Benefits of Dynamic Computational Offloading for Mobile Devices

Vinay Yadhav[1], Andrew Williams[2], Ondrej Smid[2], Jimmy Kjällman[3], Raihan Ul Islam[4,*], Joacim Halén[5,*] and Wolfgang John[2]

[1]*Ericsson Research, Bangalore, India*

[2]*Ericsson Research, Stockholm, Sweden*

[3]*Ericsson Research, Jorvas, Finland*

[4]*East West University, Dhaka, Bangladesh*

[5]*Independent Researcher, Stockholm, Sweden*

Keywords: Computational Offloading, Edge Computing, WebAssembly, Edge-Native Application Development, 5G, 6G.

Abstract: The proliferation of applications across mobile devices coupled with fast mobile broadband have led to expectations of better application performance, user experiences, and extended device battery life. To address this, we propose a dynamic computational offloading solution that migrates critical application tasks to remote compute sites within mobile networks. Offloading is particularly advantageous for lightweight devices, as it enables access to more capable processing hardware. Application developers can also leverage the offloading service to customize features, address privacy concerns, and optimize performance based on user requirements. Moreover, the solution facilitates local synchronization among collaborating users. Our solution focuses on ad-hoc deployment and dynamic scheduling of fine-grained application tasks triggered by changes in device metrics, without extensive development efforts. It extends application functionality from mobile devices to remote compute environments, complementing the cloud-to-edge paradigm. We introduce a distributed execution framework based on portable, lightweight, and secure WebAssembly runtimes. Additionally, we present a programming model to simplify ad-hoc deployment and dynamic invocation of task modules during runtime. We demonstrate the benefits of our solution, showing significant performance improvements of the application, and reduced energy consumption and heat generation on the mobile device.

1 INTRODUCTION

With the advent of a global app economy, most users have a large variety of applications running on mobile phones and increasingly also on other types of user equipment like XR headsets, cars, or drones, all connected via cellular networks. Many of these connected, mobile devices offer limited computational capabilities and/or energy capacity to run certain applications with high quality of experience for more than a limited time. Computational Offloading is a subset of the distributed computing paradigm that concerns the ability to migrate a component of a running application from a mobile User-Equipment (UE) to a remote offloading site. The end-user of an application would benefit from the service's automated capabilities to balance compute and energy tradeoffs between the UE and the offloading site. Executing certain critical parts of an application in an external compute infrastructure can increase the application's quality of experience by giving offloaded tasks remote access to domain specific hardware accelerators such as GPUs. At the same time, offloading can prolong battery life by decreasing the UE's power utilization and device heat. For very lightweight devices (e.g., head-mounted devices, IoT sensors), this might actually be the only option to offer richer application experience. Application developers can also leverage an offloading service to customize features, address privacy concerns, and optimize performance based on user requirements and contextual factors.

The core idea behind the dynamic computational offloading solution described in this paper is to expand application functionality from connected user equipment to a remote compute environment, e.g., located within the cellular network. This differs from typical edge computing solutions, which target the edge from the opposite direction, i.e., moving applica-

*has been with Ericsson when writing this paper.

Yadhav, V., Williams, A., Smid, O., Kjällman, J., Islam, R., Halén, J. and John, W.
Benefits of Dynamic Computational Offloading for Mobile Devices.
DOI: 10.5220/0012719800003711
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 14th International Conference on Cloud Computing and Services Science (CLOSER 2024)*, pages 265-276
ISBN: 978-989-758-701-6; ISSN: 2184-5042
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

tion functionality traditionally running in the cloud to edge or on-premises compute facilities using largely static, pre-deployed application servers targeting vertical use-cases with big groups of users. In contrast, our solution aims at dynamic offload and deployment of highly granular application tasks, triggered by the application based on situational changes such as changes in the device (e.g., battery levels), network (e.g. radio quality) or application needs. This implies that this solution is use-case agnostic, offering sandboxed remote computational resources to any application on an UE, able to execute ad-hoc deployed application tasks, invoked dynamically essentially anytime and anywhere. We believe that such a solution has the potential to target even the long tail of regional enterprises and developers that usually would not engage in the heavy burdens of deployment, management, and contract handling related to edge solutions like MEC (ESTI, 2023). In this paper we evaluate such a solution based upon two novel features:

- A *distributed execution framework* that takes advantage of the light weight and portability of standalone WebAssembly (Wasm) runtimes
- A *programming model* and associated development toolchain that abstracts away the complexities of communication between a device application and an offloaded component

This paper is structured as follows: Section 2 describes the requirements and design of our proposed offloading solution. Section 3 describes the demo setup and proof-of-concept (PoC) implementation of the solution concept. In Section 4 we present and discuss our measurement results, showing when offloading is beneficial for the application or the mobile device. Section 5 summarizes and relates current state of the art. Finally, Section 6 concludes the paper and points out next steps towards our vision of dynamic computational offloading as a network service.

2 SYSTEM DESIGN

This section describes details about the design of our proposed offloading framework, including requirements, the basic system architecture, the proposed programming model and development toolchain for optimal usage of the offloading framework, as well as crucial security considerations.

2.1 Requirements

In order to design a dynamic computational offloading solution according to our vision and idea, we state

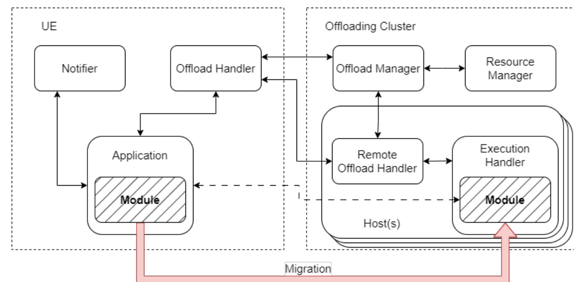


Figure 1: System architecture with communication interfaces (control plane in solid, user-plane in dashed lines).

the following qualitative requirements:

- *Platform independent*, to allow an offloaded task to execute on a variety of different hardware architecture and operating system combinations.
- *Lightweight* so that the framework does not impose an excessive overhead and negatively affect offloaded component load time.
- *Low footprint* so that the framework size is minimized compared to the offloaded component.
- *Secure*, providing isolation between tenants and between tenants and infrastructure.
- *Language independent and open*, allowing developer flexibility and extensibility.

