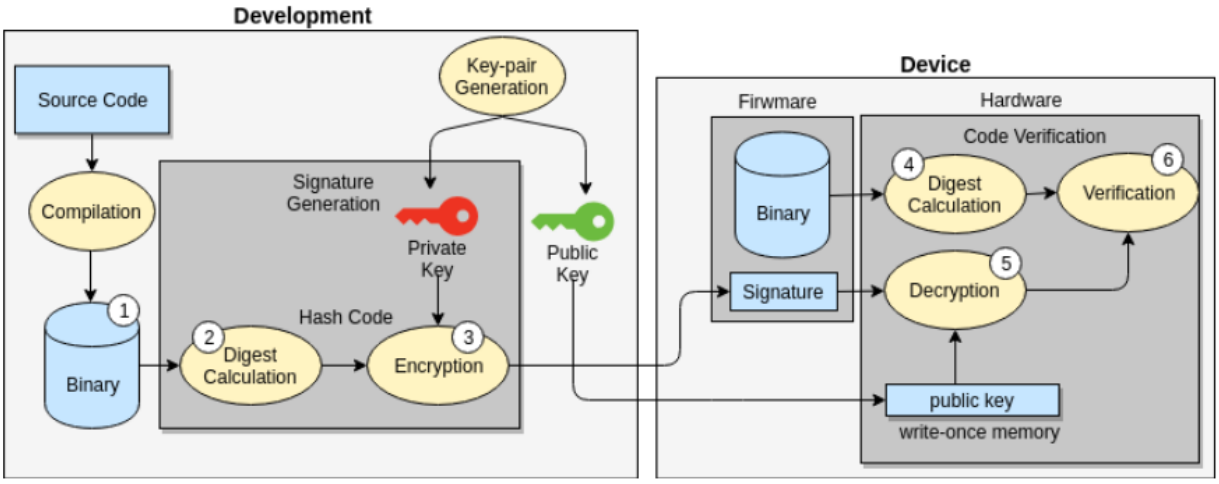



Figure 4.1 – Code signature generation and verification process.

loader (i.e., a piece of software that runs before any operating system) embedded in the hardware or stored in a protected memory that is able to verify the signature of the next boot stage. This scheme enables an RoT that should be used to ensure all running software's integrity and authenticity. By anchoring this RoT in the hardware, tampering becomes more difficult. Once RoT has been established, the initial software component should make identity and integrity checks with the virtualization layer in the boot chain (see Figure 4.2-B). If successful, then the same process will take place in the next boot stage of the CoT until the software stack is fully protected [130].

Since a virtualized environment can be composed of VMs from different providers, an additional secure boot verification level should be performed. Establishing a CoT between the virtualization layer and VMs is mandatory, allowing it to be securely anchored to the hardware and trusted for all operations (see Figure 4.2-C). For example, a vendor can mix third-party software (e.g., a robotic arm controller running in a virtualized application with software developed in-house). Virtualization makes the integration of different software environments easier since it is possible to keep them completely separated. The proposed security architecture allows the software providers to sign their VMs independently, i.e., each VM has a signature header (see Figure 4.2-D). The device vendor stores the provider's public key within the virtualization layer, allowing it to check the integrity/authenticity of each VM individually.

While virtualization allows software stacks from different providers to be integrated, individually signed VMs ensure code integrity, authenticity, and non-repudiation. Thus, the proposed architecture guarantees that the program was not modified (integrity). The VM is from the provider it claims to be from (authenticity). Moreover, if a VM causes a malfunction, the responsible party cannot deny it (non-repudiation).

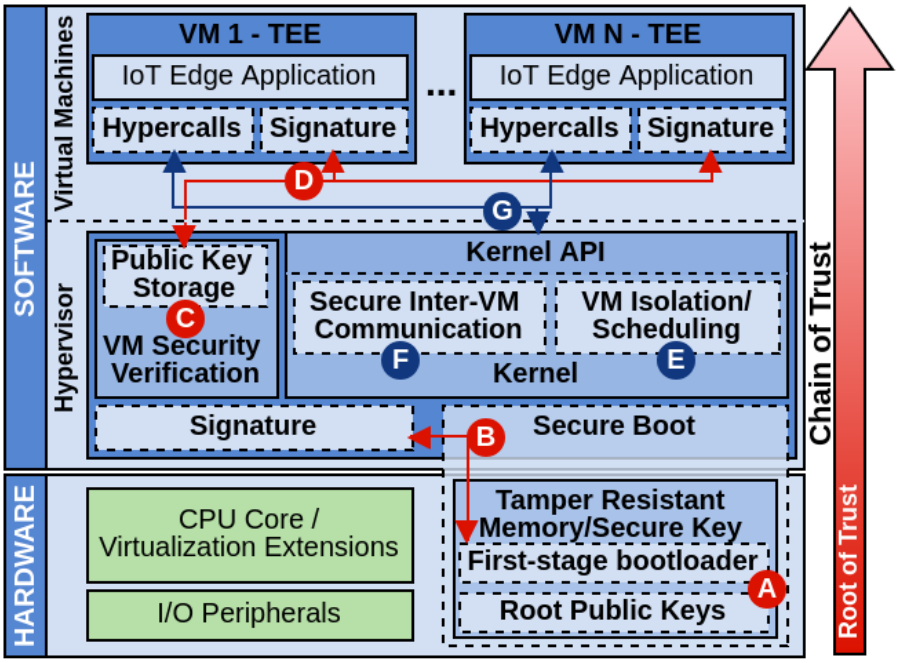



Figure 4.2 – Lightweight security architecture for resource-constrained edge devices.

### 4.1.2 VIRTUALIZATION PROTECTION

Once the CoT is established, the system is still vulnerable to runtime attacks, and virtualization plays an essential role in keeping TEEs [174]. Additionally, the use of hardware-assisted virtualization, i.e., specialized hardware to simplify the virtualization layer implementation and improve the system's performance, contributes to reducing the attack surface.

Many different approaches are discussed about hardware-assistance for virtualization layers [38] [113]. As a result, most modern embedded hypervisors use a hybrid approach, applying full-virtualization of the CPU (no modifications required in the virtualized software for basic functionalities), and para-virtualization (virtualized software must be modified) is required for extended services, such as inter-VM communication. Different virtualization layer subsystems improve device security, such as spatial isolation, temporal separation, and secure inter-VM communication.

The most common way of providing spatial isolation among VMs is through an MMU, a hardware block that provides virtual memory abstractions to the system (see Figure 4.2-E). Processors with hardware-assisted virtualization implement the second stage of MMU translation controlled by the virtualization layer. Essentially, the VMs can handle the hardware in the same way as in a non-virtualized system. However, they map virtual memory to intermediate physical memory. The virtualization layer is responsible for mapping intermediate physical memory addresses to physical memory, avoiding conflicts and ensuring separation between VMs. The second stage MMU translation drastically decreases the

virtualization layer's exceptions, making it suitable for resource-constrained edge devices and a small surface for attacks. Without proper separation, a running application can have access to all processors' address space, i.e., a malicious or malfunction code can touch any memory location.

Temporal separation guarantees the correct distribution of processor time among VMs according to their execution priorities (see Figure 4.2-E). Different authors have addressed the virtualization layer's scheduler as a way to improve temporal separation and to honor real-time constraints [33]. Additionally, system interrupts require attention since they interfere directly with the VM's execution. Hardware-assisted virtualization can help manage interrupts, allowing them to be redirected to VMs without intervention from the virtualization layer.

Virtualized IoT applications may require some level of interaction with each other and with the virtualization layer itself. Also, some applications require secure channels for sensitive information. Thus, an efficient and secure inter-VM communication mechanism should be available in the virtualization layer (see Figure 4.2-F). It is implemented as para-virtualized services, i.e., using a well-defined hypercall API (VM's calls to the virtualization layer) as presented in Figure 4.2-G. Thus, the virtualization layer works as a communication arbiter, copying messages from the sender to the destination application. The virtualization layer can check the size, the number of messages, and even deny forbidden communication.

Virtualization brings an advantage for keeping the integrity of TEEs: it allows the virtualization layer to monitor the behavior of the VMs, detecting malfunction caused by software errors or attacks. There are three ways of detecting a compromised VM:

- A VM tries to access memory outside the address space defined at design time.
- A VM invokes hypercalls which should not be called.
- A VM does not periodically reset a watchdog.

If the system detects one of these situations, the VM is restarted. Hence, two things can happen:

- If the VM code was compromised after the deployment, its hash signature will not match during the CoT phase, and it will not boot up.
- If an attacker was exploiting a vulnerability based on the malfunction, it will boot up and run as expected. However, the virtualization layer can emit alerts about the reset activities enabling developers to investigate the causes.

## 4.2 EVALUATION

The proposed architecture was evaluated on the same hardware as the hypervisor in Section 3.7. It is a resource-constrained device targeting IoT and embedded markets that supports hardware-assisted virtualization. The Hellfire hypervisor, described in Chapter 3, was used as a virtualization layer to support the previous section's mechanisms. The evaluation explores the architecture's suitableness for resource-constrained edge devices and security. It analyzes two metrics: footprint and performance. Performance results are based on an average of 1000 measurement runs. Also, a discussion around how the proposed architecture can achieve confidentiality, integrity, and availability was conduct.

### 4.2.1 FOOTPRINT AND PERFORMANCE ANALYSIS

The secure boot (first-stage bootloader) was implemented by storing it in the device's boot sector, which cannot be cloned to other devices. The secure boot mechanism implements digital signature using SHA256 for hash generation and two options for cryptographic algorithms: Elliptic Curve Digital Signature Algorithm (ECDSA) and RSA. The virtualization support was implemented using the Hellfire hypervisor on top of the secure boot, which implements the same cryptographic algorithms. The Hellfire Hypervisor hosted VMs entirely separated, which used the secure inter-VM communication mechanism to interact with each other. There are some advantages of being able to divide an IoT application to execute into smaller software components. First, each application is simpler and easier to implement and debug. Second, the only exposed application is the one that implements network communication thanks to the separation enforced by the hypervisor. Thus, potential attacks on this application will not expose sensitive data of other VMs or even allow the attacker to modify their contents.

Table 4.1 – Footprint results (KB).

| **Security Mechanism** | **Storage** | **SRAM** | **Footprint** |
|---|---|---|---|
| Chain-of-Trust Protection | 33 | 32 | **65** |
| Virtualization Protection | 26 | 32 | **58** |
| Secure VM | 32 | 16 | **48** |

Table 4.1 presents footprint results for each security mechanism. Storage presents the flash memory required to store the code. SRAM presents the memory required to run each mechanism. The footprint (Storage and SRAM), including Chain-of-Trust and virtualization mechanisms, requires just 123 KB, which illustrates the architecture's small footprint. It is worth reiterating that the resulting footprint includes both cryptographic algorithms and

the support for all virtualization features. For comparison purposes, off-the-self hypervisors, as Xen and KVM [38], require tens of megabytes. Xvisor [113], a hypervisor designed for embedded systems, requires up to 18 MB of RAM. Some aspects help the hypervisor to keep its small size: the paging subsystem is simplified (the number of VMs and their physical memory map are defined at design time), there is no filesystem implementation for devices with storage in a flash, there is no support for an interactive shell or a proper filesystem, and all configuration is defined at design time. To improve the footprint analysis, a secure bare metal VM was also evaluated. It is a simple monitoring application that receives data generated by the edge device. The VM requires 32KB for storage and 16 KB of SRAM for execution, resulting in 48 KB. It includes the VM signature. The total footprint required for the security mechanisms and the VM is 171 KB, which is a promising result for resource-constrained edge devices.

Table 4.2 – Performance Results for VM hash generation and signature verification (ms).

| VM Size (KB) | Hash Generation | Verification | | Total Time | |
|---|---|---|---|---|---|
| | SHA256 | ECDSA | RSA | ECDSA | RSA |
| 32 | 11.15 | 57.10 | 39.40 | **68.25** | **50.55** |
| 64 | 23.20 | 57.30 | 39.40 | **80.50** | **62.60** |
| 128 | 46.25 | 57.70 | 39.40 | **103.95** | **85.65** |
| 256 | 92.65 | 57.50 | 39.40 | **150.15** | **132.05** |

Table 4.2 presents the architecture performance for SHA256 hash generation from VMs stored in the flash memory and the ECDSA/RSA signature verification time of these VMs. For instance, a VM with a size of 64 KB takes 23.20 ms for hash generation, 57.30 ms for ECDSA verification, and 39.40 ms for RSA verification. Note that the hash generation time increases as the VM size is bigger. On the other hand, time for signature verification is independent of the VM's size since it is based on the VM's hash, which is generated by the SHA256 algorithm and always has the same size. In these experiments, the key length was 3072 bits for RSA and 256 bits for ECDSA, which are equivalent in cryptography strength. Based on the results, ECDSA was more suitable for resource-constrained edge devices than RSA considering the key length. However, RSA presented reduced execution time, which is also important in low-latency IoT applications. Hence, the best algorithm's decision depends on the requirements imposed by the application environment and the restrictions of resources in the device.

### 4.2.2 SECURITY ANALYSIS

In this section, the security of the proposed architecture is evaluated, discussing how it has achieved the three fundamental elements of CIA: confidentiality (preventing sen-

sitive device information from reaching the wrong people), integrity (avoiding improper device boot modification or destruction and ensuring its authenticity), and availability (ensuring reliable access to the edge device).

In the proposed architecture, an application executes in a VM and takes advantage of two security mechanisms: CoT and virtualization. The CoT protection checks the software integrity at boot-time. Thus, the CoT's main purpose is to deliver a verified software stack to the runtime environment, helping to prevent booting and tampering attacks. The virtualization layer is responsible for preventing possible runtime violations. It keeps the attacker confined to the compromised VM, minimizing the severity of the attack and ensuring other services' availability. Also, the hypervisor can detect a VM's misbehavior without complicated intrusion techniques, for example, the call for a hypercall not predicted or message exchanges not expected.

The hypervisor's spatial separation, provided by the hardware MMU, improves confidentiality. If an application attempts to access a memory region of any other application or peripherals, it will be stopped by the hypervisor. Security by separation, i.e., VM isolation, improves security over the following attacks (see Table 2.2) [123]: distributed denial-of-service, hyperjacking, malware injection, hidden-channel, and privilege escalation. Additionally, the small hypervisor footprint, resulting from its simplified subsystems, helps keep a small attack surface. Recent research showed that two forms of attacks, named Meltdown and Spectre, allow for breaking the memory isolation exposing sensitive data [94]. These attacks rely on out-of-order execution on modern processors. The proposed architecture prevents such attacks in two different ways:

- It avoids the use of affected processors since most of the embedded processors from MIPS and ARM families are not vulnerable.
- The CoT circumvents the execution of non-authorized software, a premise to the attacks.

## 4.3 RELATED WORK

This section presents an analysis of works that propose the use of trust mechanisms and/or embedded virtualization to provide security for edge devices. They are analyzed based on the security requirements presented previously (see Section 2.3.2), which are essential to provide a lightweight security architecture. Table 4.3 presents the works and they are described next.

- **TrustZone**: TrustZone [119] is a hardware-based security architecture for an SoC currently used in many smartphones. The TEE protects trusted hardware and software

Table 4.3 – Security for Edge Devices.

| Works | Ref | Year | Isolation | Trusted Boot | Key Protection | Secure VMs Communication |
|---|---|---|---|---|---|---|
| TrustZone | [15] | 2009 | ✓ | ✓ | ✓ | |
| SeCRet | [75] | 2015 | | | ✓ | ✓ |
| TEE | [37] | 2015 | ✓ | ✓ | ✓ | |
| Pahl *et al.* | [112] | 2016 | | ✓ | | |
| TrustShadow | [60] | 2017 | ✓ | ✓ | ✓ | |
| IIoTEED | [117] | 2017 | ✓ | ✓ | | |
| Morabito *et al.* | [101] | 2018 | | ✓ | | |
| **This work** | — | 2021 | ✓ | ✓ | ✓ | ✓ |

resources. Hardware-based mechanisms ensure that resources in the REE's untrusted OS, or normal world, cannot access secure world resources. Two main hardware features do this. During the boot process, a chain of trust is formed by verifying the trusted second-stage boot loader and trusted OS before execution. TrustZone uses a signature scheme based on RSA. The well-known weakness of TrustZone specification is the lack of authentication mechanisms in TrustZone's architecture when the REE needs to access secure resources.