As a result of these requirements, we selected standalone WebAssembly (Wasm) runtimes as the basic execution environment in our solution. While it is relatively immature compared to more conventional portable bytecode formats such as Java, or to more heavyweight distributed computing runtime platforms such as containers or virtual machines, we believe that Wasm is a promising technology for scenarios that require portable and secure runtimes.

2.2 System Architecture

Based on our objectives and requirements, we derived a system architecture for an offloading service framework as depicted in Figure 1. It has two major communication interfaces, one for the control plane and one for the user plane of the offloading service.

The service control plane is split into two parts: (i) User-Equipment (UE) side and (ii) remote offloading cluster side. The latter consists of functionality to manage offload requests and a collection of resources handling and executing offloaded modules. For the application developer and the application running in the UE, the framework manifests as:

A **Notifier** is a local helper functionality which gathers device metrics and exposes them via an API to the running application.

An **Offload Handler** is responsible for handling the interaction with the application, discovering the closest offloading cluster, requesting and negotiating an offload of a specified application module[1], assembling and packaging all meta-data together with the module itself necessary to offload to the cluster, and handshaking the migration of the module with its corresponding part on an allocated host in an edge cluster. It also maintains a list of all offloaded modules and their corresponding applications as well as a reference to the offloading hosts. The application communicates with the UE Offload Handler using an API which is implemented as a programming language specific library and contains functions to request and recall an offload, select an offloading host from an offload resource offer, and synchronize the migration of a module's local state.

An offloading cluster instance handles and offers remote compute to UEs and has two main functional parts, depicted in Figure 1: (i) management of offload requests, mapping them to hosts (workers) based on resource requirements; and (ii) the set of hosts where each host synchronizes the code migration as well as executing the offloaded module. The remote compute part of the control plane consists of the following:

An **Offload Manager** acts as the first remote point of contact for the Offload Handler of an UE that wishes to offload. It coordinates activities such as authentication, authorization, resource management, etc. for each request.

A **Resource Manager** is responsible for the allocated resources (i.e., hosts) assigned to the offloading cluster. It selects a (virtual) host with capabilities that match the requested requirements as specified in an offload request for a module.

A **Remote Offload Handler** is responsible for synchronizing an offloading event with its counterpart in the UE, including replication of the runtime execution environment for the offloaded module as well as re-establishing the internal and external flows. Each (virtual) host will have its own Remote Offload Handler instance.

An **Execution handler** is responsible to execute the offloaded task. We base the execution of tasks on Wasm[2], an instruction format designed to be executed on a memory safe and sandboxed stack-based virtual machine (Haas et al., 2017). An application running on a host machine embeds a WebAssembly runtime, allowing it to load Wasm modules and call exported functions contained within the module, known as *guest functions*. Though sandboxed, the Wasm module can optionally access host machine functionality either through the standardized WASI system interface[3]or through non-standard *host function* calls to host resources like networking, databases, or access to HW accelerators. The offloading framework execution handler acts as the remote application and is extendable through the use of both common and developer-supplied host functions, which provides a lightweight programmatic method of providing additional capabilities to the offloaded module. For communication, the execution handler exposes a RESTful API to the Remote Offload Handler that allows for the invocation of Wasm modules and the calling of exposed Wasm functions. It also exposes an abstracted interface to the Wasm runtime, allowing various implementations such as Wasmedge[4] or Wasmtime[5] to be selected at runtime depending on application requirements. Once instantiated, the offloaded module communicates with the parent application directly rather than through the service framework.

2.3 Programming Model

One of the ambitions of this service is to hide the distributed nature of an application with offloadable components from the developer. The programming models for offloading should therefore guide a developer in the task of dividing the application functionality in a simple way. They should feel intuitive and as natural as possible for the developer, that is, they should preferably be modifications or extensions to well-known existing models. We have focused on two models to realize the dynamic offloading solution, namely channels and functions. Below, one can find brief descriptions of both.

Firstly, we developed a point-to-point communication method, known as *elastic channel*. Normally, channel endpoints are created and connected at the location of the threads or processes and exist throughout their lifetime. An elastic channel is an extension of the channel model where endpoints can transparently change location during their lifetime. In the proposed offloading service framework, elastic channels are asynchronous and bidirectional supporting both one-to-one and one-to-many communication. In the current implementation, one endpoint remains stationary and fixed in the UE device, while the other endpoint is flexible, implying that its location can freely move around, e.g., the closest offloading cluster at the time. This flexibility and static nature of endpoints are explicitly declared by the developer, en-

[1] A module is part of the app possible to be offloaded, including single functions, tasks, or the whole application

[2] https://webassembly.org

[3] https://github.com/WebAssembly/WASI/

[4] https://wasmedge.org/

[5] https://wasmtime.dev/

abling elastic channels to be constructed as a library without the need for external resources.

Secondly, a form of Remote Procedure Call (RPC) API known as *elastic functions* was developed, based upon elastic channel network functionality. Functions and procedures are the fundamental way to abstract an expression or a set of statements in most programming languages. As functions and procedures are so fundamental, it would be good if they could be made elastic in a similar way as the elastic channels. That is, a call (invocation) should look the same both when a module containing the implementation of the function or procedure is residing in the same execution runtime as when the it has been moved to an execution runtime at a different location. In contrast to RPC, here the remote end is not fixed and can be executed locally or remotely (offloaded). To do that, we must be able to identify the functions and procedures that could be moved during an offload and transform them to functions and procedures with the same interface and behaviour but are movable in a transparent way. We realize this via an extra pre-processing step in the compiler toolchain that permits the developer to *tag* functions as offloadable. Functions with such a tag have their interfaces rewritten prior to compilation as elastic channel functions. In this case two versions of the function are built, one that will be natively run on the device, and another compiled to WebAssembly format that can be readily run remotely, independent of the exact hardware and software platform choices at the offloading cluster.

2.4 Offload Sequence

We assume the following pre-conditions when operating this service: the UE and Offload Manager are aware of each other; there is pre-established communication link between the UE Offload Handler and the Offload Manager; and the application is monitoring offload triggers. The former two of these conditions could, e.g, be established by adaptations to existing control plane functions of existing or future cellular networks (i.e., 5G/6G).

Given these pre-conditions, a module offload sequence (using the control plane in Fig. 1) includes the following steps:

1. Information from the notifier results in an offloading trigger threshold being reached.
2. The application, via the offload handler, informs the remote offload manager that there is a pending offload request.
3. The offload manager responds with a set of offers of resources obtained from the resource manager.
4. The application selects one of the offers and informs the offload manager, which instructs the relevant remote offload handler to assign resources.
5. The local and remote offload handlers communicate directly to transfer the module.
6. The execution handler initializes the module, and communication with the local application commences using elastic channels or functions.

2.5 Security Considerations

We identified an important requirement in security and isolation, especially for a solution that executes code imported from untrusted sources (i.e., the UEs and their users) in a remote compute environment. Here we consider security in the dynamic offloading architecture from the point of view of the users and applications on one hand, and the system itself on the other. We outline a few key technologies we see as building blocks for securing our solution.

In addition to utilizing WebAssembly-based sandboxing, we support offloading modules into Trusted Execution Environments (TEEs), such as Confidential Virtual Machines (Guanciale et al., 2022), in cases where offloaded modules and data need to be protected also from the offloading host. The UE side attests these TEEs (Ménétrey et al., 2022) before modules are migrated to them.

We also need to ensure that users are authenticated and authorized to perform offloading and use remote resources. In a scenario where the offloading system is deployed as a service in a mobile network, the Authentication and Key Management for Applications (AKMA) (Huang et al., 2021) mechanism can perform authentication and authorization of the UE side based on network subscription credentials, as well as distribute shared symmetric keys to the UE and the remote offloading system. These keys can be used within Transport Layer Security (TLS) handshakes, in which we also perform TEE attestation when needed.

3 DEMO PoC AND EXPERIMENTAL SETUP

In order to validate the functionality of the described offloading solution and assess the resulting performance, we developed an experiment setup based on a search and rescue scenario involving hazardous environment exploration.

3.1 Use-Case Scenario

We assume a scenario in which a rescue team arrives at a disaster site to explore the potentially dangerous unknown environment using battery-driven robot vehicles or drones. The goal is to discover the location of specific objects such as human survivors or hazardous materials. In such a critical situation it would be advantageous to use more than one robot to simultaneously scout the area, each contributing to a full picture of the search area by exploring a part of the total environment. As such, this scenario requires collaboration between multiple robots on the same task. To prolong the operation time of the battery-driven robots and potentially even increase their scouting performance, each robot can choose to offload some of its functions to a remote offloading site.

As the disaster site was a priori unknown, this scenario takes advantage of ad-hoc deployment of specific application tasks at exactly the place and time when needed. These tasks - here either computational heavy functions or collaborative tasks - can then be scheduled dynamically based on situational changes, seamlessly during the applications initially run on each robot into distributed application.

3.2 Demo Application

The application written to support the proposed scenario consists of three main tasks:

- The *Navigation* function, which was designed to capture a video stream from a vehicle's onboard camera and forward it to the object detection.
- The *Object Detection* function, which receives video frames and uses a trained image detection library to identify objects within a scene.
- The *Map Server* function, which acts as a communication hub between vehicles, constructing a common view of the region and objects provided by the partial views supplied by each vehicle.

The Navigation function runs permanently onboard the device due to its dependency upon the onboard camera. The Map Server is offloaded by the first vehicle entering the scenario, with subsequent arrivals connecting to it rather than offloading their own copy. The resource heavy Object Detection task initially executes onboard each robot device, but may be offloaded to a local edge site once certain trigger thresholds are met. It is this latter task that we use to measure both the benefits and drawbacks in application performance and resource utilization when migrating a task between onboard and remote execution.

3.3 PoC Implementation

In our proof of concept (PoC) implementation, several network-connected, battery-driven four-wheeled robots[6] were deployed on a flat area with opaque obstructions representing buildings or rocky landscape. Scattered across the environment we placed a number of hazardous objects that must be identified by the vehicles. In this layout, a robot entering from one side of the region could only see a limited number of these objects. In order to detect all the objects, a vehicle would need to roam in the area, or multiple vehicles could collaborate by pooling their partial view of the environment to create a complete picture in less time.

Figure 2 describes the deployment of the application as a vehicle arrives on site. Initially the three components of the application are running in natively compiled binary format on the resource-constrained UE robot hardware under the supervision of the offloading framework. In this scenario, the first robot arriving on site triggers an offload of the collaborative Map Server in portable Wasm format to the offloading cluster co-located with the network access point. It then proceeds to attempt to identify objects through the Object Detection function, which makes use of the YoloV4 neural network software for object detection (Bochkovskiy et al., 2020) based upon OpenCV 4.5.4[7]. Subsequent vehicles will attempt to deploy their own Map Server, but are redirected to connect to the existing server instead. This collaborative task is being shared among all robot UEs. Under certain conditions, e.g., a low battery level at a robot, offloading of the computational heavy Object Detection function is triggered as well. In this case, every robot will offload its own individual instance of this function in portable Wasm format, which can then be scheduled dynamically during operations, depending on the current situation. As shown in Figure 2, the computational offloading framework is deployed across both the UEs and the offloading site, and allows communication between the UE device and the offloading site through elastic functions as described in Section 2.3.