- **SeCRet:** Jang *et al.* [75] propose SeCRet, a framework that builds secure communications between REE and TEE. SeCReT creates a session key to sign the messages transferred during inter-domain communication. To prevent the key from being exposed to an attacker who already compromised the REE kernel, SeCReT flushes the key from memory every time the processor switches into kernel mode. They evaluated SeCReT's performance on Arndale board that offers a Cortex-A15 at 1.7 GHz dual-core processor. Results show that enabling SeCReT creates a performance overhead of 16.41 percent (from 1642.5 $\mu$s to 1912.1 $\mu$s) with an input payload of 256 B.

- **TEE**: Dai *et al.* [37] present TEE. This architecture uses the Xen hypervisor to allow multiple VMs on a commodity cloud-end platform to enjoy DRTM-like secure execution environments. However, according to Sabt *et al.* [130], Dynamic Root of Trust for Measurement (DRTM) is not suitable for low-overhead applications. They evaluated TEE's performance with an Intel Core Duo processor running at 1.8 GHz and 2 GB RAM. Results show that the time to create the TEE Domain is 173 ms with one vCPU and 64 MB memory, the TEE kernel is 1.30 MB, and the time consumed for encryption is 436.9 ms (on average).

- **Pahl *et al.*:** The work presented in [112] analyzes how containers can provide a lightweight edge-to-cloud PaaS (Platform-as-a-Service). Its architecture is based on the Edge layer and uses containerization to provide security by separation. The authors used a Raspberry Pi board in their evaluations (700 MHz processor and 512

MB of RAM). They concluded that their work needs further investigation regarding the management of data, networks, and architecture.

- **TrustShadow:** Guan *et al.* [60] present TrustShadow, a system that takes advantage of TrustZone technology to isolate secure applications from untrusted ones. It also implements trust boot and secure key storage mechanisms. They evaluated TrustShadow on a Freescale i.MX6q ARM development board that integrates an ARM Cortex-A9 processor, 1GB DRAM, and 256KB RAM for evaluation. The latency overhead upon primitive operating system operations was 70 percent (on average).

- **IIoTEED:** Pinto *et al.* [117] propose a TrustZone-based architecture named IIoTTEED, which implements the basic building blocks of a TEE to protect edge devices. Confidentiality and availability are provided based on security by separation mechanisms, and integrity is provided at boot time through the trust boot process. The authors used a dual ARM Cortex-A9 running at 600 MHz for evaluation and concluded that IIoTEED must complement other critical security strategies to guarantee tight industrial security for devices.

- **Morabito *et al.*:** Authors in [101] presented the Edge Gateway-as-a-Service as an efficient and lightweight device able to pre-process IoT data using containers. The architecture consists of a single layer close to the edge device. They used security by separation to protect the applications inside containers. They used two Raspberry Pi boards in their evaluations (Quad-Core 900MHz/1GB of RAM and Quad-Core 1.2GHz/1GB of RAM). They concluded that virtualization technologies as containers have an almost negligible impact regarding performance in resource-constrained devices.

Analyzing the related works, most of them provide mechanisms for trust boot. Isolation and key protection features are the second most targeted, while only one work covers secure communication between TEEs. Although all the analyzed works provide security for edge devices, they differ in their architecture and the devices protected with such mechanisms. None of them presents significant results for resource-constrained edge devices.

## 4.4 SUMMARY

This chapter presented a lightweight security architecture for resource-constrained edge devices using chain-of-trust and virtualization protection mechanisms. The Hellfire hypervisor was used as virtualization layer. The chain-of-trust mechanism uses an RoT to build a CoT from the hardware until the application. Afterward, a trustworthy virtualization layer is booted up. The intrinsic virtualization characteristics, such as separation, ensure

protection during the VM's runtime states. Cryptography algorithms can be used to provide integrity/authenticity to the system. Results for footprint and overall performance were presented, showing that the architecture is feasible for resource-constrained edge devices. Finally, the proposed security architecture was compared to related work where advantages could be observed regarding its applicability to resource-constrained edge devices.

# 5. DEADLINE-AWARE TASK ASSIGNMENT AND SCHEDULING MECHANISM

This chapter presents the proposed deadline-aware task assignment and scheduling mechanism. Section 5.1 presents the problem formulation. The proposed mechanism is described in Section 5.2. Also, Section 5.3 presents the evaluation, while Section 5.4 presents the related work.

## 5.1 PROBLEM FORMULATION

In the task assignment and scheduling problem there are $n$ tasks $N = \{1, 2, ..., n\}$ to be assigned and scheduled to $m$ devices $M = E \cup F \cup C$ in a edge-fog-cloud architecture, where $E = \{1, 2, ..., e\}$ is a set of edge devices, $F = \{1, 2, ..., f\}$ is a set of fog devices, and $C = \{1, 2, ..., c\}$ is a set of cloud devices. The devices in $E$ are responsible for generating the tasks. Let $m_i$ be the device that generated task $i$. Each device $j \in M$ has $s_j$ storage capacity and $\lambda_j$ cores. The set of cores of a device $j \in M$ is denoted $C_j = \{1, 2, ..., \lambda_j\}$. Each core $k \in C_j$ has $g_{jk}$ computing capacity.

Each task $i \in N$ has size $\gamma_i \in \mathbb{N}$ which denotes the amount of storage that task $i$ requires on a device in order to be processed, i.e., task $i$ can be assigned to device $j$ if and only if device $j$ has $\gamma_i$ free storage available. A deadline $d_i$ is associated with each task $i \in N$ that denotes the finishing date, i.e., task $i$ must be processed until $d_i$ to be useful. The processing time of a task $i \in N$ on core $k$ is given by $p_{ik}$. This value can be calculated based on the length of the task $i$ and the processing power of core $k$. Thus, if we denote $s_i$ as the starting time of task $i$, we have that its completion time $c_i$ is equal to $s_i + p_{ik}$, such that $k$ is the assigned core. A task may be assigned to execute on the device that generated it, the fog, or the cloud. Thus, network latency should be considered. There is no latency to send a task to a core $k \in C_j$ of device $j \in E$. Therefore, the latency to send a task from device $j \in E$ to another device $j' \in (F \cup C)$ is defined as $\theta_{jj'}$.

The maximal storage capacity $s_j^{\max}$ of a device $j \in M$ should not be violated. Therefore, the sum of the tasks sizes assigned to all cores of device $j$ should not be greater than its storage capacity, i.e., for each $j \in M$ we have that

$$\sum_{k=1}^{C_j} \sum_{i=1}^{A_k} \gamma_i \leq s_j^{\max}. \tag{5.1}$$

The problem consists of defining a set $A_k \subseteq N$ which denotes the tasks assigned to core $k$, and a set $S_k$ that denotes the precedence relations $(i, j)$ between the tasks assigned to core $k$. Therefore, for any two arbitrary tasks $i, j \in N$ where $i$ directly precedes $j$ on core

$k$ we have that $(i, j) \in S_k$. Since the processing of two tasks assigned to the same core can not overlap, we have that for each $(i, j) \in S_k$, then

$$c_i \leq s_j. \tag{5.2}$$

The objective is to *minimize* the total tardiness $T_{\max} = \max_{\forall i \in 1,...,n}\{0, c_i - d_i\}$.

The following constraints are also considered. A task should be assigned to no more than one core. Devices' storage capacities should not be exceeded. A core must process a task entirely before starting a new task. Table 5.1 summarizes the constants and sets that define an instance of the task assignment and scheduling problem in an edge-fog-cloud architecture.

Table 5.1 – Summary of problem data (constants and sets) that define an instance of the problem.

| Symbol | Interpretation |
|---|---|
| $n$ | number of tasks |
| $m$ | number of devices |
| $N = \{1, 2, ..., n\}$ | set of tasks |
| $M = E \cup F \cup C$ | set of devices |
| $E = \{1, 2, ..., e\}$ | set of edge devices |
| $F = \{1, 2, ..., f\}$ | set of fog devices |
| $C = \{1, 2, ..., c\}$ | set of cloud devices |
| $m_i, i \in N$ | device that generated task $i$ |
| $s_j, j \in M$ | storage capacity in device $j$ |
| $\lambda_j, j \in M$ | number of cores in device $j$ |
| $C_j = \{1, 2, ..., \lambda_j\}$ | set of cores in device $j$ |
| $g_{jk}, j \in M, k \in C_j$ | computing capacity of a core $k$ in device $j$ |
| $\gamma_i, i \in N$ | size of task $i$ |
| $d_i, i \in N$ | deadline of task $i$ |
| $p_{ik}, i \in N, k \in C_j$ | processing time of task $i$ on core $k$ |
| $s_i, i \in N$ | starting time of task $i$ |
| $c_i, i \in N$ | completion time of task $i$ |
| $\theta_{jj'}, j \in E, j' \in (F \cup C)$ | latency to send a task from device $j$ to device $j'$ |
| $s_j^{\max}, j \in M$ | maximal storage capacity of a device $j$ |
| $A_k \subseteq N, k \in C_j$ | set of tasks assigned to core $k$ |
| $S_k \subseteq N, k \in C_j$ | set of precedence relation between the tasks assigned to core $k$ |

## 5.2 PROPOSED MECHANISM

The proposed algorithm is named DTAS-Edge (Deadline-aware Task Assignment and Scheduling mechanism at the Edge). It executes on edge devices. Thus, they are responsible for finding the best assignment and scheduling for each generated task.

An architectural overview of DTAS-Edge is depicted in Figure 5.1. It is composed of a deadline checking mechanism and a waiting queue. The **Deadline Checking** is responsible for finding the best assignment and scheduling for a task on a core. This component is located only on the edge device. Thus, the deadline checking can be done practically simultaneously as the task generation, avoiding spending time sending the task to be checked in a higher layer device. The result of the deadline checking indicates which core is most suitable to process the task.

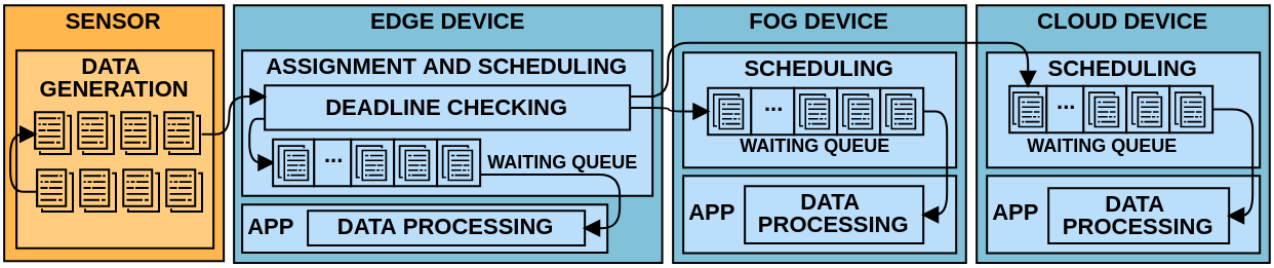



Figure 5.1 – DTAS-Edge in an Edge-Fog-Cloud architecture.

Each core of a device has a **Waiting Queue** for tasks sequencing (i.e., the order in which tasks are processed), as shown in Figure 5.1. The algorithm schedules a task to the queue of the device to which it was assigned. The waiting queue follows the FCFS (First Come, First Served) concept, i.e., the first task that is queued is the first one that will be processed. It is depicted in Figure 5.2. When the core becomes available, the first task in the queue starts processing.

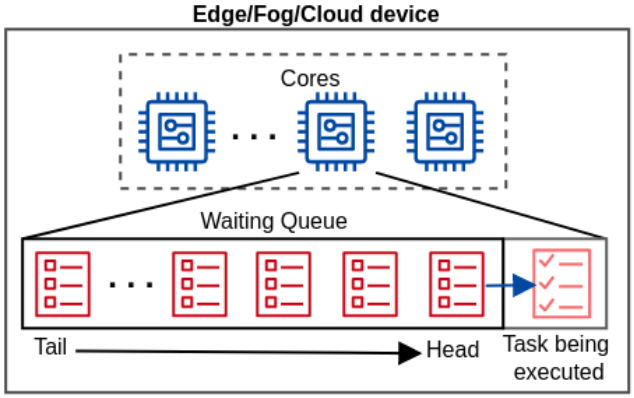



Figure 5.2 – The waiting queue of a core in an edge/fog/cloud device.

The algorithm that describes the proposed mechanism is presented in Algorithm 5.1. The algorithm's definition is based on the fact that there is a mechanism that prevents race conditions in the network.

---

**Input:** $N, E, F, C, M, C_j$
**Output:** $A_k, S_k$

---

1: **for** each $i \in N$ **do**
2: Set $k \leftarrow 0$, and $c_i^{\text{best}}$ an extremely large number
3: Set $s_i^{\text{tmp}} \leftarrow 0$, and $c_i^{\text{tmp}} \leftarrow 0$
4: Let $a_1, a_2, a_3$ where $a_1 = (m_i)$, and $a_2$ and $a_3$ are permutations of $F, C$
5: **for** $b \leftarrow 1$ to 3 **do**
6: **for** each $j \in a_b$ **do**
7: **for** each $k' \in C_j$ **do**
8: $i' \leftarrow \text{argmax}_{i'' \in A_{k'}} \{c_{i''}\}$
9: $s_i^{\text{tmp}} \leftarrow \max\{0, c_{i'}\}$
10: $c_i^{\text{tmp}} \leftarrow s_i^{\text{tmp}} + p_{ik'} + \theta_{m_i j}$
11: **if** $(c_i^{\text{tmp}} \leq d_i) \wedge (c_i^{\text{tmp}} < c_i^{\text{best}}) \wedge ((\sum_{k''=1}^{C_j} \sum_{i'=1}^{A_{k''}} \gamma_{i'}) + \gamma_i \leq s_j^{\max})$ **then**
12: $k \leftarrow k'$
13: $c_i^{\text{best}} \leftarrow c_i^{\text{tmp}}$
14: **end if**
15: **end for**
16: **end for**
17: **if** $k \neq 0$ **break** and go to line 19
18: **end for**
19: **if** $k \neq 0$ **then**
20: $A_k \leftarrow A_k \cup i$
21: $s_i \leftarrow s_i^{\text{tmp}}$
22: $c_i \leftarrow c_i^{\text{tmp}}$
23: $S_k \leftarrow S_k \cup (i', i)$
24: **end if**
25: **end for**

---

Algorithm 5.1 – DTAS-Edge algorithm.

The algorithm checks the best assignment for each generated task (line 1). The first check involves only the edge device that generated the task (to avoid spending time during transmission over the network). The algorithm checks task $i$ for each core $k'$ of each device $j$ (lines 6 and 7). First, the processing time of the tasks assigned to core $k'$ is obtained

(line 8), also called the waiting time of the core's $k'$ queue. The expected starting time of task $i$ in a core $k'$ is obtained in line 9. This value can be 0, if the core's queue is empty. Then, the expected completion time of task $i$ is calculated by adding the starting time, processing time, and network latency to transfer the task $i$ from $m_i$ to device $j$ (line 10). Three conditions are tested in line 11. First, if the expected completion time of task $i$ on core $k'$ is lower than or equal to the task $i$ deadline. Second, if the last found expected completion time is lower than the best one found so far. Third, if the device $j$ has enough capacity to receive task $i$. If true for all conditions, then core $k'$ represents the best assignment for task $i$. In that case, task $i$ is assigned to core $k$ (line 20), receives the starting and completion time calculated values (lines 21 and 22), and is added in the scheduling queue (i.e., waiting queue) of core $k$, keeping the assignment order.

If no core is chosen, the same process repeats for fog and cloud layers. However, the task is tested for all cores of each layer. This process happens in lines 4 and 5. Suppose there are none available after testing all cores in the architecture. In that case, the algorithm ends with a "resource unavailable" error, i.e., there is no core in the architecture that can meet the task deadline and device capacity requirements, and the task is discarded. However, if there is a core with enough capacity and that meets the deadline requirement, the task is assigned and scheduled to it (lines 20 and 23).

## 5.3 EVALUATION

In this section, a set of experiments was conducted to evaluate DTAS-Edge. The tests were divided into three experiments:

- **Experiment 1:** The comparison of DTAS-Edge with two other approaches to verify which is the best layer in an edge-fog-cloud architecture to perform the deadline checking;

- **Experiment 2:** The comparison of DTAS-Edge with other literature algorithms.

- **Experiment 3:** The evaluation and comparison of DTAS-Edge simulating a real-world application.

The simulator and settings used in all tests are presented in Section 5.3.1. Specific settings and metrics evaluated in each experiment are detailed in the respective section.

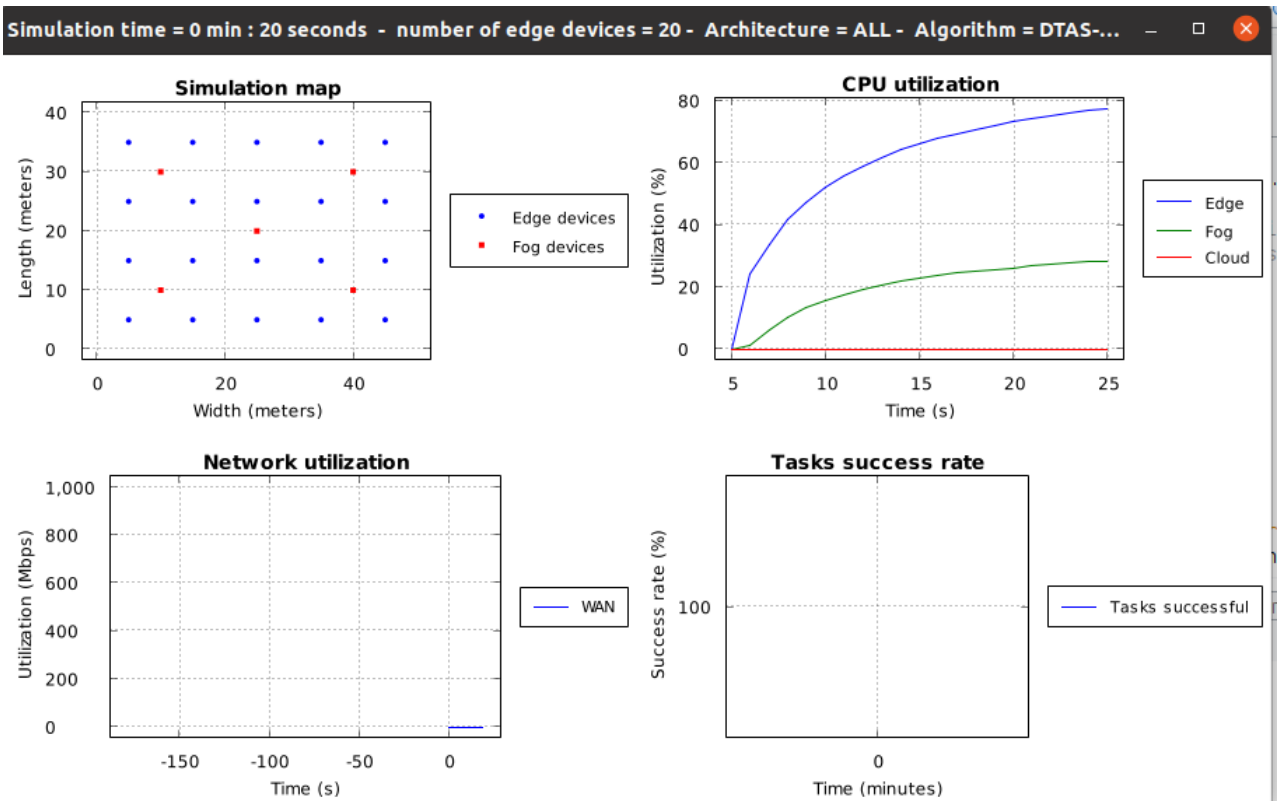



Figure 5.3 – PureEdgeSim simulation view.

### 5.3.1 ENVIRONMENT SETUP

The proposed mechanism was evaluated using PureEdgeSim [92], a simulation toolkit based on CloudSim Plus designed to simulate Cloud, Fog, and Edge Computing environments. The decision to use PureEdgeSim is mainly due to the possibility of extending it with orchestration algorithms, allowing the creation of strategies to assign and schedule tasks among cores of devices. Also, PureEdgeSim allows evaluating resource management strategies' performance in terms of network usage, latency, resource utilization, energy consumption, among others. A view of PureEdgeSim during a simulation is presented in Figure 5.3.

There are other popular simulators in literature but with some limitations regarding features presented in this work. The iFogSim, a CloudSim-based simulation framework designed to simulate Fog Computing environments [61], provides a static topology and allows extension only around the application's placement strategies. Simulating large-scale scenarios requires adding and linking devices one by one, which is inconvenient, involves much effort, and time-consuming. EmuFog [91] is an emulation framework for Fog Computing environments. It lacks a generic interface and cannot deal with global metrics, such as response delay. EdgeCloudSim [140] is another CloudSim-based simulator for Edge Com-

puting that addresses some iFogSim limitations. Moreover, it can not execute tasks locally on edge devices, limiting its use to some Edge Computing scenarios.

The computer used during simulations was configured with Ubuntu 20.04 LTS (64-bit), Quad-Core 2.3 GHz and 6GB of RAM. The chosen parameters simulate computation-intensive and time-sensitive smart city applications, such as real-time data processing applications (heart rate monitoring, traffic lights, augmented reality, among others). They are hard to handle due to the quick turnaround requirements of ultra-short time and large amounts of computation necessary [44]. Such applications may require a response in less than a second [134]. The simulation parameters used in all experiments are presented in Table 5.2.

Table 5.2 – Simulation Parameters.

| **Parameter** | **Value** |
|---|---|
| Simulation Time (min) | 30 |
| WAN bandwidth (Mbps) | 1000 |
| WLAN bandwidth (Mbps) | 1000 |
| WAN Latency (seconds) | 0.2 |
| Task Deadline (seconds) | 1 |
| Time during simulation (min) | 00-05 → 05-10 → 10-20 → 20-25 → 25-30 |
| Task request size (KB) | 17-22 → 22-27 → 27-32 → 22-27 → 17-22 |
| Task result size (KB) | 12-17 → 17-22 → 22-27 → 17-22 → 12-17 |
| Task length for processing (MI) | 475-525 → 775-825 → 1175-1225 → 775-825 → 475-525 |

It is worth pointing out some specifics of the simulator. The bandwidth values for WAN and WLAN are divided by the cores of each device. For example, considering a 1000 Mbps bandwidth, a fog device with eight cores will have 125 Mbps for each core. The WLAN bandwidth is used between edge devices and fog devices, while the WAN bandwidth is used between fog and cloud devices. A limitation of the simulator is that the network latency can only be set for the WAN. Since the latency on the WLAN is generally low, a zero value was assumed. Regarding the task deadline, a value of 1 second was set, which is the smallest value accepted by the simulator. Nevertheless, it is a real value usually observed in low-latency applications [64] [125] [13].

Regarding the tasks transmitted over the network, they have one size before being processed (task request size) and another size after being processed (task result size). These sizes are given in KB and are considered for network usage calculations. The “task length” attribute refers to the size of the task for processing. This value is given in MI (Millions of instructions). These three attributes were changed during runtime to simulate peak moments in an application. Also, such values were chosen randomly with a uniform distribution over a predefined interval. Therefore, the last four rows of Table 5.2 can be read as “during minutes zero to five”:

- The task request size is a random value between 17 and 22;

- The task result size is a random value between 12 and 17;
- The task length is a random value between 475 and 525.

And so on for the remaining time intervals. Thus, the simulator was extended to simulate peak times in applications.

Table 5.3 presents the number of tasks and devices used for experiments 1, 2, and 3.

Table 5.3 – Number of tasks and devices for Experiments 1, 2 and 3.

| Parameter | Experiments 1 and 2 | Experiment 3 |
|---|---|---|
| Task Generation Rate (device/min) | 960 | 1800 |
| Total tasks (all simulation) | 1440000 | 1080000 |
| Number of Edge devices | 50 | 20 |
| Number of Fog devices | 5 | 4 |
| Number of Cloud devices | 1 | 1 |

The devices' configurations used in all experiments are presented in Table 5.4. The attribute "cores" represents the number of cores available for processing in a device. Each core has a processing capacity, as shown in the second row of the table, given in MIPS (Millions of instructions per second). RAM and storage capacities are also defined.

Table 5.4 – Devices' configurations.

| Parameter | Edge Device | Fog Device | Cloud Device |
|---|---|---|---|
| Cores | 4 | 8 | 32 |
| Processing capacity (MIPS per core) | 2400 | 8000 | 13000 |
| RAM (GB) | 1 | 8 | 16 |
| Storage (GB) | 0.512 | 1000 | 10000 |

The DTAS-Edge mechanism was evaluated against five other algorithms. They were implemented in the simulator and evaluated under the same simulation parameters. The algorithms are:

- **DTAS-Fog**: It is a variation of the mechanism proposed in this thesis. For comparison purposes, it executes on fog devices. Thus, its assignment strategy prioritizes fog, edge, and cloud devices, respectively. The factors used to decide the best assignment are the same as in the DTAS-Edge mechanism.
- **DTAS-Cloud**: It is a variation of the mechanism proposed in this thesis. For comparison purposes, it executes on cloud devices. Thus, its assignment strategy prioritizes cloud, fog, and edge devices, respectively. The factors used to decide the best assignment are the same as in the DTAS-Edge mechanism.

- **Round-Robin (RR)**: It executes on edge devices. It iterates through the whole list of devices and chooses one by one successively to the assignment. It does not consider the waiting time of tasks and does not prioritize any device over the others.
- **Prioritized Task Scheduling (PTS)**: It executes on edge devices. The assignment is based on the core with fewer waiting tasks considering a pre-defined priority. The order of priority is Fog, Edge, and Cloud, in which Fog is the highest priority.
- **Increase Lifetime (IL) [92]**: It executes on edge devices. The assignment decision is to the core with more computing capacity and fewer waiting tasks considering all cores in the architecture.

The experiments evaluate the following aspects:

- **Task Deadline Violation**: The percentage of tasks successfully executed but failed due to the violation of their deadlines. A lower value is better.
- **Resource Utilization**: The percentage average CPU utilization of devices. A higher value means a more efficient use of resources.
- **Network Usage**: The amount of data traffic over the network between edge, fog, and cloud in GB. A lower value means less network usage.

### 5.3.2 EXPERIMENT 1 - DTAS-EDGE ANALYSIS

The first experiment compares DTAS-Edge with two similar algorithms: DTAS-Fog and DTAS-Cloud. The factors used for deadline checking are the same for all algorithms (core's waiting queue, network latency, and task completion time). However, each one was deployed on a different layer. Thus, the priority of task execution is of the device that is checking the task for assignment. It would be unfair, for example, to send the task to the cloud to check its deadline and bring it back to the edge for execution. Therefore, with this setting, a task checked in the cloud has priority to be executed in the cloud. If there are no resources available, it is checked for lower layers, according to the previous section's priorities. The same logic is applied to the DTAS-Fog mechanism.

The results represent the average for three runs and are shown in Figure 5.4 (see Table A.1 for all results). The DTAS-Cloud and DTAS-Fog algorithms had 24.5% and 16.6% of unprocessed tasks, respectively. It means that the algorithms could not find any available core to process the tasks meeting their deadlines, and they were discarded. This happens because the tasks' deadlines are low, i.e., there is not enough time to send the tasks to other layers and process them meeting the deadline. For the DTAS-Edge, all tasks were

processed. Regarding deadline violations, the DTAS-Cloud and DTAS-Fog algorithms had 0.3% and 1.4% of tasks that could not meet the deadline. On the other hand, the DTAS-Edge mechanism did not have any deadline violations.

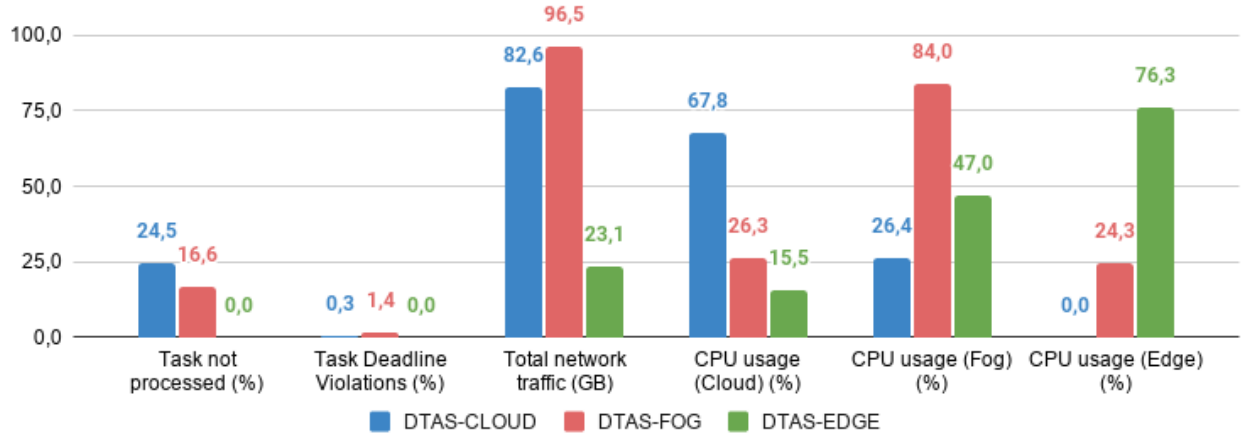



Figure 5.4 – Comparing DTAS-Edge, DTAS-Fog, and DTAS-Cloud.

The algorithm that showed the highest network usage was DTAS-Fog since when it cannot allocate tasks to devices in its layer, it needs to send data to the edge or cloud, which generates a lot of task traffic. The DTAS-Cloud algorithm also showed high network usage. However, it was not higher than DTAS-Fog because the cloud has more resources for task execution than the fog, so it could assign most of the tasks to its cores, avoiding more transmissions on the network. The best performance for network usage was observed in DTAS-Edge, which transmitted only 23 GB, approximately 60 GB less than the other two algorithms' average. As expected, DTAS-Edge keeps most of the processing at the edge devices, making transmissions to other layers happen only when the deadline cannot be reached at the edge.

Finally, the average CPU usage for each algorithm was evaluated. As expected, DTAS-Cloud keeps most of the processing in the cloud (67.8%). On the other hand, no tasks were processed on the edge devices. DTAS-Fog obtained an average CPU usage on fog devices of 84%, as expected. The DTAS-Edge had its highest average CPU usage on the edge devices with 76.3%. The fog layer had 47%, while the cloud had only 15.5%.

Analyzing the results, DTAS-Edge had no deadline violations, the lowest network usage and the best resource utilization. Thus, the best place to perform deadline checking is at edge devices.

### 5.3.3 EXPERIMENT 2 - COMPARISON WITH LITERATURE ALGORITHMS

The second experiment compares DTAS-Edge to three literature algorithms: Round Robin (RR), Prioritized Task Scheduling (PTS), and Increase Lifetime (IL). The results rep-

resent the average for three runs and are shown in Figure 5.5 (see Table A.1 for all results). The algorithms assigned all generated tasks, i.e., no task was discarded. Regarding deadline violations, a good performance was observed for all algorithms except RR. It happens because RR does not consider a waiting queue of tasks to decide the assignment. Thus, most of the tasks violate the deadline. Such violation happens mainly on edge devices, where there are fewer resources to process tasks. Also, the number of edge devices in this experiment is high (50 devices). Thus, most of the tasks are assigned to them. As the queue becomes long, many tasks cannot be processed within the expected deadline. On the other hand, IL and PTS algorithms consider each core's waiting queue's to make the assignment. This strategy reflects positively in the results since the algorithms had only 0.3% and 0.2%, respectively, of deadline violations.