3.4 Experiment Setup

We conducted a number of experiments to examine the performance characteristics of offloading a particular function, Object Detection, using the described offloading framework. To collect relevant measurements from the devices, we developed a telemetry service. It permitted the gathering of metrics such

[6] Robots built from a *Freenove* 4WD Smart Car Kit
[7] https://opencv.org

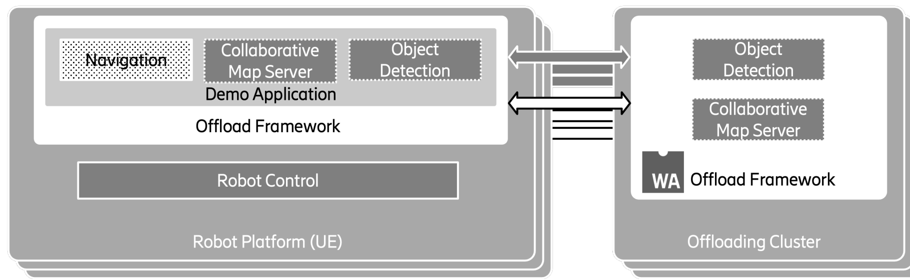


Figure 2: Application configuration showing tasks running both on the UE (left) and the edge cluster (right) within the computational offloading framework. Offloaded tasks communicate with parent application using elastic channels.

as CPU/GPU utilization, CPU/GPU temperature, network utilization, and power consumption from both the UE device and the host at the offloading site.

In order to reflect different UE capabilities in our experiments, we employed two different types of devices as base platform for the robot vehicle kits. The first one is a Raspberry Pi 3, which is an example of a device with limited computational capabilities. The second device is a Nvidia Jetson Nano, which is more powerful and comes with a 128-core Maxwell GPU. Wireless network connectivity was provided by a third party 5G adapter compatible with both the Nvidia Jetson and the Raspberry Pi, which connected to a private 5G base station and a 5G standalone (SA) core network with access to a lab environment acting as edge site. As a reference, we also performed offloading via wired LAN connected directly to the same edge site. Note that, compared to the LAN, the 5G connection included an experimental core network and three additional IP hops to reach the edge site. We list detailed specifications of the hardware and operating systems for both devices in Table 1. The table also includes information about the offloading cluster.

We have given several live demos of the search & rescue scenario with multiple robots serving as successful functional validation of the framework. For the follow measurements, however, we constrained the operation to provide a higher level of consistency. Specifically, the car-kit was removed not to interfere with the energy readings, and we used a recorded image stream taken from the camera whilst under testing. Furthermore, only one UE was used at a time during the measurement campaigns in order to avoid network and processor congestion.

Once all application components on the UE were started, a spin-up time of 150 sec was allowed for the system to reach a stable state. Subsequently, the object detection module was locally executed for 150 sec before being offloaded to the offloading cluster. Remote execution of the object detection task was then performed for 150 sec. This local and remote execution was repeated three times for each device.

Table 1: Experiment equipment.

Mobile devices

	Raspberry Pi 3B	**Nvidia Jetson Nano**
CPU	Quad Core 1.2GHz Broadcom BCM2837 64bit CPU	Quad-core ARM A57 @ 1.43 GHz
GPU		128-core Maxwell
RAM	1 GB	4 GB
5G Adapter	Waveshare 500Q-GL 5G HAT	
OS	Ubuntu 22.04 server	Ubuntu 20.04 Desktop

Offloading Cluster

CPU	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz 40 Core
RAM	62 GB
GPU	2560-core NVIDIA Tesla T4
OS	Ubuntu 22.04 server

4 RESULTS AND DISCUSSIONS

In our experiments, we focused on the measurable advantages of offloading compute-intensive tasks on two examples of mobile devices: the Raspberry Pi and Nvidia Jetson Nano. These measurements encompass various aspects and performance metrics of the devices, shedding light on the efficiency and behavior of the Object Detection module and the offloading solution throughout the experimentation process.

4.1 Raspberry Pi

Figure 3 presents the measurements of CPU, temperature, network and power usage of a Raspberry Pi during local and remote execution of the object detection module, represented as graphs of resource usage versus time. We have conducted a cycle of three local and three remote executions of this module, each of 150

sec duration. Those regions of the graphs are shaded dark (local) and light grey (remote execution), respectively. During the first instance of offloading, the object detection module is transferred from the device to the remote host and instantiated, thus imposing an additional resource burden equivalent to a cold start. Subsequent offload events simply activate the already transferred but dormant module code, more closely resembling a warm start event.

It can be observed from the Figure 3[a] that CPU utilization rises from idle to around 90% during local execution and returns to on average below 10% during offload execution of the computationally heavy module. When executed locally, the object detection task constitutes the bulk of CPU usage. On the other hand, when offloaded the application handles data transmission and reception over the network, along with associated serialization and de-serialization of this information. During the initial idle period, two spikes in CPU utilization are observed. These spikes occur as a result of the messaging involved in the initial setup of the offloading procedure and loading the application into memory. We conclude that for compute-intensive AI tasks like object detection, the offloading to a remote site significantly decreases CPU utilization of a resource constrained device, clearly outweighing the additional networking burden.

Figure 3[b] illustrates the temperature of the CPU during the experiments. During onboard execution the processor temperature rises significantly before dropping off sharply when offloading commences. Local execution was limited in time in order to prevent CPU throttling, which sets in at around 80 degrees Celcius on a Raspberry Pie, eventually rendering the device unresponsive.

Power usage is shown in Figure 3[c], clearly showing the relation between CPU usage and power consumption. When offloaded, power usage is slightly elevated over the idle value of approximately 2 Watts due to the energy drain of the 5G adaptor, but still significantly below the approximately 5 Watts consumed during local execution.

Figures 3[d] and 3[e] demonstrate the increase in data transmission between the application and the offloaded component as a consequence of remote execution of this task. During local execution the network usage is negligible. A spike in resource usage during initial local execution is visible, representing the transfer of the code module.

Figure 4 shows the Object detection response time when it is executed locally, remotely over 5G, and LAN respectively. This consists of the total time taken to detect an object on a supplied video frame, including data transmission time when offloaded. The observed variability can be partially attributed to the fact that the time taken to perform the object detection depends on parameters other than resource availability, such as the complexity of the video frame in question and the number of objects represented. The order of magnitude performance improvement when executed remotely, from almost 1300 ms locally on the device to about 90 ms (5G) or 65 ms (LAN), significantly validates the performance benefits of offloading from a resource constrained device.

Table 2: Power vs Performance for Raspberry Pi.

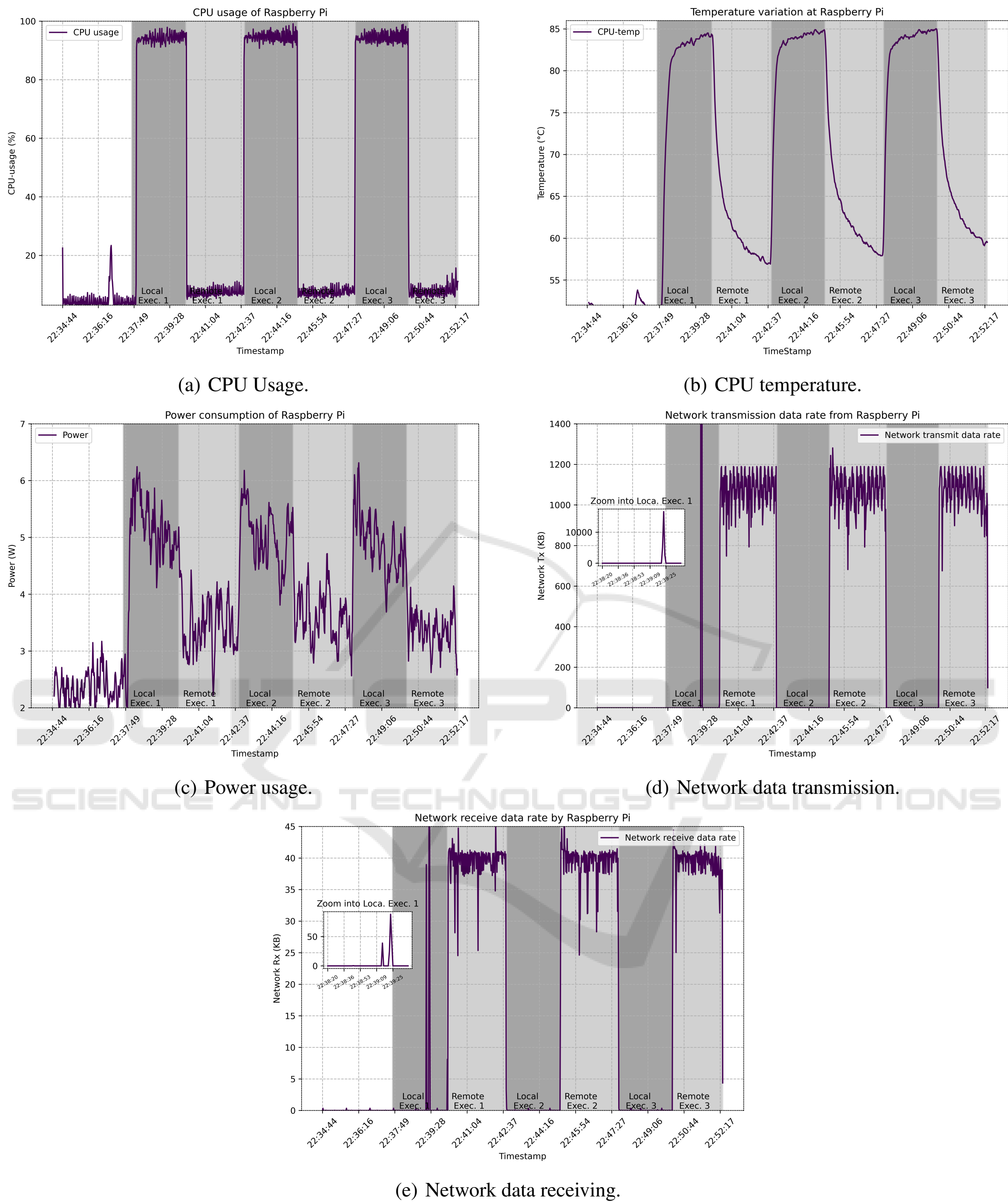
	Energy (in mWh per frame)
Local Execution	2.02
Remote Execution - 5G	0.09
Remote Execution - LAN	0.06

To further analyze the UE energy consumption of an Object Detection execution task, we calculated the energy required to receive a reply from the object detection task for each frame from the simulated video stream, measured in Milliwatt-hours (mWh). This metric, referred to as *mWh per frame*, provides a normalized measure of the energy efficiency of the object detection process from an UE point of view (i.e., high CPU utilization when processing onboard, vs lower CPU utilization but additional data transfer when offloaded). Table 2 presents the results, indicating that the energy consumption per frame is over an order of magnitude lower for remote execution compared to local execution for this device. This suggests that offloading the object detection task to a remote host results in more energy-efficient execution. Note also the slight difference when using 5G vs LAN connectivity. This gives and indication of the additional energy usage of the 5G HAT, which is negligible compared to the energy used to power the CPU on full capacity.

In summary, by offloading the object detection module and executing it remotely, the CPU usage, power consumption, and temperature of the Raspberry Pi have notably decreased compared to local execution. However, the network usage during remote execution has correspondingly increased, which, however, did not lead to a significant increase in power consumption. We conclude that offloading computational-heavy tasks is beneficial, reducing CPU usage, heat, and power consumption, which extend the device's longevity when running on batteries.

4.2 Nvidia Jetson Nano

Figure 5 illustrates the measurements of CPU/GPU utilization, temperature, network utilization, and power usage for the Nvidia Jetson Nano device, us-



(a) CPU Usage.

(b) CPU temperature.

(c) Power usage.

(d) Network data transmission.

(e) Network data receiving.

Figure 3: Measurement results on a Raspberry Pi.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

ing onboard and offloading periods of 150 seconds.