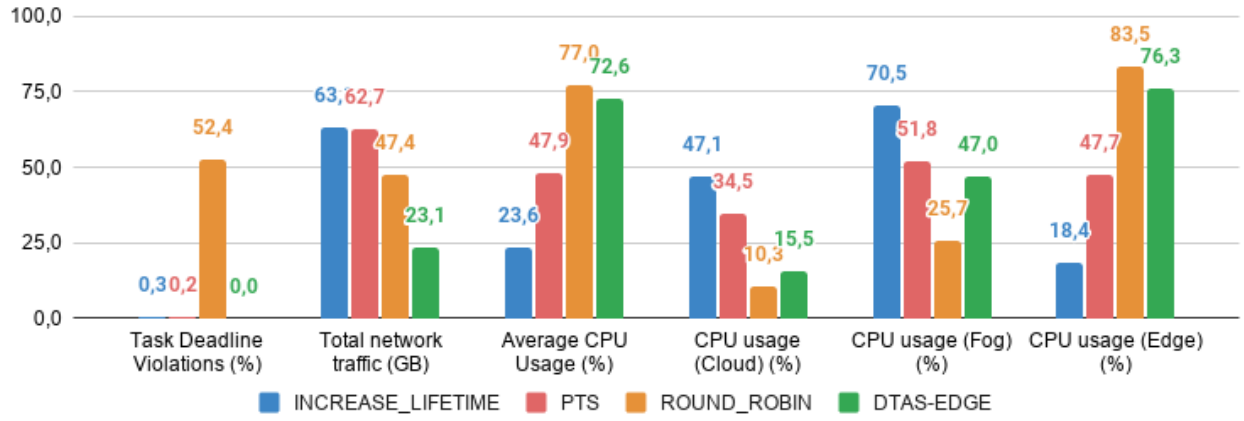



Figure 5.5 – Comparison with literature algorithms.

The best performance was observed in DTAS-Edge, which had no deadline violations. This result is due to several factors considered when deciding the core in which the task should be assigned, as described in Section 5.2. The edge-first approach also contributes to this result. It allows most of the tasks to be processed in edge devices, avoiding the time wasted during transmission over the network.

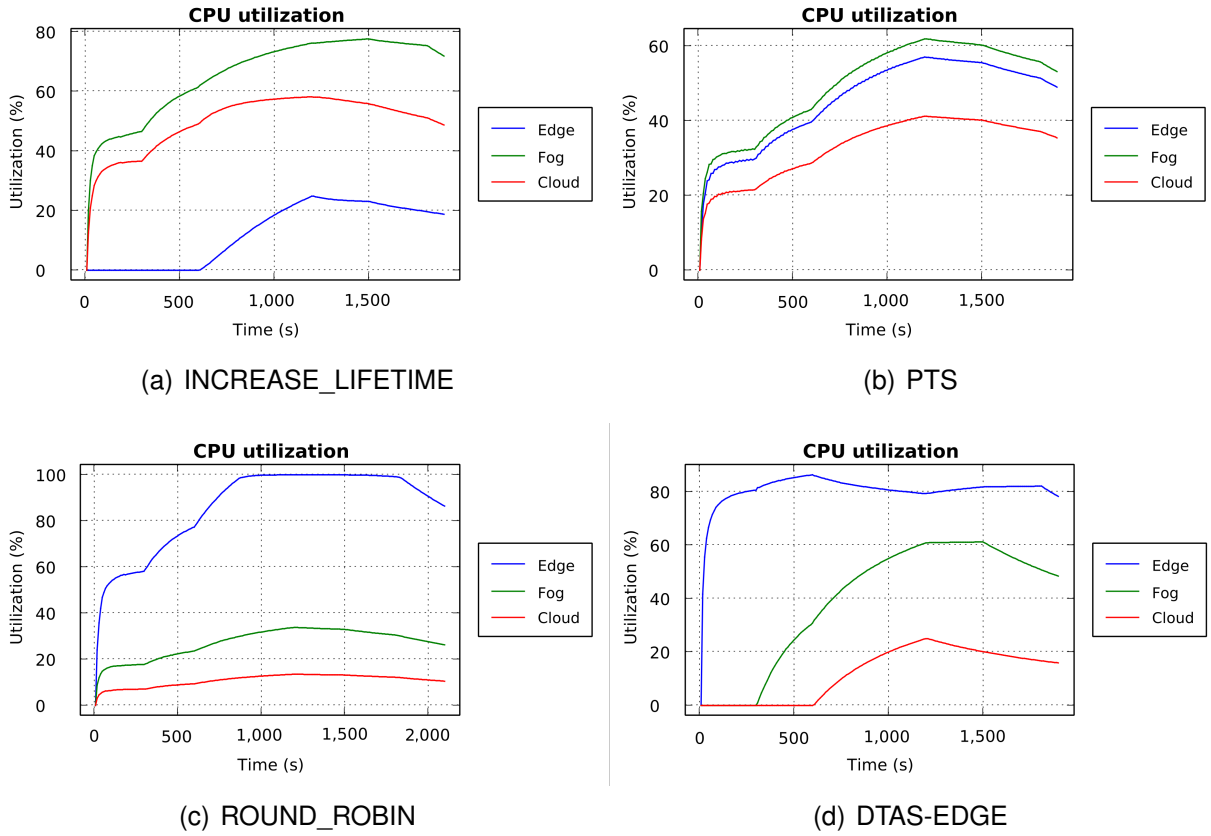
Regarding network usage, the DTAS-Edge had the lowest network traffic, only 23.1 GB. Again, the task assignment considering the edge-first approach is responsible for the positive result, i.e., if a task can be processed meeting the deadline in the device that generated it, it does not need to be send to higher layers. Regarding RR, since it assigns tasks to one core at a time, regardless of the core's waiting queue, most of them are assigned to the edge, reducing the number of tasks transmitted over the network. However, this does not represent a reduction in deadline violation, as discussed earlier. On the other hand, despite presenting more network usage, the IL and PTS algorithms had fewer deadline violations. However, it is essential to mention that DTAS-Edge could process the same tasks meeting all deadlines and avoiding almost 40 GB of tasks from being transmitted over the network compared to IL and PTS performances.

The best performance regarding average resource utilization is noted in the RR, which assigns most of the tasks to edge devices, contributing to the overall average resource utilization. On the other hand, the overload on edge devices causes many deadline violations, as discussed earlier. The second-best resource utilization was observed in the DTAS-Edge, which prioritizes tasks to edge devices. The average CPU usage of edge devices was 76.3%, contributing to the overall average of 72.6%. The PTS algorithm showed an average resource utilization of 47.9%. Its strategy prioritizes the assignment of tasks to fog, edge, and cloud, respectively. Among the compared algorithms, it is the one that best divides the tasks among the layers. However, lower utilization of edge devices makes the overall average resource utilization not as high as in RR and DTAS-Edge algorithms. Finally, as IL algorithm has a task assignment strategy that chooses cores by processing capacity available, most of the tasks are assigned to fog and cloud devices. It reflects negatively on the average resource utilization since the scenario is composed of several edge devices. As they are under-utilized, they decrease the overall average resource utilization to 23.6%. On the other hand, such strategy does not negatively impact the tasks' deadlines since the violation rate observed was low.

The graphs in Figure 5.6 allow an analysis of how each algorithm works regarding resource utilization during a peak moment in an application. The beginning of peak moments causes changes in the task assignment priority between layers. For the IL algorithm, all tasks are assigned to fog and cloud layers at the beginning. When the peak moment reaches its maximum point (minutes 10 to 20), the algorithm assigns tasks to the edge devices to handle all the necessary processing. Simultaneously, note that task processing in the fog increases while processing in the cloud decreases. When the peak moment is over, the assignment of tasks to edge devices starts to decrease.

The PTS algorithm had a more homogeneous assignment of tasks among the architecture layers. Again, an increase can be noted in resource utilization as the peak moment starts. However, during the whole simulation, including the peak moment, the assignment of tasks among the layers remains in the same proportion, prioritizing fog, edge, and cloud devices, respectively.

The RR algorithm, on the other hand, despite assigning tasks among all layers since the beginning of the simulation, assigns most of them to edge devices. As its strategy does not check the core's queue to make the assignment, the edge devices are overloaded with tasks, especially during the peak time. Although there is a small increase in the cloud resource utilization during the peak time (minutes 10 to 20), it is not due to the assignment of more tasks to the cloud, but to the increase of tasks' length, which takes longer to process. The same happens at fog. That is, the algorithm makes the assignment of tasks regardless of the peak scenario. It reflects negatively on the deadline violation reduction, as discussed earlier.




(a) INCREASE_LIFETIME (b) PTS

(c) ROUND_ROBIN (d) DTAS-EDGE

Figure 5.6 – Resources Utilization Graphs.

Finally, since DTAS-Edge is based on an edge-first approach, as long as there are processing capacity and storage in the cores' queues, the tasks are assigned to edge devices. When peak time starts and the tasks' length increases, other layers are considered for task assignment — first the fog, and second the cloud. Thus, CPU usage slows down on edge devices and increases on fog and cloud devices during the peak time. When it is over, the edge devices receive most of the tasks for processing again.

The DTAS-Edge mechanism had the best results compared to other algorithms. The strategy of using fog and cloud during peak times allowed no deadline violations during the experiment. Simultaneously, network utilization remained the lowest, while resource utilization was the second-best observed.

### 5.3.4 EXPERIMENT 3 - SMART SURVEILLANCE REAL WORLD APPLICATION

The third experiment compares the proposed mechanism with literature algorithms regarding a real-world application. Compared to the previous experiments, the main change is the increase of the task generation rate per minute per device from 960 to 1800, representing 30 tasks generated per second per device. This adaptation approximates the simulation

of a camera that generates 30 frames per second. The simulation had 20 edge devices, four fog devices, and one cloud device to handle all the generated tasks.

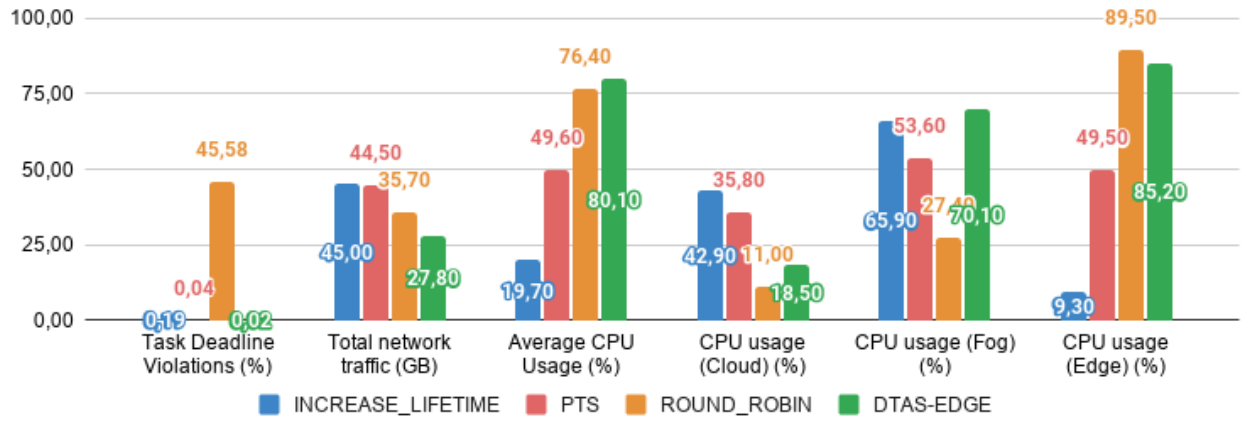



Figure 5.7 – Results for a Smart Surveillance Application

The results represent the average for three runs and are shown in Figure 5.7 (see Table A.1 for all results). They are proportionally similar to the results obtained in the two previous experiments. Again, DTAS-Edge obtained the best performance for the three analyzed aspects: lower number of deadline violations, lower network usage, and better average resource utilization. A more detailed discussion regarding the number of deadline violations is needed since DTAS-Edge presented deadline violations for the first time. All the violations happened for tasks assigned to the cloud. Since the task generation rate is high, the network is more overloaded with tasks, and they take longer to be transmitted. This, along with the low task deadline, contributed to this result. The result is still considered good since the number of tasks processed meeting the deadline was 99.98%.

## 5.4 RELATED WORK

This section presents a literature review of works that present solutions for task assignment and scheduling in the context of edge-fog-cloud computing. They were analyzed according to the following criteria [163] [6]:

- **Architecture:** In centralized (C) scheduling, a single scheduler makes all the mappings [6]. The implementation is straightforward, but the problem with this architecture is that when the scheduler fails, the whole system fails, too. In the distributed (D) scheduling, the mapping of resources to tasks is made via distributed schedulers, meaning that the workload is distributed between several schedulers [6]. Distributed scheduling, compared with the centralized method, enjoys higher scalability. Therefore, it fits the distributed computing environments such as edge computing.

- **Assignment:** A decision on task assignment may result in [86]: *Edge (E)*: the tasks are assigned locally, at edge devices; *Fog (F)*: the tasks are assigned to be processed by fog devices; and *Cloud (C)*: the tasks are assigned to be processed by cloud devices.
- **Scheduling:** The order in which tasks are assigned for execution on a core or device.

The analyzed works are presented in Table 5.5 and described next:

Table 5.5 – Task assignment and scheduling in edge-fog-cloud.

| Solutions | Ref | Year | Architecture | Assignment | Scheduling |
|---|---|---|---|---|---|
| Wang et al. | [164] | 2016 | D | F | MCFEDF |
| Fan et al. | [49] | 2017 | C | F,C | FIFO |
| Choudhari et al. | [34] | 2018 | D | F,C | Priority |
| Fang et al. | [50] | 2019 | D | F | - |
| Sharma and Saini | [133] | 2019 | D | F,C | EDF |
| Dedas | [95] | 2020 | D | F,C | - |
| **This work** | — | 2021 | D | E,F,C | FCFS |

- **Wang et al.** [164]: Authors study the impact of both task assignment and scheduling on the overall operational cost of multi-cloudlet based mobile edge clouds. They aim to optimize per-task cost and ensure the quality of experience by enforcing hard deadlines for offloaded tasks through joint task assignment and scheduling in multi-cloudlet environments. Upon the arrival of an offloaded task, they determine which cloudlet this task will be assigned and how execution sequence the tasks assigned to the same cloudlet will follow. Most Critical First with EDF (MCFEDF) algorithm is used to decide such sequence. They developed a discrete-time multi-cloudlet simulator in Python to evaluate their approach. They compare the proposed algorithm with a baseline approach where tasks are assigned to the closest cloudlets, and the scheduling of tasks is done following an FCFS manner complemented with EDF for tasks that arrive simultaneously. Experiments showed that the admission rate is significantly improved (up to 30%) in the proposed approach. The proposed assignment and scheduling algorithm can help reduce the average per-task cost (up to 20%).
- **Fan et al.** [49]: The authors presented a deadline-constrained task scheduling framework for IoT systems with a joint fog and cloud computing architecture. The goal is to maximize the total net profit received by service providers through task scheduling and placement while meeting application deadline requirements and satisfying resource capacity constraints in both fog and cloud networks. A scheduling algorithm based on Ant Colony Optimization (ACO) is presented to maximize the total net profit for the fog service provider. Numerical results show that the solution outperforms FCFS and Min-min algorithms.

- **Choudhari et al.** [34]: Authors presented a task scheduling algorithm based on priorities in the fog computing environment. The objective is to reduce the overall response time and decrease the total cost. A task is analyzed regarding its deadline. If it cannot be executed in the remaining time, it is rejected. Otherwise, it is placed in the appropriate priority queue based on its priority level (high, medium, or low). The tasks are first processed in the fog layer based on their priority levels. Only when all the fog devices are saturated that tasks are propagated to the cloud layer. The scheduling algorithm was simulated on the Cloud Analyst Simulator, which is built on top of CloudSim. The results indicate that the overall response time is decreased, along with the cost.