The CPU utilization of Jetson Nano is illustrated in Figure 5[a]. During the local execution of the object detection module, the relatively low CPU utilization compared to the Raspberry Pi of less than 30% can be attributed to the greater utilization of the integrated GPU, as evident from Figure 5[b]. A proportion of the CPU utilization can be attributed to data transmission from CPU memory to GPU memory. The same spikes due to messaging, setup and module transfer as in the case of the Raspberry Pi can be observed.

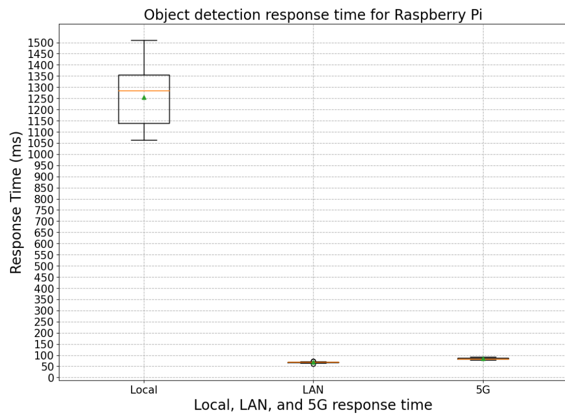


Figure 4: Object detection response time for Raspberry Pi.

Figure 5[c] shows CPU/GPU temperature during the experiments. The Jetson Nano utilizes a heat sink, which effectively dissipates heat and prevents significant temperature increases. As a result, the temperatures remain relatively stable throughout the experiments compared to the Raspberry Pi.

Figure 5[d] illustrates the power utilization of the Jetson Nano device. Onboard and offloaded power consumption follow the same patterns as on the Raspberry Pi despite the increased local performance, in this case approximately 6 and 2 Watts respectively. Also here the results show significant reduction in device power consumption as a benefit of offloading. As expected, Figures 5[d] and 5[e] confirm the significant increase in network traffic caused by offloading.

Figure 6 illustrates the results of the object detection response times for local and remote execution, using the same setup as in the earlier measurements. The average local execution time for object detection on the Jetson Nano device is measured at 60.01 milliseconds (ms), which is comparable to the average execution times for remote execution over 5G and LAN, which are 84.40 ms and 63.01 ms, respectively. This is due to the more capable hardware on the Nvidia Jetson, including a GPU. Also the extra delay of the experimental 5G network compared to the direct LAN connectivity is more relevant on these timescales.

We note that energy efficiency of remote execution from a UE device perspective is still significantly improved when the object detection task is offloaded, see Table 3. These results indicate that computational offloading of compute-intensive tasks may not result in meaningful performance improvements when the UE has similar compute capabilities than the remote site. However, the reduced power drain and heat dissipation on the UE may prove to be a decisive factor when choosing whether to offload or not.

Table 3: Power vs Performance for Jetson Nano.

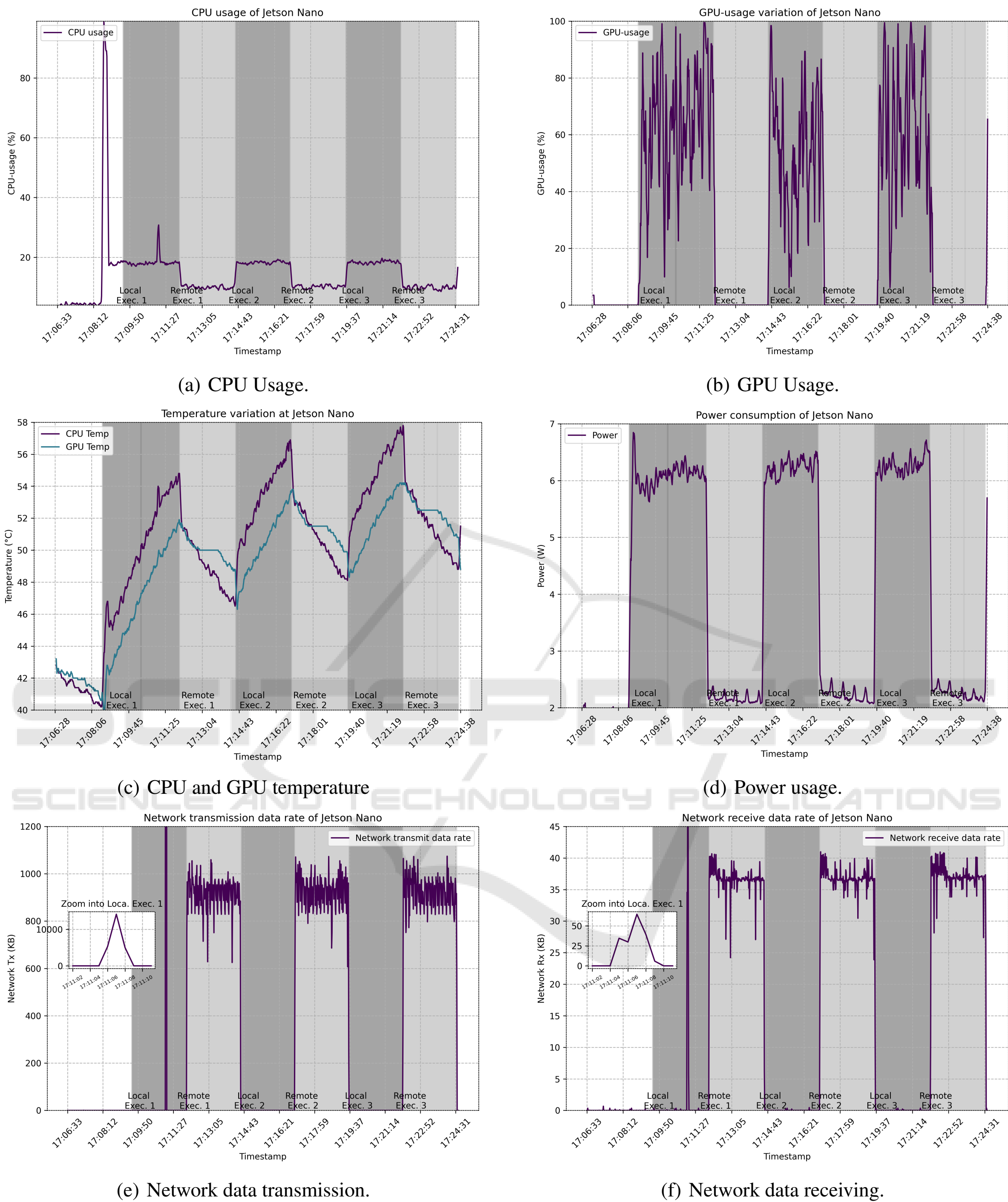
	Energy (in mWh per frame)
Local Execution	0.101
Remote Execution - 5G	0.058
Remote Execution - LAN	0.050