- **Fang et al.** [50]: Authors propose a dynamic online policy for task scheduling problem in fog computing. They aim to reduce the application delay and network usage by making tasks executing in fog devices. In their approach, fog devices are responsible for scheduling tasks among fog and cloud devices. The decision is based on the task's expected completion time. To evaluate the algorithm, the authors use the iFogSim toolkit. They compared the approach with a static task scheduling strategy. The results showed that the proposed approach performance could reduce 33.07% of latency and 66.22% of network usage compared with a static module.

- **Sharma and Saini** [133]: Authors proposed delay-aware scheduling and load balancing architecture within fog environments. They made use of the EDF task scheduling algorithm for scheduling tasks in the fog. In case a fog device fails to get the resource needed, the request is forwarded to the cloud tier. They validated their proposed work over a real-time VSOT application using iFogSim. The performance was evaluated based on response time, scheduling time, energy consumption rate, delay, and load balancing rate.

- **Dedas** [95]: Authors propose Dedas, an online dispatching and scheduling algorithm to maximize the number of tasks that meet the deadlines and minimize the average completion time of the tasks by jointly scheduling of networking and computing resources. In a fog or cloud device, Dedas inserts the new task in a position or replaces an existing task if there is a deadline violation due to adding the new task to generate a feasible schedule with the minimum average completion time. Based on the schedule method, Dedas dispatches the new task to the fog device such that the number of completed tasks is maximized and the average completion time is minimized. A task is dispatched to the cloud only if fog resources can not satisfy its requirements. In this work, the authors adopt a real-world data-trace from the Google cluster for large-scale emulations. Testbed experiments demonstrate that the deadline miss ratio of Dedas is stable for online tasks, which is reduced by up to 60% compared with state-of-the-art methods. Moreover, Dedas performs well in minimizing the average task completion time.

In [163] authors propose a taxonomy of task offloading in edge-cloud environments to investigate and classify related research articles. In [6] authors present a survey on the task scheduling algorithms proposed by different researchers for the cloud-fog environment, their advantages and disadvantages, and various tools and issues regarding the scheduling methods, and their restrictions were discussed. The results indicated that about 58% of the scheduling algorithms use static scheduling. The other 42% use dynamic scheduling, and the delay metric is the most important parameter considered in most studies with a share of nearly 17%.

The work developed by Wang et al. [162] propose a heuristic task scheduling method to address the problem of optimizing deadline violations for executing tasks in heterogeneous computational environments. It does not fit among the related works as it does not consider an edge-fog-cloud architecture, but its contribution is relevant. The algorithm iteratively schedules a task to the first core. The accumulated slack time of all scheduled tasks is minimum until the core cannot finish any task and executes tasks with the earliest deadline first in each core to execute as many tasks as possible in a core. Experiment results based on a real-world trace show that the method has up to 100% less task violations and has the best performance in resource efficiency optimization in overall, compared with eight classical and state-of-the-art heuristic methods.

The literature also presents several works that address the task assignment and scheduling problem in an edge-fog-cloud context, but without mentioning deadline violations [74] [114] [171] [169] [172] [144] [64] [14] [125] [78]. This work differs from existing works by distributing the proposed mechanism in edge devices and assigning tasks to the three layers of an edge-fog-cloud architecture, prioritizing edge, fog, and cloud. Also, the proposed mechanism considers challenges imposed by peak moments of low-latency applications to decide the best assignment and scheduling for a task, which contributes to finishing as many tasks as possible meeting their deadlines and is not addressed in the analyzed works.

## 5.5 SUMMARY

This chapter presented a deadline-aware mechanism to assign and schedule tasks in an edge-fog-cloud architecture. The mechanism has the edge as a priority to assign the tasks. If there are no cores available, they are assigned to the fog or cloud layers. Each edge device is responsible for checking and assigning its generated tasks. The mechanism was evaluated against five other approaches. The proposed mechanism was better than DTAS-Fog and DTAS-Cloud regarding the best place to do the deadline checking. Also, the comparison results with IL, PTS, and RR algorithms showed that DTAS-Edge reduces deadline violations when considering peak moments of low-latency IoT applications. Finally, the proposed mechanism was compared to related works, where advantages could be observed

regarding the assignment strategy and the consideration of peak moments in low-latency IoT applications.

# 6. FINAL CONSIDERATIONS

This chapter presents the final considerations related to this work. First, the contributions and publications related to the developed work are presented. Finally, the conclusions and future work.

## 6.1 CONTRIBUTIONS

Among the contributions presented in this work, it is possible to identify the main contributions in the following items:

- The review of the background technologies used for providing security for resource-constrained edge devices (Chapter 2). Published in [156] [153] [11] [104].
- The definition of a taxonomy for the security provision for edge devices encompassing attacks and security key requirements (Chapter 2). Published in [156] [104].
- The review of the state-of-the-art in both security for edge devices and lightweight embedded virtualization. (Chapters 3 and 4). Published in [156] [151].
- The definition of a lightweight security architecture for resource-constrained edge devices (Chapters 4). Published in [156].
- The use of a lightweight hypervisor to enable edge computing and security in resource-constrained edge devices (Chapter 3 and 4). Published in [155].
- The execution of tests to validate the security architecture (Chapter 4). Published in [156].
- The review of the state-of-the-art of task assignment and scheduling in an edge-fog-cloud architecture. (Chapters 2 5).
- The definition of a deadline-aware mechanism to assign and schedule tasks in an edge-fog-cloud architecture considering peak times of low-latency IoT applications (Chapter 5).
- The execution of tests to validate the DTAS-Edge mechanism including the simulation of a real-world application (Chapter 5).
- This work was awarded with an Australia–Americas PhD Research Internship Program scholarship to be developed partially in Australia. It was developed in collaboration with the Deakin University. This collaboration has strengthened the relation between

PUCRS and Deakin University, Geelong. This work contributed to the Growth and Innovation Project delivered by the Australian Academy of Science.

- The publication of 12 (twelve) scientific papers during the Ph.D. as shown in Section 6.2. The publications are 4 (four) international journals, 3 (three) book chapters, and 5 (five) international conferences.

## 6.2 PUBLICATIONS

During the advancement of this research work and achieving the set of previously stated research questions and objectives, results were presented in papers published or submitted to be published. Table 6.1 presents the papers chronologically (from the newest to the oldest).

Table 6.1 – Papers published during the PhD degree.

| Ref. | Work Title | Venue and Publisher | Year | Impact Factor |
|---|---|---|---|---|
| [155] | A Lightweight Virtualization Model to Enable Edge Computing in Deeply Embedded Systems | Journal of Software: Practice and Experience | 2021 | 1.78 |
| [43] | Context information sharing for the Internet of Things: A survey | Elsevier Computer Networks | 2020 | 3.11 |
| [104] | Privacy and security of Internet of Things devices | Real-Time Data Analytics for Large Scale Sensor Data (Academic Press) | 2020 | — |
| [154] | Evaluating the DTLS Protocol from CoAP in Fog-to-Fog Communications | IEEE International Conference on & Service-Oriented System Engineering (SOSE) | 2019 | — |
| [156] | Lightweight Security Architecture Based on Embedded Virtualization and Trust Mechanisms for IoT Edge Devices | IEEE Communications Magazine | 2019 | 11.05 |
| [42] | Providing Context-Aware Security for Environments Through Context Sharing Feature | IEEE International Conference On Trust, Security And Privacy In Computing And Communications (TrustCom) | 2018 | — |
| [41] | Context Interoperability for IoT through an Edge-centric Context Sharing Architecture | IEEE Symposium on Computers and Communications (ISCC) | 2018 | — |
| [40] | A Sensing-as-a-Service Context-Aware System for Internet of Things Environments | IEEE Consumer Communications & Networking Conference (CCNC) | 2017 | — |
| [152] | Evaluating the Use of TLS and DTLS Protocols in IoT Middleware Systems Applied to E-health | Consumer Communications & Networking Conference (CCNC) | 2017 | — |
| [151] | The Role of Lightweight Approaches Towards the Standardization of a Security Architecture for IoT Middleware Systems | IEEE Communications Magazine | 2016 | 11.05 |
| [153] | Security Challenges in 5G-Based IoT Middleware Systems | Internet of Things (IoT) in 5G Mobile Technologies (Springer International Publishing) | 2016 | — |
| [11] | Middleware Technology for IoT Systems: Challenges and Perspectives Toward 5G | Internet of Things (IoT) in 5G Mobile Technologies (Springer International Publishing) | 2016 | — |

The paper in [11] introduces definitions and implementation of middleware systems to IoT applications, showing how it manages devices and provides interoperability. The papers in [151] [152] [153] [156] [154] [104] [155] explain how security can be reached in the IoT. The paper in [40] explains details of the context-aware feature of IoT systems and how it can be used in a system/architecture. The studies [41] and in [43] relate to the context sharing provision. The study [42] relates to the context-aware security provision.

## 6.3 REVISITING THE HYPOTHESES AND RESEARCH QUESTIONS

This thesis investigated two hypotheses: (i) the integration of embedded virtualization and trust mechanisms can provide a lightweight security architecture to improve the security of resource-constrained edge devices, nonetheless, keeping a small memory footprint; and (ii) a deadline-aware task assignment and scheduling mechanism can reduce the number of deadline violations in low-latency IoT applications during peak times.

For the first hypothesis, the definition of a lightweight security architecture presented in Chapter 4 shown that the use of a lightweight hypervisor (described in Chapter 3) and trust mechanisms can improve the security of resource-constrained edge devices from attacks presented in Section 2.3.1. The evaluation presented in Sections 3.7 and 4.2 shown that the Hellfire hypervisor can provide security for resource-constrained edge devices while keeping a small footprint.

For the second hypothesis, the definition of a deadline-aware task assignment and scheduling mechanism is presented in Chapter 5. The evaluation presented in Section 5.3 shown that the proposed mechanism can reduce the number of deadline violations considering peak times in low-latency applications compared to existing algorithms.

In the Chapter 2, the **Research Question** *"What are the most common security threats that could compromise edge devices, and what requirements should be considered to improve their security?"* was answered by the definition of the taxonomy of "security for edge devices" which presents challenges to overcome, relevant attacks, how they can violate software, and the key security requirements to provide a lightweight security architecture.

In the Chapters 3 and 4, the **Research Question** *"How to define a lightweight security architecture with a high-security level but keeping a small footprint with tens of kilobytes?"* was answered by defining a security architecture that integrates a lightweight virtualization layer and trust mechanisms. The hellfire hypervisor is used to keep a small footprint, enabling such architecture in resource-constrained edge devices.

In the Chapter 5, the **Research Question** *"Where should a task be assigned to have a better chance of being processed meeting its deadline?"* was answered by defining the DTAS-Edge mechanism, which considers factors such as cores' waiting queues, task

completion time, and network latency. Also, it evaluates all assignment possibilities from the edge to the cloud, which leads us to the best assignment between available cores, prioritizing edge, fog, and cloud.

In the Chapter 5, the **Research Question** *"Which architecture layer should the assignment and scheduling mechanism be deployed to reduce the number of deadline violations?"* was answered by the definition of the DTAS-Edge mechanism and the results observed in the evaluation. A task should be checked for assignment right after it is generated, so no time is wasted during transmission over the network before checking the assignment possibilities, i.e., the best place for DTAS-Edge mechanism is in edge devices.

## 6.4 CONCLUSION

The large amount of data generated by edge devices in IoT applications brings two main challenges. First, edge devices are increasingly targeted by attackers, who try to compromise an entire network of devices or gain access to data. Second, edge devices cannot handle all data generated when facing peak times in low-latency IoT applications. This thesis proposed solutions to both challenges.

Regarding security, this work defined a lightweight security architecture for resource-constrained edge devices. It was validated using the hellfire hypervisor, a virtualization layer capable of providing security by separation and keeping a small footprint. Furthermore, trust mechanisms such as trust boot have been incorporated into the device boot and the hypervisor, creating a CoT and allowing secure applications to execute on virtual machines.

Compared to related work, the security architecture has the smallest footprint and meets all the security requirements presented in Section 2.3.2. Although there are similar approaches in the literature that combine virtualization and trust mechanisms, none of them presented such a small footprint, making the proposed architecture feasible in the most resource-constrained edge devices.

Regarding scheduling, this work defined a deadline-aware task assignment and scheduling mechanism named DTAS-Edge. It assigns tasks right after their generation, allowing the best assignment among cores and prioritizing edge, fog, and cloud, respectively. Such a mechanism can reduce the violation of deadlines that usually happen during peak times of low-latency IoT applications.

Compared with other algorithms, the proposed mechanism proved efficient in reducing the number of deadline violations for several configurations. Regarding related work, the proposed mechanism stands out for having its scheduling mechanism placed on edge devices in an edge-first approach. According to the results obtained, it is the best way to reduce deadline violations during peak times in low-latency applications.

Based on the conducted study, it is possible to conclude that processing most of the data on edge devices is a trend and can bring benefits such as decreased deadline violations. However, we must consider the need for a secure device since attackers increasingly target them. It is essential to care about each device's characteristics to allow for a consolidation of processing centered on the edge.

## 6.5 FUTURE WORK

As future work, there are the following possibilities:

- The area of security is more and more in focus within edge computing. The use of Machine Learning also appears as a trend in the Computer Science area. It could be used as a more precise technique to identify attacks on edge devices. The big challenge is how to incorporate machine learning algorithms in the context of resource-constrained devices. There are already some works in the area, but they need further investigation to achieve significant results.
- Another major challenge in security considering resource-constrained devices is how to secure the communication between such devices. Lightweight Blockchains may be a way to integrate these devices securely and reliably. However, although works are exploring these solutions, it is a field that needs more research.
- Regarding data processing in edge devices, there are several low-latency applications where the order of the generated data should be taken into account by the scheduler. An improvement of DTAS-Edge to fit these applications' requirement is a future work that needs further investigation.

# REFERENCES


[1] Abdul-Ghani, H. A.; Konstantas, D.; Mahyoub, M. “A Comprehensive IoT Attacks Survey based on a Building-blocked Reference Model”, *International Journal of Advanced Computer Science and Applications*, vol. 9–3, Mar 2018, pp. 1–19.

[2] Aguiar, A.; Hessel, F. “Virtual hellfire hypervisor: Extending hellfire framework for embedded virtualization support”. In: International Symposium on Quality Electronic Design, 2011, pp. 1–8.

[3] Ahmed, E.; Ahmed, A.; Yaqoob, I.; Shuja, J.; Gani, A.; Imran, M.; Shoaib, M. “Bringing Computation Closer toward the User Network: Is Edge Computing the Solution?”, *IEEE Communications Magazine*, vol. 55–11, Nov 2017, pp. 138–144.

[4] Ai, Y.; Peng, M.; Zhang, K. “Edge computing technologies for Internet of Things: a primer”, *Digital Communications and Networks*, vol. 4–2, Apr 2018, pp. 77–86.

[5] Alam, M.; Rufino, J.; Ferreira, J.; Ahmed, S. H.; Shah, N.; Chen, Y. “Orchestration of Microservices for IoT Using Docker and Edge Computing”, *IEEE Communications Magazine*, vol. 56–9, Sep 2018, pp. 118–123.

[6] Alizadeh, M. R.; Khajehvand, V.; Rahmani, A. M.; Akbari, E. “Task scheduling approaches in fog computing: A systematic review”, *International Journal of Communication Systems*, vol. 33–16, Aug 2020, pp. 1–36.

[7] Alrawais, A.; Alhothaily, A.; Hu, C.; Cheng, X. “Fog Computing for the Internet of Things: Security and Privacy Issues”, *IEEE Internet Computing*, vol. 21–2, Mar 2017, pp. 34–42.

[8] Altran. “picoTCP”. Source: http://picotcp.altran.be/, Dec 2020.

[9] Alves, M. P.; Delicato, F. C.; Santos, I. L.; Pires, P. F. “LW-CoEdge: a lightweight virtualization model and collaboration process for edge computing”, *World Wide Web*, vol. 23–2, Nov 2020, pp. 1127–1175.

[10] Alwarafy, A.; Al-Thelaya, K. A.; Abdallah, M.; Schneider, J.; Hamdi, M. “A Survey on Security and Privacy Issues in Edge-Computing-Assisted Internet of Things”, *IEEE Internet of Things Journal*, vol. 8–6, Mar 2021, pp. 4004–4022.

[11] Amaral, L. A.; Matos, E.; Tiburski, R. T.; Hessel, F.; Lunardi, W. T.; Marczak, S. “Middleware Technology for IoT Systems: Challenges and Perspectives Toward 5G”. Springer International Publishing, 2016, chap. 15, pp. 333–367.

[12] Ameen, M. A.; Liu, J.; Kwak, K. “Security and privacy issues in wireless sensor networks for healthcare applications”, *Journal of Medical Systems*, vol. 36–1, Mar 2012, pp. 93–101.

[13] Ananthanarayanan, G.; Bahl, P.; Bodík, P.; Chintalapudi, K.; Philipose, M.; Ravindranath, L.; Sinha, S. “Real-Time Video Analytics: The Killer App for Edge Computing”, *Computer*, vol. 50–10, Oct 2017, pp. 58–67.

[14] Apat, H. K.; s. Compt, B.; Bhaisare, K.; Maiti, P. “An Optimal Task Scheduling Towards Minimized Cost and Response Time in Fog Computing Infrastructure”. In: International Conference on Information Technology, 2019, pp. 160–165.

[15] ARM Security Technology. “Building a Secure System using TrustZone Technology”, Technical Report, ARM, 2009, 108p.

[16] Atzori, L.; Iera, A.; Morabito, G. “The Internet of Things: A survey”, *Computer Networks*, vol. 54–15, Oct 2010, pp. 2787–2805.

[17] Axual. “Top things to know about real-time data processing”. Source: https://axual.com/top-things-to-know-about-real-time-data-processing/, Dec 2020.

[18] Banafa, A. “Eight Trends of the Internet of Things in 2018”. Source: https://iot.ieee.org/newsletter/january-2018/eight-trends-of-the-internet-of-things-in-2018, Jan 2018.

[19] Barham, P.; Dragovic, B.; Fraser, K.; Hand, S.; Harris, T.; Ho, A.; Neugebauer, R.; Pratt, I.; Warfield, A. “Xen and the Art of Virtualization”. In: Symposium on Operating Systems Principles, 2003, pp. 164–177.

[20] Barik, R. K.; Dubey, A. C.; Tripathi, A.; Pratik, T.; Sasane, S.; Lenka, R. K.; Dubey, H.; Mankodiya, K.; Kumar, V. “Mist Data: Leveraging Mist Computing for Secure and Scalable Architecture for Smart and Connected Health”, *Procedia Computer Science*, vol. 125–1, Feb 2018, pp. 647–653.

[21] Basavaraj, D.; Tayeb, S. “Limitations and Challenges of Fog and Edge-Based Computing”. In: International Internet of Things, Electronics and Mechatronics Conference, 2021, pp. 1–6.

[22] Bertino, E.; Islam, N. “Botnets and Internet of Things Security”, *Computer*, vol. 50–2, Feb 2017, pp. 76–79.

[23] Bhatia, M.; Sood, S. K. “Exploring temporal analytics in fog-cloud architecture for smart office healthcare”, *Mobile Networks and Applications*, vol. 24–4, Jan 2019, pp. 1392–1410.

[24] Biswas, A. R.; Giaffreda, R. “IoT and cloud convergence: Opportunities and challenges”. In: World Forum on Internet of Things, 2014, pp. 375–376.

[25] Bonomi, F.; Milito, R.; Zhu, J.; Addepalli, S. “Fog Computing and Its Role in the Internet of Things”. In: Workshop on Mobile Cloud Computing, 2012, pp. 13–16.

[26] Bruneo, D.; Distefano, S.; Longo, F.; Merlino, G.; Puliafito, A.; D’Amico, V.; Sapienza, M.; Torrisi, G. “Stack4Things as a fog computing platform for Smart City applications”. In: International Conference on Computer Communications, 2016, pp. 848–853.

[27] Buerki, R.; Rueegsegger, A.-K. “Muen - An x86/64 Separation Kernel for High Assurance”, Technical Report, University of Applied Sciences Rapperswil, 2013, 99p.

[28] Byers, C. C. “Architectural Imperatives for Fog Computing: Use Cases, Requirements, and Architectural Techniques for Fog-Enabled IoT Networks”, *IEEE Communications Magazine*, vol. 55–8, Aug 2017, pp. 14–20.

[29] Cao, H.; Wachowicz, M. “An edge-fog-cloud architecture of streaming analytics for internet of things applications”, *Sensors*, vol. 19–16, Aug 2019, pp. 1–32.

[30] Carvalho, G.; Cabral, B.; Pereira, V.; Bernardino, J. “Edge computing: current trends, research challenges and future directions”, *Computing*, vol. 103–5, Jan 2021, pp. 993–1023.

[31] Chen, M.; Hao, Y.; Hu, L.; Hossain, M. S.; Ghoneim, A. “Edge-CoCaCo: Toward Joint Optimization of Computation, Caching, and Communication on Edge Cloud”, *IEEE Wireless Communications*, vol. 25–3, Jul 2018, pp. 21–27.

[32] Chen, X.; Wang, L.; Wang, C.; Jin, R. “Predictive offloading in mobile-fog-cloud enabled cyber-manufacturing systems”. In: Industrial Cyber-Physical Systems, 2018, pp. 167–172.

[33] Cheng, K.; Bai, Y.; Wang, R.; Ma, Y. “Optimizing Soft Real-Time Scheduling Performance for Virtual Machines with SRT-Xen”. In: International Symposium Cluster, Cloud and Grid Computing, 2015, pp. 169–178.

[34] Choudhari, T.; Moh, M.; Moh, T.-S. “Prioritized Task Scheduling in Fog Computing”. In: Southeast Conference, 2018, pp. 1–8.

[35] Choy, S.; Wong, B.; Simon, G.; Rosenberg, C. “A hybrid edge-cloud architecture for reducing on-demand gaming latency”, *Multimedia Systems*, vol. 20–5, Apr 2014, pp. 503–519.

[36] Coppolino, L.; D’Antonio, S.; Mazzeo, G.; Romano, L. “A comprehensive survey of hardware-assisted security: From the edge to the cloud”, *Internet of Things*, vol. 6–6, Jun 2019, pp. 1–17.

[37] Dai, W.; Jin, H.; Zou, D.; Xu, S.; Zheng, W.; Shi, L.; Yang, L. T. “TEE: A virtual DRTM based execution environment for secure cloud-end computing”, *Future Generation Computer Systems*, vol. 49–8, Aug 2015, pp. 47–57.

[38] Dall, C.; Nieh, J. “KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor”. In: International Conference on Architectural Support for Programming Languages and Operating Systems, 2014, pp. 333–348.

[39] De Donno, M.; Dragoni, N.; Giaretta, A.; Mazzara, M. “AntibIoTic: Protecting IoT Devices Against DDoS Attacks”. In: International Conference in Software Engineering for Defence Applications, 2018, pp. 59–72.

[40] de Matos, E.; Amaral, L. A.; Tiburski, R. T.; Schenfeld, M.; Hessel, F.; de Azevedo, D. “A Sensing-as-a-Service Context-Aware System for Internet of Things Environments”. In: Annual Consumer Communications & Networking Conference, 2017, pp. 725–728.

[41] de Matos, E.; Tiburski, R. T.; Amaral, L. A.; Hessel, F. “Context Interoperability for IoT Through an Edge-Centric Context Sharing Architecture”. In: Symposium on Computers and Communications, 2018, pp. 667–670.

[42] de Matos, E.; Tiburski, R. T.; Amaral, L. A.; Hessel, F. “Providing Context-Aware Security for IoT Environments Through Context Sharing Feature”. In: International Conference on Trust, Security and Privacy in Computing and Communications, 2018, pp. 1711–1715.

[43] de Matos, E.; Tiburski, R. T.; Moratelli, C. R.; Filho, S. J.; Amaral, L. A.; Ramachandran, G.; Krishnamachari, B.; Hessel, F. “Context information sharing for the Internet of Things: A survey”, *Computer Networks*, vol. 166–1, Jan 2020, pp. 1–19.

[44] Deng, X.; Li, J.; Liu, E.; Zhang, H. “Task allocation algorithm and optimization model on edge collaboration”, *Journal of Systems Architecture*, vol. 110–11, Nov 2020, pp. 1–14.

[45] Deogirikar, J.; Vidhate, A. “Security attacks in IoT: A survey”. In: International Conference on IoT in Social, Mobile, Analytics and Cloud, 2017, pp. 32–37.

[46] Dogo, E. M.; Salami, A. F.; Aigbavboa, C. O.; Nkonyana, T. “Taking Cloud Computing to the Extreme Edge: A Review of Mist Computing for Smart Cities and Industry 4.0 in Africa”. Springer International Publishing, 2019, chap. 7, pp. 107–132.

[47] Dolui, K.; Datta, S. K. “Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing”. In: Global Internet of Things Summit, 2017, pp. 1–6.

[48] Dutta, J.; Roy, S. “IoT-fog-cloud based architecture for smart city: Prototype of a smart building”. In: International Conference on Cloud Computing, Data Science Engineering - Confluence, 2017, pp. 237–242.

[49] Fan, J.; Wei, X.; Wang, T.; Lan, T.; Subramaniam, S. “Deadline-Aware Task Scheduling in a Tiered IoT Infrastructure”. In: Global Communications Conference, 2017, pp. 1–7.

[50] Fang, J.; Li, K.; Ma, A. “Latency aware online tasks scheduling policy for edge computing system”, *Journal of Physics: Conference Series*, vol. 1325–10, Oct 2019, pp. 1–8.

[51] Feng, J.; Yang, L. T.; Zhang, R. “Practical Privacy-Preserving High-Order Bi-Lanczos in Integrated Edge-Fog-Cloud Architecture for Cyber-Physical-Social Systems”, *ACM Transactions on Internet Technology*, vol. 19–2, Apr 2019.

[52] Forbes. “The Top 8 IoT Trends For 2018”. Source: https://www.forbes.com/sites/danielnewman/2017/12/19/the-top-8-iot-trends-for-2018, Jan 2018.

[53] Gama, E. S.; Immich, R.; Bittencourt, L. F. “Towards a Multi-Tier Fog/Cloud Architecture for Video Streaming”. In: International Conference on Utility and Cloud Computing Companion, 2018, pp. 13–14.

[54] Ganz, F.; Puschmann, D.; Barnaghi, P.; Carrez, F. “A Practical Evaluation of Information Processing and Abstraction Techniques for the Internet of Things”, *IEEE Internet of Things Journal*, vol. 2–4, Aug 2015, pp. 340–354.

[55] Garcia Lopez, P.; Montresor, A.; Epema, D.; Datta, A.; Higashino, T.; Iamnitchi, A.; Barcellos, M.; Felber, P.; Riviere, E. “Edge-centric Computing: Vision and Challenges”, *ACM SIGCOMM Computer Communication Review*, vol. 45–5, Sep 2015, pp. 37–42.

[56] Garey, M. R.; Johnson, D. S.; Sethi, R. “The complexity of flowshop and jobshop scheduling”, *Mathematics of Operations Research*, vol. 1–2, May 1976, pp. 117–129.

[57] Gartner. “Edge computing promises near real-time insights and facilitates localized actions”. Source: https://www.gartner.com/smarterwithgartner/what-edge-computing-means-for-infrastructure-and-operations-leaders, Jan 2021.

[58] Graham, R. L.; Lawler, E. L.; Lenstra, J. K.; Kan, A. H. G. R. “Optimization and approximation in deterministic sequencing and scheduling: a survey”. In: *Discrete Optimization II*, Elsevier, 1979, pp. 287–326.

[59] Greenstein, B. “IoT Trends in 2018: AI, Blockchain, and the Edge”. Source: https://iot.ieee.org/newsletter/january-2018/iot-trends-in-2018-ai-blockchain-and-the-edge, Jan 2018.

[60] Guan, L.; Liu, P.; Xing, X.; Ge, X.; Zhang, S.; Yu, M.; Jaeger, T. “TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone”. In: International Conference on Mobile Systems, Applications, and Services, 2017, pp. 488–501.

[61] Gupta, H.; Vahid Dastjerdi, A.; Ghosh, S. K.; Buyya, R. “iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments”, *Software: Practice and Experience*, vol. 47–9, Jun 2017, pp. 1275–1296.

[62] Gusev, M.; Dustdar, S. “Going Back to the Roots—The Evolution of Edge Computing, An IoT Perspective”, *IEEE Internet Computing*, vol. 22–2, Mar 2018, pp. 5–15.

[63] Guthaus, M. R.; Ringenberg, J. S.; Ernst, D.; Austin, T. M.; Mudge, T.; Brown, R. B. “MiBench: A free, commercially representative embedded benchmark suite”. In: International Workshop on Workload Characterization, 2001, pp. 3–14.

[64] Han, Z.; Tan, H.; Li, X.; Jiang, S. H. .; Li, Y.; Lau, F. C. M. “OnDisc: Online Latency-Sensitive Job Dispatching and Scheduling in Heterogeneous Edge-Clouds”, *IEEE/ACM Transactions on Networking*, vol. 27–6, Nov 2019, pp. 2472–2485.

[65] Hassija, V.; Chamola, V.; Saxena, V.; Jain, D.; Goyal, P.; Sikdar, B. “A Survey on IoT Security: Application Areas, Security Threats, and Solution Architectures”, *IEEE Access*, vol. 7–6, Jun 2019, pp. 82721–82743.

[66] Heer, T.; Garcia-Morchon, O.; Hummen, R.; Keoh, S. L.; Kumar, S. S.; Wehrle, K. “Security Challenges in the IP-based Internet of Things”, *Wireless Personal Communications*, vol. 61–3, Sep 2011, pp. 527–542.

[67] Hernandez, L.; Cao, H.; Wachowicz, M. “Implementing an edge-fog-cloud architecture for stream data management”. In: Fog World Congress, 2017, pp. 1–6.

[68] Hu, P.; Dhelim, S.; Ning, H.; Qiu, T. “Survey on fog computing: architecture, key technologies, applications and open issues”, *Journal of Network and Computer Applications*, vol. 98–11, Nov 2017, pp. 27–42.

[69] Hu, W.; Tan, T.; Wang, L.; Maybank, S. “A survey on visual surveillance of object motion and behaviors”, *Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 34–3, Aug 2004, pp. 334–352.

[70] Hurbungs, V.; Bassoo, V.; Fowdur, T. “Fog and edge computing: concepts, tools and focus areas”, *International Journal of Information Technology*, vol. 13–2, Jan 2021, pp. 511–522.

[71] Huynh, V.; Radenkovic, M. “Interdependent Multi-layer Spatial Temporal-based Caching in Heterogeneous Mobile Edge and Fog Networks”. In: International

Conference on Pervasive and Embedded Computing and Communication Systems, 2019, pp. 34–45.

[72] Imagination Technologies Ltd. “Virtualization Module of the MIPS32 Architecture”, Technical Report, Imagination Technologies Ltd, 2013, 175p.

[73] Iorga, M.; Feldman, L.; Barton, R.; Martin, M. J.; Goren, N. S.; Mahmoudi, C. “Fog computing conceptual model”, Technical Report, National Institute of Standards and Technology, 2018, 14p.

[74] Jamil, B.; Shojafar, M.; Ahmed, I.; Ullah, A.; Munir, K.; Ijaz, H. “A job scheduling algorithm for delay and performance optimization in fog computing”, *Concurrency and Computation: Practice and Experience*, vol. 32–7, Nov 2020, pp. 1–13.

[75] Jang, J.; Kong, S.; Kim, M.; Kim, D.; Kang, B. B. “SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment”. In: Network and Distributed System Security Symposium, 2015, pp. 1–15.

[76] Jiang, C.; Cheng, X.; Gao, H.; Zhou, X.; Wan, J. “Toward Computation Offloading in Edge Computing: A Survey”, *IEEE Access*, vol. 7–8, Aug 2019, pp. 131543–131558.

[77] Jing, Q.; Vasilakos, A.; Wan, J.; Lu, J.; Qiu, D. “Security of the Internet of Things: perspectives and challenges”, *Wireless Networks*, vol. 20–8, Jun 2014, pp. 2481–2501.