5 RELATED WORK

Concepts related to computational offloading have been significant subject of research in the last decade. An early survey on mobile edge computing (Mach and Becvar, 2017) (MEC, later re-labelled multi-access edge computing), analyzed more than 100 papers, focusing on the decision mechanisms for computational offloading as well as the allocation of computing resources. The assumed virtualization technology and granularity in these papers has been VMs, which does not meet our requirements of lightweightness and portability. However, a few general lessons learned from this paper are relevant also for our proposed solution, e.g., that VM migration is impractical if a sufficiently large amount of data needs to be transmitted, and partial offloading can save significantly more energy at the UE compared to full offloading.

The survey by Lin et al. (Lin et al., 2019) reviews research on computation offloading, and identified three main groups of challenges: application partitioning; task allocation and resource management; and distributed task execution. In terms of application partitioning, challenges related to lightweight programming models as well as the partitioning granularity are pointed out. In our solution, we address this challenges with a novel programming model in Section 2.3, and the flexibility to offload arbitrary code granularity, including both whole tasks or components down to the level of methods/functions. Task allocation and resource management with optimal strategy are not part of the present paper, but we are working on a related publication. With respect to distributed task execution, our solution deviates from the VM and container based solution discussed in this survey by building our solution based on portable, lightweight and secure Wasm runtimes.

Another survey focused on service migration strategies in the context of MEC (Wang et al., 2018). As part of this survey, execution environments to host MEC applications are compared. The conclusions are that VMs have good isolation properties but are large in terms of footprint and slow to boot and run. Containers, on the other hand, have a smaller footprint and faster startup times, but perform sub-optimally across operating systems and hardware platforms. Finally,



(a) CPU Usage.

(b) GPU Usage.

(c) CPU and GPU temperature

(d) Power usage.

(e) Network data transmission.

(f) Network data receiving.

Figure 5: Measurement results on a Nvidia Jetson Nano.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

agent-based systems have very small footprint and rapid boot and runtime performance with the promise of convenient administration. However, these systems are still of experimental nature without contemporary stable existing frameworks. Together with our review of several concepts related to VMs or container migration (Junior et al., 2020)(Machen et al., 2018)(Benjaponpitak et al., 2020), we conclude that literature confirms our our choice of Wasm a promising execution environment.

While the above surveys focus on mobile edge computing based on classical Cloud technologies

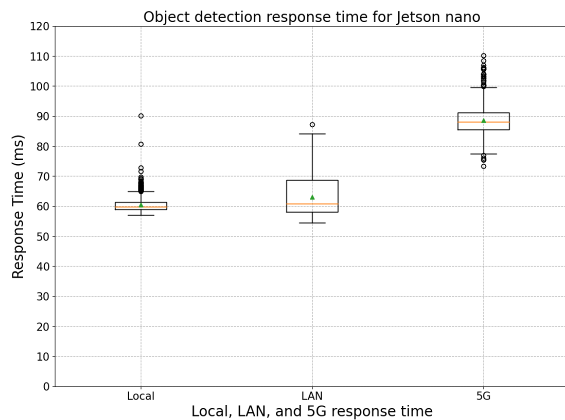


Figure 6: Object detection response time for Jetson Nano.

such as VMs and containers, there is more recent work applying lightweight, portable runtimes such as Wasm for computational offloading. The authors of (Hoque and Harras, 2022) argue along these lines, whereas (Ménétrey et al., 2022) point out benefits of Wasm for faster migration time due to smaller memory footprint, and (Long et al., 2021) show better performance of Wasm runtimes than OS containers.

Authors of (Tachibana et al., 2022) suggested application-driven dynamic task offloading by task classification on real-time basis. Application-driven offloading allows the application developers to decide when and where to offload, but this might not always be optimal. The network or computation provider has to be able to offer only edge hosts or nodes which are capable to execute the offloaded task and the developers and/or users should be able to choose the node. The ability to choose arbitrary nodes would be especially needed in multi-provider environment where providers might compete for users.

Host to host Wasm based modules migration has been proposed in (Nieke et al., 2021), where authors migrate mobile agents - application server instance. Host to host migration is very important for load balancing or compliance with QoS of the application-module communication during user mobility, but the task has to be already represented as a Wasm module in order to migrate it which is not optimal for UE to host migration. In this case, it is a natural choice for the task is to be executed in the original device in native code. In Section 2.3 we proposed a UE to host migration method addressing this issue.

As mentioned, Wasm modules have benefits for compute migration, but are not fully protected from threats from other malicious tenants or the service provider. Security measures like utilizing Trusted Execution Environments and Wasm enclaves are addressed in (Ménétrey et al., 2021)(Pop et al., 2022).

6 CONCLUSION AND NEXT STEPS

We presented a solution that enables an application running on a mobile device to dynamically offload critical parts of its functionality, as identified by the developer, to a remote site, seamlessly during application runtime. We identified a number of requirements, designed, built, and verified an according solution. Our solution consists of a novel programming model and associated toolchain, with particular focus on developer ease of use, as well as a distributed execution framework based on portable and lightweight standalone Wasm runtimes.