[78] Kai, C.; Zhou, H.; Yi, Y.; Huang, W. “Collaborative Cloud-Edge-End Task Offloading in Mobile-Edge Computing Networks with Limited Communication Capability”, *IEEE Transactions on Cognitive Communications and Networking*, vol. 1–1, Aug 2020, pp. 1–11.

[79] Kimovski, D.; Matha, R.; Hammer, J.; Mehran, N.; Hellwagner, H.; Prodan, R. “Cloud, Fog or Edge: Where to Compute?”, *IEEE Internet Computing*, vol. 1–1, Jan 2021, pp. 1–8.

[80] Klein, G.; Andronick, J.; Elphinstone, K.; Murray, T.; Sewell, T.; Kolanski, R.; Heiser, G. “Comprehensive Formal Verification of an OS Microkernel”, *ACM Transactions on Computer Systems*, vol. 32–1, Feb 2014, pp. 1–70.

[81] Klein, G.; Elphinstone, K.; Heiser, G.; Andronick, J.; Cock, D.; Derrin, P.; Elkaduwe, D.; Engelhardt, K.; Kolanski, R.; Norrish, M.; Sewell, T.; Tuch, H.; Winwood, S. “SeL4: Formal Verification of an OS Kernel”. In: Symposium on Operating Systems Principles, 2009, pp. 207–220.

[82] Kolias, C.; Kambourakis, G.; Stavrou, A.; Voas, J. “DDoS in the IoT: Mirai and other botnets”, *Computer*, vol. 50–7, Jul 2017, pp. 80–84.

[83] Leung, J. Y. “Handbook of scheduling: algorithms, models, and performance analysis”. CRC Press, 2004, 1216p.

[84] Lunardi, W. T.; Birgin, E. G.; Laborie, P.; Ronconi, D. P.; Voos, H. “Mixed Integer linear programming and constraint programming models for the online printing shop scheduling problem”, *Computers & Operations Research*, vol. 123–11, Nov 2020, pp. 1–20.

[85] Lunardi, W. T.; Birgin, E. G.; Ronconi, D. P.; Voos, H. “Metaheuristics for the online printing shop scheduling problem”, *European Journal of Operational Research*, vol. 1–1, Sep 2020, pp. 419–441.

[86] Mach, P.; Becvar, Z. “Mobile Edge Computing: A Survey on Architecture and Computation Offloading”, *IEEE Communications Surveys Tutorials*, vol. 19–3, Mar 2017, pp. 1628–1656.

[87] Maene, P.; Götzfried, J.; de Clercq, R.; Müller, T.; Freiling, F.; Verbauwhede, I. “Hardware-Based Trusted Computing Architectures for Isolation and Attestation”, *IEEE Transactions on Computers*, vol. 67–3, Mar 2018, pp. 361–374.

[88] Margolis, J.; Oh, T. T.; Jadhav, S.; Jeong, J. P.; Kim, Y. H.; Kim, J. N. “Analysis and Impact of IoT Malware”. In: International Conference on Information Technology Education, 2017, pp. 187–187.

[89] Marques, W. d. S.; Souza, P. S. S. d.; Rossi, F. D.; Rodrigues, G. d. C.; Calheiros, R. N.; Conterato, M. d. S.; Ferreto, T. C. “Evaluating container-based virtualization overhead on the general-purpose IoT platform”. In: Symposium on Computers and Communications, 2018, pp. 8–13.

[90] Martins, J.; Tavares, A.; Solieri, M.; Bertogna, M.; Pinto, S. “Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems”. In: Workshop on Next Generation Real-Time Embedded Systems, 2020, pp. 1–14.

[91] Mayer, R.; Graser, L.; Gupta, H.; Saurez, E.; Ramachandran, U. “EmuFog: Extensible and scalable emulation of large-scale fog computing infrastructures”. In: Fog World Congress, 2017, pp. 1–6.

[92] Mechalikh, C.; Taktak, H.; Moussa, F. “PureEdgeSim: A Simulation Toolkit for Performance Evaluation of Cloud, Fog, and Pure Edge Computing Environments”. In: International Conference on High Performance Computing Simulation, 2019, pp. 700–707.

[93] Mell, Peter and Grance, Timothy. “The NIST definition of cloud computing”, Technical Report, National Institute of Standards and Technology, 2011, 7p.

[94] Meltdown Attack. “Meltdown and spectre”. Source: https://meltdownattack.com, Oct 2018.

[95] Meng, J.; Tan, H.; Li, X.; Han, Z.; Li, B. “Online Deadline-Aware Task Dispatching and Scheduling in Edge Computing”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 31–6, Dec 2020, pp. 1270–1286.

[96] Mirzamohammadi, S.; Sani, A. A. “The Case for a Virtualization-Based Trusted Execution Environment in Mobile Devices”. In: Asia-Pacific Workshop on Systems, 2018, pp. 1–8.

[97] Mohan, N.; Kangasharju, J. “Edge-Fog cloud: A distributed cloud for Internet of Things computations”. In: Cloudification of the Internet of Things, 2016, pp. 1–6.

[98] Morabito, R. “Virtualization on Internet of Things Edge Devices With Container Technologies: A Performance Evaluation”, *IEEE Access*, vol. 5–1, May 2017, pp. 8835–8850.

[99] Morabito, R.; Beijar, N. “Enabling Data Processing at the Network Edge through Lightweight Virtualization Technologies”. In: International Conference on Sensing, Communication and Networking, 2016, pp. 1–6.

[100] Morabito, R.; Cozzolino, V.; Ding, A. Y.; Beijar, N.; Ott, J. “Consolidate IoT Edge Computing with Lightweight Virtualization”, *IEEE Network*, vol. 32–1, Jan 2018, pp. 102–111.

[101] Morabito, R.; Petrolo, R.; Loscrì, V.; Mitton, N. “LEGIoT: A Lightweight Edge Gateway for the Internet of Things”, *Future Generation Computer Systems*, vol. 81–4, Apr 2018, pp. 1–15.

[102] Moratelli, C.; Johann, S.; Neves, M.; Hessel, F. “Embedded Virtualization for the Design of Secure IoT Applications”. In: International Symposium on Rapid System Prototyping, 2016, pp. 2–6.

[103] Moratelli, C. R. “A lightweight virtualization layer with hardware-assistance for embedded systems”, PhD Thesis, Computer Science Graduate Program, PUCRS, 2016, 155p.

[104] Moratelli, C. R.; Tiburski, R. T.; de Matos, E.; Portal, G.; Johann, S. F.; Hessel, F. “Privacy and security of Internet of Things devices”. In: *Real-Time Data Analytics for Large Scale Sensor Data*, Academic Press, 2020, pp. 183–214.

[105] Mosenia, A.; Jha, N. K. “A Comprehensive Study of Security of Internet-of-Things”, *IEEE Transactions on Emerging Topics in Computing*, vol. 5–4, Oct 2017, pp. 586–602.

[106] Munir, A.; Kansakar, P.; Khan, S. U. “IFCIoT: Integrated Fog Cloud IoT: A novel architectural paradigm for the future Internet of Things”, *IEEE Consumer Electronics Magazine*, vol. 6–3, Jul 2017, pp. 74–82.

[107] Naha, R. K.; Garg, S.; Georgakopoulos, D.; Jayaraman, P. P.; Gao, L.; Xiang, Y.; Ranjan, R. “Fog Computing: Survey of Trends, Architectures, Requirements, and Research Directions”, *IEEE Access*, vol. 6–8, Aug 2018, pp. 47980–48009.

[108] Nuijten, W. P. M.; Aarts, E. H. L. “A computational study of constraint satisfaction for multiple capacitated job shop scheduling”, *European Journal of Operational Research*, vol. 90–2, Apr 1996, pp. 269–284.

[109] Olatunji, I. E.; Cheng, C.-H. “Video Analytics for Visual Surveillance and Applications: An Overview and Survey”. Springer International Publishing, 2019, chap. 15, pp. 475–515.

[110] Omoniwa, B.; Hussain, R.; Javed, M. A.; Bouk, S. H.; Malik, S. A. “Fog/Edge Computing-Based IoT (FECIoT): Architecture, Applications, and Research Issues”, *IEEE Internet of Things Journal*, vol. 6–3, Jun 2019, pp. 4118–4149.

[111] OpenFog Consortium. “OpenFog Reference Architecture for Fog Computing”, Technical Report, OpenFog Consortium, 2017, 162p.

[112] Pahl, C.; Helmer, S.; Miori, L.; Sanin, J.; Lee, B. “A Container-Based Edge Cloud PaaS Architecture Based on Raspberry Pi Clusters”. In: International Conference on Future Internet of Things and Cloud Workshops, 2016, pp. 117–124.

[113] Patel, A.; Daftedar, M.; Shalan, M.; El-Kharashi, M. W. “Embedded Hypervisor Xvisor: A Comparative Analysis”. In: International Conference on Parallel, Distributed and Network-Based Processing, 2015, pp. 682–691.

[114] Patman, J.; Lovett, P.; Banning, A.; Barnert, A.; Chemodanov, D.; Calvam, P. “Data-Driven Edge Computing Resource Scheduling for Protest Crowds Incident Management”. In: International Symposium on Network Computing and Applications, 2018, pp. 1–8.

[115] Pék, G.; Buttyán, L.; Bencsáth, B. “A Survey of Security Issues in Hardware Virtualization”, *ACM Computing Surveys*, vol. 45–3, Jun 2013, pp. 1–34.

[116] Pinedo, M. “Scheduling”. Springer, 2012, 673p.

[117] Pinto, S.; Gomes, T.; Pereira, J.; Cabral, J.; Tavares, A. “IIoTEED: An Enhanced, Trusted Execution Environment for Industrial IoT Edge Devices”, *IEEE Internet Computing*, vol. 21–1, Jan 2017, pp. 40–47.

[118] Pinto, S.; Oliveira, D.; Pereira, J.; Cardoso, N.; Ekpanyapong, M.; Cabral, J.; Tavares, A. “Towards a lightweight embedded virtualization architecture exploiting ARM TrustZone”. In: International Conference on Emerging Technology and Factory Automation, 2014, pp. 1–4.

[119] Pinto, S.; Santos, N. “Demystifying Arm TrustZone: A Comprehensive Survey”, *ACM Computing Surveys*, vol. 51–6, Feb 2019, pp. 1–36.

[120] Pokrovskaia, N.; Khansuvarova, T.; Khansuvarov, R. “Network decentralized regulation with the fog-edge computing and blockchain for business development”. In: European Conference on Management, Leadership & Governance, 2018, pp. 205–212.

[121] Portilla, J.; Mujica, G.; Lee, J.; Riesgo, T. “The Extreme Edge at the Bottom of the Internet of Things: A Review”, *IEEE Sensors Journal*, vol. 19–9, May 2019, pp. 3179–3190.

[122] Preden, J. S.; Tammemae, K.; Jantsch, A.; Leier, M.; Riid, A.; Calis, E. “The Benefits of Self-Awareness and Attention in Fog and Mist Computing”, *Computer*, vol. 48–7, Jul 2015, pp. 37–45.

[123] Rakotondravony, N.; Taubmann, B.; Mandarawi, W.; Weishäupl, E.; Xu, P.; Kolosnjaji, B.; Protsenko, M.; De Meer, H.; Reiser, H. P. “Classifying malware attacks in IaaS cloud environments”, *Journal of Cloud Computing*, vol. 6–1, Dec 2017, pp. 1–12.

[124] Razzaque, M. A.; Milojevic-Jevric, M.; Palade, A.; Clarke, S. “Middleware for Internet of Things: A Survey”, *IEEE Internet of Things Journal*, vol. 3–1, Feb 2016, pp. 70–95.

[125] Ren, J.; Yu, G.; He, Y.; Li, G. Y. “Collaborative Cloud and Edge Computing for Latency Minimization”, *IEEE Transactions on Vehicular Technology*, vol. 68–5, Mar 2019, pp. 5031–5044.

[126] Roman, R.; Lopez, J.; Mambo, M. “Mobile edge computing, Fog et al.: A survey and analysis of security threats and challenges”, *Future Generation Computer Systems*, vol. 78–2, Jan 2018, pp. 680–698.

[127] RTInsights. “In 2018, Get Ready for the Convergence of IoT, AI, Fog, and Blockchain”. Source: https://www.rtinsights.com/in-2018-get-ready-for-the-convergence-of-iot-ai-fog-and-blockchain, Jan 2018.

[128] Russell, R. “Virtio: Towards a De-facto Standard for Virtual I/O Devices”, *ACM SIGOPS Operating Systems Review*, vol. 42–5, Jul 2008, pp. 95–103.

[129] Sabella, D.; Vaillant, A.; Kuure, P.; Rauschenbach, U.; Giust, F. “Mobile-Edge Computing Architecture: The role of MEC in the Internet of Things”, *IEEE Consumer Electronics Magazine*, vol. 5–4, Oct 2016, pp. 84–91.

[130] Sabt, M.; Achemlal, M.; Bouabdallah, A. “Trusted Execution Environment: What It is, and What It is Not”. In: International Conference on Trust, Security and Privacy in Computing and Communications, 2015, pp. 57–64.

[131] Sandström, K.; Vulgarakis, A.; Lindgren, M.; Nolte, T. “Virtualization technologies in embedded real-time systems”. In: International Conference on Emerging Technologies Factory Automation, 2013, pp. 1–8.

[132] Sarddar, D.; Bose, R. “A mobile cloud computing architecture with easy resource sharing”, *International Journal of Current Engineering and Technology*, vol. 4–3, Jun 2014, pp. 1249–1254.

[133] Sharma, S.; Saini, H. “A novel four-tier architecture for delay aware scheduling and load balancing in fog environment”, *Sustainable Computing: Informatics and Systems*, vol. 24–12, Dec 2019, pp. 1–12.

[134] Shen, H.; Bai, G.; Hu, Y.; Wang, T. “P2TA: Privacy-preserving task allocation for edge computing enhanced mobile crowdsensing”, *Journal of Systems Architecture*, vol. 97–1, Aug 2019, pp. 130–141.

[135] Shi, W.; Cao, J.; Zhang, Q.; Li, Y.; Xu, L. “Edge Computing: Vision and Challenges”, *IEEE Internet of Things Journal*, vol. 3–5, Oct 2016, pp. 637–646.

[136] Siboni, S.; Sachidananda, V.; Meidan, Y.; Bohadana, M.; Mathov, Y.; Bhairav, S.; Shabtai, A.; Elovici, Y. “Security testbed for internet-of-things devices”, *IEEE Transactions on Reliability*, vol. 68–1, Mar 2019, pp. 23–44.

[137] Silberschatz, A.; Galvin, P. B.; Gagne, G. “Operating System Concepts”. Wiley Publishing, 2012, 992p.

[138] Smith, J.; Nair, R. “Virtual Machines: Versatile Platforms for Systems and Processes”. Morgan Kaufmann Publishers, 2005, 656p.

[139] Soltesz, S.; Pötzl, H.; Fiuczynski, M. E.; Bavier, A.; Peterson, L. “Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors”. In: European Conference on Computer Systems, 2007, pp. 275–287.

[140] Sonmez, C.; Ozgovde, A.; Ersoy, C. “EdgeCloudSim: An environment for performance evaluation of Edge Computing systems”. In: International Conference on Fog and Mobile Edge Computing, 2017, pp. 39–44.

[141] Souza, A.; Cacho, N.; Noor, A.; Jayaraman, P. P.; Romanovsky, A.; Ranjan, R. “Osmotic Monitoring of Microservices between the Edge and Cloud”. In: International Conference on High Performance Computing and Communications, 2018, pp. 758–765.

[142] Sun, X.; Ansari, N. “EdgeIoT: Mobile Edge Computing for the Internet of Things”, *IEEE Communications Magazine*, vol. 54–12, Dec 2016, pp. 22–29.

[143] Taherizadeh, S.; Stankovski, V.; Grobelnik, M. “A Capillary Computing Architecture for Dynamic Internet of Things: Orchestration of Microservices from Edge Devices to Fog and Cloud Providers”, *Sensors*, vol. 18–9, Sep 2018, pp. 1–23.

[144] Tan, H.; Han, Z.; Li, X.; Lau, F. C. M. “Online job dispatching and scheduling in edge-clouds”. In: International Conference on Computer Communications, 2017, pp. 1–9.

[145] Taneja, M.; Davy, A. “Resource aware placement of IoT application modules in Fog-Cloud Computing Paradigm”. In: Symposium on Integrated Network and Service Management, 2017, pp. 1222–1228.

[146] Tanenbaum, A. S.; Wetherall, D. J. “Computer Networks”. Pearson Prentice Hall, 2011, 933p.

[147] Tang, B.; Chen, Z.; Hefferman, G.; Wei, T.; He, H.; Yang, Q. “A Hierarchical Distributed Fog Computing Architecture for Big Data Analysis in Smart Cities”. In: ASE BigData & SocialInformatics, 2015, pp. 1–6.

[148] Tanganelli, G.; Vallati, C.; Mingozzi, E. “Edge-Centric Distributed Discovery and Access in the Internet of Things”, *IEEE Internet of Things Journal*, vol. 5–1, Feb 2018, pp. 425–438.

[149] Tank, D.; Aggarwal, A.; Chaubey, N. “Virtualization vulnerabilities, security issues, and solutions: a critical study and comparison”, *International Journal of Information Technology*, vol. 1–2, Feb 2019, pp. 1–16.

[150] The seL4 Foundation. “The seL4 Microkernel An Introduction”. Source: https://cdn.hackaday.io/files/1713937332878112/seL4-whitepaper.pdf, Oct 2020.

[151] Tiburski, R. T.; Amaral, L. A.; de Matos, E.; de Azevedo, D. F. G.; Hessel, F. “The Role of Lightweight Approaches Towards the Standardization of a Security Architecture for IoT Middleware Systems”, *IEEE Communications Magazine*, vol. 54–12, Dec 2016, pp. 56–62.

[152] Tiburski, R. T.; Amaral, L. A.; de Matos, E.; Hessel, F.; de Azevedo, D. “Evaluating the Use of TLS and DTLS Protocols in IoT Middleware Systems Applied to E-health”. In: Annual Consumer Communications & Networking Conference, 2017, pp. 480–485.

[153] Tiburski, R. T.; Amaral, L. A.; Hessel, F. “Security Challenges in 5G-Based IoT Middleware Systems”. Springer International Publishing, 2016, chap. 17, pp. 399–418.

[154] Tiburski, R. T.; de Matos, E.; Hessel, F. “Evaluating the DTLS Protocol from CoAP in Fog-to-Fog Communications”. In: International Conference on Service-Oriented System Engineering, 2019, pp. 90–905.

[155] Tiburski, R. T.; Moratelli, C. R.; Johann, S. F.; de Matos, E.; Hessel, F. “A lightweight virtualization model to enable edge computing in deeply embedded systems”, *Software: Practice and Experience*, vol. 1–1, Mar 2021, pp. 1–18.

[156] Tiburski, R. T.; Moratelli, C. R.; Johann, S. F.; Neves, M. V.; de Matos, E.; Amaral, L. A.; Hessel, F. “Lightweight Security Architecture Based on Embedded Virtualization and Trust Mechanisms for IoT Edge Devices”, *IEEE Communications Magazine*, vol. 57–2, Feb 2019, pp. 67–73.

[157] Tien, C.; Tsai, T.; Chen, I.; Kuo, S. “UFO - Hidden Backdoor Discovery and Security Verification in IoT Device Firmware”. In: International Symposium on Software Reliability Engineering Workshops, 2018, pp. 18–23.

[158] Tong, L.; Li, Y.; Gao, W. “A hierarchical edge cloud architecture for mobile computing”. In: International Conference on Computer Communications, 2016, pp. 1–9.

[159] Vasconcelos, F. F.; Sarmento, R. M.; Rebouças Filho, P. P.; de Albuquerque, V. H. C. “Artificial intelligence techniques empowered edge-cloud architecture for brain CT image analysis”, *Engineering Applications of Artificial Intelligence*, vol. 91–5, May 2020, pp. 1–13.

[160] Vilalta, R.; Lopez, V.; Giorgetti, A.; Peng, S.; Orsini, V.; Velasco, L.; Serral-Gracia, R.; Morris, D.; De Fina, S.; Cugini, F.; Castoldi, P.; Mayoral, A.; Casellas, R.; Martinez, R.; Verikoukis, C.; Munoz, R. “TelcoFog: A Unified Flexible Fog and Cloud Computing Architecture for 5G Networks”, *IEEE Communications Magazine*, vol. 55–8, Aug 2017, pp. 36–43.

[161] Wang, A.; Liang, R.; Liu, X.; Zhang, Y.; Chen, K.; Li, J. “An Inside Look at IoT Malware”. In: International Conference on Industrial IoT Technologies and Applications, 2017, pp. 176–186.

[162] Wang, B.; Song, Y.; Wang, C.; Huang, W.; Qin, X. “A Study on Heuristic Task Scheduling Optimizing Task Deadline Violations in Heterogeneous Computational Environments”, *IEEE Access*, vol. 8–11, Nov 2020, pp. 205635–205645.

[163] Wang, B.; Wang, C.; Huang, W.; Song, Y.; Qin, X. “A Survey and Taxonomy on Task Offloading for Edge-Cloud Computing”, *IEEE Access*, vol. 8–10, Oct 2020, pp. 186080–186101.

[164] Wang, L.; Jiao, L.; Kliazovich, D.; Bouvry, P. “Reconciling task assignment and scheduling in mobile edge clouds”. In: International Conference on Network Protocols, 2016, pp. 1–6.

[165] Wang, R.; Yan, J.; Wu, D.; Wang, H.; Yang, Q. “Knowledge-Centric Edge Computing Based on Virtualized D2D Communication Systems”, *IEEE Communications Magazine*, vol. 56–5, May 2018, pp. 32–38.

[166] Wired. “Hackers Remotely Kill a Jeep on the Highway—With Me in It”. Source: https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway, Nov 2020.

[167] Wired. “The Jeep Hackers Are Back to Prove Car Hacking Can Get Much Worse”. Source: https://www.wired.com/2016/08/jeep-hackers-return-high-speed-steering-acceleration-hacks, Nov 2020.

[168] wolfSSL. “Embedded SSL/TLS Library”. Source: https://www.wolfssl.com/, Nov 2020.

[169] Wu, Q.; Zhang, H.; Du, P.; Li, Y.; Guo, J.; He, C. “Enabling adaptive deep neural networks for video surveillance in distributed edge clouds”. In: International Conference on Parallel and Distributed Systems, 2019, pp. 525–528.

[170] Xiao, Y.; Jia, Y.; Liu, C.; Cheng, X.; Yu, J.; Lv, W. “Edge Computing Security: State of the Art and Challenges”, *Proceedings of the IEEE*, vol. 107–8, Aug 2019, pp. 1608–1631.

[171] Yang, M.; Ma, H.; Wei, S.; Zeng, Y.; Chen, Y.; Hu, Y. “A Multi-Objective Task Scheduling Method for Fog Computing in Cyber-Physical-Social Services”, *IEEE Access*, vol. 8–3, Mar 2020, pp. 65085–65095.

[172] Yi, S.; Hao, Z.; Zhang, Q.; Zhang, Q.; Shi, W.; Li, Q. “LAVEA: Latency-Aware Video Analytics on Edge Computing Platform”. In: International Conference on Distributed Computing Systems, 2017, pp. 2573–2574.

[173] Zhang, J.; Chen, B.; Zhao, Y.; Cheng, X.; Hu, F. “Data Security and Privacy-Preserving in Edge Computing Paradigm: Survey and Open Issues”, *IEEE Access*, vol. 6–3, Mar 2018, pp. 18209–18237.

[174] Zhang, P.; Zhou, M.; Fortino, G. “Security and trust issues in Fog computing: A survey”, *Future Generation Computer Systems*, vol. 88–11, Nov 2018, pp. 16–27.

[175] Zhang, W.; Chen, J.; Zhang, Y.; Raychaudhuri, D. “Towards Efficient Edge Cloud Augmentation for Virtual Reality MMOGs”. In: Symposium on Edge Computing, 2017, pp. 1–14.

[176] Zimba, A.; Wang, Z.; Mulenga, M. “Cryptojacking injection: A paradigm shift to cryptocurrency-based web-centric internet attacks”, *Journal of Organizational Computing and Electronic Commerce*, vol. 29–1, Feb 2019, pp. 40–59.

# APPENDIX A – RESULTS FOR TASK ASSIGNMENT AND SCHEDULING EXPERIMENTS

Table A.1 – Results for Experiments 1, 2 and 3.

| Orchestration algorithm | Edge devices count | Generated tasks | Tasks successfully executed | Task not executed (No resources available) | Tasks failed (deadline) | Total tasks executed (Cloud) | Tasks successfully executed (Cloud) | Total tasks executed (Fog) | Tasks successfully executed (Fog) | Total tasks executed (Edge) | Tasks successfully executed (Edge) | Total network traffic (GB) | Average CPU Usage (%) | CPU usage (Cloud) (%) | CPU usage (Fog) (%) | CPU usage (Edge) (%) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DTAS-EDGE | 50 | 1440000 | 1440000 | 0 | 0 | 160830 | 160830 | 291220 | 291220 | 987950 | 987950 | 24,2 | 72,5 | 15,6 | 47,1 | 76,1 |
| DTAS-EDGE | 50 | 1440000 | 1440000 | 0 | 0 | 161020 | 161020 | 291140 | 291140 | 987840 | 987840 | 24,2 | 72,4 | 15,6 | 47,1 | 76,1 |
| DTAS-EDGE | 50 | 1440000 | 1440000 | 0 | 0 | 157440 | 157440 | 290410 | 290410 | 992150 | 992150 | 20,9 | 72,8 | 15,2 | 47,0 | 76,5 |
| DTAS-FOG | 50 | 1440000 | 1180520 | 239120 | 20360 | 297170 | 285900 | 695760 | 686670 | 327510 | 327510 | 96,5 | 29,5 | 26,3 | 84,0 | 24,2 |
| DTAS-FOG | 50 | 1440000 | 1181200 | 238380 | 20420 | 295750 | 284410 | 694470 | 685390 | 330590 | 330590 | 96,5 | 29,7 | 26,3 | 84,0 | 24,4 |
| DTAS-FOG | 50 | 1440000 | 1179970 | 239520 | 20510 | 295150 | 283890 | 694440 | 685190 | 330650 | 330650 | 96,4 | 29,7 | 26,2 | 84,0 | 24,4 |
| DTAS-CLOUD | 50 | 1440000 | 1074900 | 360920 | 4180 | 1098910 | 1098910 | 160630 | 156450 | 0 | 0 | 82,5 | 3,6 | 67,2 | 26,7 | 0,0 |
| DTAS-CLOUD | 50 | 1440000 | 1086030 | 349000 | 4970 | 1094880 | 1094880 | 170620 | 165650 | 0 | 0 | 82,8 | 3,6 | 67,7 | 27,1 | 0,0 |
| DTAS-CLOUD | 50 | 1440000 | 1088590 | 347020 | 4390 | 1115660 | 1115660 | 150830 | 146440 | 0 | 0 | 82,5 | 3,5 | 68,5 | 25,4 | 0,0 |
| INCREASE_LIFETIME | 50 | 1440000 | 1436070 | 0 | 3930 | 717980 | 716060 | 560270 | 560170 | 161750 | 159840 | 64,7 | 23,8 | 47,0 | 70,5 | 18,6 |
| INCREASE_LIFETIME | 50 | 1440000 | 1435980 | 0 | 4020 | 718590 | 716470 | 559780 | 559740 | 161630 | 159770 | 64,7 | 23,4 | 47,3 | 70,5 | 18,2 |
| INCREASE_LIFETIME | 50 | 1440000 | 1435850 | 0 | 4150 | 717450 | 715200 | 559790 | 559770 | 162760 | 160880 | 59,9 | 23,5 | 47,1 | 70,4 | 18,4 |
| PTS | 50 | 1440000 | 1437120 | 0 | 2880 | 516740 | 516530 | 387260 | 387160 | 536000 | 533430 | 64,2 | 47,9 | 34,5 | 51,8 | 47,7 |
| PTS | 50 | 1440000 | 1437170 | 0 | 2830 | 516730 | 516560 | 387270 | 387140 | 536000 | 533470 | 64,2 | 47,9 | 34,5 | 51,8 | 47,7 |
| PTS | 50 | 1440000 | 1437230 | 0 | 2770 | 516790 | 516680 | 387210 | 387040 | 536000 | 533510 | 59,5 | 47,8 | 34,5 | 51,7 | 47,7 |
| ROUND_ROBIN | 50 | 1440000 | 660580 | 0 | 779420 | 169600 | 169600 | 212000 | 212000 | 1058400 | 278980 | 47,7 | 78,2 | 10,3 | 25,7 | 84,8 |
| ROUND_ROBIN | 50 | 1440000 | 690030 | 0 | 749970 | 169600 | 169600 | 212000 | 212000 | 1058400 | 308430 | 47,9 | 76,8 | 10,3 | 25,7 | 83,2 |
| ROUND_ROBIN | 50 | 1440000 | 705320 | 0 | 734680 | 169600 | 169600 | 212000 | 212000 | 1058400 | 323720 | 46,7 | 76,1 | 10,3 | 25,7 | 82,5 |
| DTAS-EDGE | 20 | 1080000 | 1079530 | 0 | 210 | 207290 | 207080 | 429050 | 429050 | 443660 | 443660 | 27,8 | 80,0 | 18,5 | 70,1 | 85,1 |
| DTAS-EDGE | 20 | 1080000 | 1079540 | 0 | 260 | 206860 | 206600 | 429050 | 429050 | 444090 | 444090 | 27,8 | 80,1 | 18,5 | 70,1 | 85,2 |
| DTAS-EDGE | 20 | 1080000 | 1079500 | 0 | 200 | 206960 | 206760 | 429290 | 429290 | 443750 | 443750 | 27,8 | 80,1 | 18,5 | 70,1 | 85,2 |
| INCREASE_LIFETIME | 20 | 1080000 | 1077900 | 0 | 2100 | 632650 | 630770 | 414030 | 414030 | 33320 | 33100 | 45,0 | 20,6 | 42,5 | 66,2 | 10,4 |
| INCREASE_LIFETIME | 20 | 1080000 | 1078220 | 0 | 1780 | 646500 | 644830 | 407550 | 407550 | 25950 | 25840 | 45,0 | 18,7 | 43,5 | 65,3 | 8,1 |
| INCREASE_LIFETIME | 20 | 1080000 | 1077830 | 0 | 2170 | 634980 | 632970 | 414460 | 414460 | 30560 | 30400 | 45,0 | 19,9 | 42,7 | 66,4 | 9,5 |
| PTS | 20 | 1080000 | 1079490 | 0 | 510 | 536000 | 535990 | 321600 | 321570 | 222400 | 221930 | 44,5 | 49,6 | 35,8 | 53,6 | 49,5 |
| PTS | 20 | 1080000 | 1079520 | 0 | 480 | 536000 | 536000 | 321600 | 321590 | 222400 | 221930 | 44,5 | 49,6 | 35,8 | 53,7 | 49,5 |
| PTS | 20 | 1080000 | 1079670 | 0 | 330 | 536000 | 536000 | 321600 | 321590 | 222400 | 222080 | 44,5 | 49,6 | 35,8 | 53,6 | 49,5 |
| ROUND_ROBIN | 20 | 1080000 | 589790 | 0 | 490210 | 240000 | 240000 | 240000 | 240000 | 600000 | 109790 | 35,6 | 76,1 | 11,0 | 27,4 | 89,0 |
| ROUND_ROBIN | 20 | 1080000 | 582450 | 0 | 497550 | 240000 | 240000 | 240000 | 240000 | 600000 | 102450 | 35,7 | 77,1 | 11,0 | 27,4 | 90,4 |
| ROUND_ROBIN | 20 | 1080000 | 590810 | 0 | 489190 | 240000 | 240000 | 240000 | 240000 | 600000 | 110810 | 35,7 | 76,1 | 11,0 | 27,4 | 89,0 |


Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br