We demonstrated and evaluated the viability of the presented solution, providing a compelling case for performance improvements through ad-hoc computational offloading in resource-constrained devices and reduced power consumption in general, despite the increased network usage. We realized and demonstrated the presented solution using two examples of UE devices with varying computational capacities (Raspberry Pi and Nvidia Jetson Nano). We also applied two different categories of connection between the device and the edge servers - wired LAN as the baseline reference, as well as standalone (SA) 5G wireless as a realistic, futureproof wide-area wireless connectivity example. For both devices, module offloading reduces device power usage. The Raspberry Pi extends its compute capability for computation heavy tasks, whereas the Jetson Nano does not see this benefit due to its onboard GPU. However, in both cases, device energy efficiency is improved.

We assert that this type of use-case agnostic solution - offering sandboxed remote computational resources to any application on a mobile device basically anytime and anywhere - has the potential to target even the long tail of regional enterprises and developers that usually would not engage in the heavy burdens of deployment, management, and contract handling related to edge solutions. We are aware that there are still numerous open issues to reach this vision, and we conclude this article by pointing out a few open research questions we plan to address as next steps. For instance, an important building block to make this solution viable is a comprehensive cost function service which can intelligently decide when and where to offload in order to provide the best overall performance for the application, device and user. As security and isolation are of utmost importance in remote computation scenarios, we also work on extending our solution with a more comprehensive security architecture based on an detailed analysis of security and privacy threats. Moreover, we plan to

further explore specific features in the areas described in this work, such as granular isolation of Wasm modules into TEEs similar to (Nieke et al., 2021). While we had very positive experiences of using Wasm runtimes when building our prototype, there are still issues with proprietary system interfaces to host function related to efficient realization of the execution handler. We thus encourage the networking community to engage in consortia like the W3C WASI working group to make sure that future Wasm related standards meet the requirements of advanced use-cases like proposed in this paper. Finally, we are exploring alternatives to offering computational offloading as a service via cellular mobile broadband networks,e.g., 5G advanced or 6G (Wikstrom et al., 2022). This includes integration points like service registration and discovery, management of the offloading service user plans and associated mobility support, control plane handling of offload triggers, as well as crucial authentication and billing mechanisms.

REFERENCES

Benjaponpitak, T., Karakate, M., and Sripanidkulchai, K. (2020). Enabling live migration of containerized applications across clouds. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*.

Bochkovskiy, A., Wang, C.-Y., and Liao, H.-Y. M. (2020). Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*.

ESTI (2023). ETSI TS 123 558 V17.7.0 5G; Architecture for enabling Edge Applications (3GPP TS 23.558 version 17.7.0 Release 17). Technical Report TS 123 558, European Telecommunications Standards Institute.

Guanciale, R., Paladi, N., and Vahidi, A. (2022). Sok: Confidential quartet - comparison of platforms for virtualization-based confidential computing. In *IEEE Int. Sym. on Secure and Private Execution Environment Design (SEED)*.

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. (2017). Bringing the web up to speed with webassembly. *SIGPLAN Not.*, 52(6):185–200.

Hoque, M. N. and Harras, K. A. (2022). Webassembly for edge computing: Potential and challenges. *IEEE Communications Standards Magazine*, 6(4):68–73.

Huang, X., Tsiatsis, V., Palanigounder, A., Su, L., and Yang, B. (2021). 5g authentication and key management for applications. *IEEE Communications Standards Magazine*, 5(2):142–148.

Junior, P. S., Miorandi, D., and Pierre, G. (2020). Stateful container migration in geo-distributed environments. In *IEEE Int. Conf. on Cloud Computing Technology and Science (CloudCom)*.

Lin, L., Liao, X., Jin, H., and Li, P. (2019). Computation offloading toward edge computing. *Proceedings of the IEEE*, 107(8):1584–1607.

Long, J., Tai, H.-Y., Hsieh, S.-T., and Yuan, M. J. (2021). A lightweight design for serverless function as a service. *IEEE Software*, 38(1):75–80.

Mach, P. and Becvar, Z. (2017). Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communications Surveys & Tutorials*, 19(3):1628–1656.

Machen, A., Wang, S., Leung, K. K., Ko, B. J., and Salonidis, T. (2018). Live service migration in mobile edge clouds. *IEEE Wireless Communications*, 25(1).

Ménétrey, J., Pasin, M., Felber, P., and Schiavoni, V. (2022). Webassembly as a common layer for the cloud-edge continuum. In *Workshop on Flexible Resource and Application Management on the Edge: FRame*.

Ménétrey, J., Göttel, C., Khurshid, A., Pasin, M., Felber, P., Schiavoni, V., and Raza, S. (2022). Attestation mechanisms for trusted execution environments demystified. In *IFIP Intr. Conf. on Distributed Applications and Interoperable Systems: DAIS*.

Ménétrey, J., Pasin, M., Felber, P., and Schiavoni, V. (2021). Twine: An embedded trusted runtime for webassembly. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 205–216.

Nieke, M., Almstedt, L., and Kapitza, R. (2021). Edgedancer: Secure mobile webassembly services on the edge. In *Int. Workshop on Edge Systems, Analytics and Networking*, EdgeSys ’21.

Pop, V. A. B., Niemi, A., Manea, V., Rusanen, A., and Ekberg, J.-E. (2022). Towards securely migrating webassembly enclaves. *Proceedings of the 15th European Workshop on Systems Security*.

Tachibana, T., Sawada, K., Fujii, H., Maruyama, R., Yamada, T., Fujii, M., and Fukuda, T. (2022). Open multi-access network platform with dynamic task offloading and intelligent resource monitoring. *IEEE Communications Magazine*, 60(8):52–58.

Wang, S., Xu, J., Zhang, N., and Liu, Y. (2018). A survey on service migration in mobile edge computing. *IEEE Access*, 6:23511–23528.

Wikstrom, G., Persson, P., Parkvall, S., Mildh, G., Dahlman, E., Rune, G., Arkko, J., John, W., et al. (2022). 6g—connecting a cyber-physical world. *Ericsson White Paper*, 28.