VEDLIoT

Very Efficient Deep Learning in IoT

ICT-56-2020 — Next Generation Internet of Things

D5.3
Second implementation of Security, Safety and Robustness mechanisms and tools

Version 1

Document Information	
Contract Number	957197
Project Website	https://vedliot.eu/
Dissemination Level	PU (Public)
Nature	R (Report)
Contractual Deadline	31 January 2023
Author	Piotr Zierhoffer (ANT)
Contributors	Wojciech Rajtar (ANT), Kamil Zwarycz (ANT), Marcelo Pasin (UNINE), Jämes Ménétrey (UNINE), Pascal Felber (UNINE), Valerio Schiavoni (UNINE), Carina Marcus (VEONEER), Olof Eriksson (VEONEER), Anum Khurshid (RI.SE), Shahid Raza (RI.SE), Tiago Carvalho (FC.ID), José Cecílio (FC.ID), Bakary Badjie (FC.ID), Erik Funke (CHR)
Reviewers	Mario Porrmann (UOS), Pedro Trancoso (Chalmers)

The VEDLIoT project has received funding from the European Union's Horizon 2020 research and innovation programme under the Grant Agreement No 957197.

Change Log		
Version	**Date**	**Description of Change**
c9f544a	2022-11-14	Initial structure
31490ec	2022-11-14	Gitlab CI setup
7432512	2022-11-14	Add GH CI
acd337f	2022-11-14	Document stubs
e95075b	2022-11-22	Added UNINE titles and contrib proposal
6233f22	2022-12-14	Initial draft for SIRE's chapter
6270287	2022-12-19	Veeoneer submission
8e1b672	2022-12-19	Add file with RI.SE submission
5448bfe	2022-12-19	Add figures to RI.SE chapter
9267045	2022-12-20	Add WebAssembly chapter
2687916	2022-12-20	TOCTOU - use numbered sections
1ff0b3e	2022-12-20	Rename Renode chapter
d997758	2022-12-20	Update robustness chapter
b99f4a4	2022-12-20	Update of tables and figures
13d41ac	2022-12-20	Add robustness figures
6a0982a	2023-01-03	Add chapter: Co-simulation improvements and testing in Renode
cd46035	2022-12-28	Review WebAssembly chapter
f75a33b	2022-12-28	Add full names to abbreviations
93ad26f	2022-12-28	Review TOCTOU chapter
6b5bce2	2022-12-28	Review Runtime Monitoring
64bc626	2022-12-29	Review SIRE
5f911ce	2022-12-29	Review Robustness
3a57d57	2023-01-05	Modified WebAssembly intro, added ref to wasm-continuum paper
5849e8e	2023-01-05	Reformatted Robustness chapter
b7ae091	2023-01-06	Review the introduction of Robustness chapter
30e44c8	2023-01-10	webassembly: Minor review
0914fb6	2023-01-10	toctou: Minor review

c4e392e	2023-01-10	Reordered chapters
eba9930	2023-01-10	Review of Robustness chapter
eb5ff9b	2023-01-10	Use paragraph rather than subsubsection in TOCTOU chapter
71ea2f7	2023-01-10	Review SIRE chapter
48af1f9	2023-01-11	Add future work of Task T5.2 to the conclusion
41a5b76	2023-01-10	Review and edit intro in Runtime Monitoring
013c9b7	2023-01-10	Add Conclusions chapter
af49b2a	2023-01-11	Add renode-issue-reproduction-template reference
4a5e90f	2023-01-11	Review and expand Conclusions chapter
bd1df49	2023-01-11	Add future plans for SIRE
6e2a8a6	2023-01-12	Intro clarification in Runtime Monitoring
c8cf7f1	2023-01-12	Add future work to TOCTOU paragraph in Conclusions
0f013df	2023-01-16	Add conclusions section to Runtime Monitoring
e18b13c	2023-01-16	Review Renode chapter
02f9eb4	2023-01-16	Fix SIRE figure placement
3769c07	2023-01-16	Reference previous deliverables
7f21d8b	2023-01-16	Renode review
b74bb6e	2023-01-12	Add Intro chapter
b03dfc5	2023-01-17	robustness: Format the tables, fix labels
6bfdf7b	2023-01-17	Finish intro
979fc87	2023-01-17	Add missing figure
cd3ef93	2023-01-17	Add contributors and changelog
9c3c14d	2023-01-17	Minor fixes in Runtime Monitoring
359c9c7	2023-01-17	Add reviewers
41c3bb4	2023-01-31	Applied changes suggested by reviewers

This log reflects major revision numbers from Gitlab (version control software used).

Table of Contents

List of Figures

List of Tables

1 Executive Summary

The work described in this document, "D5.3: Second implementation of Security, Safety and Robustness mechanisms and tools", is part of Work Package 5 (WP5) of the European Union funded **VEDLIoT** project and focuses on development of the part of the platform ecosystem related to tools and mechanisms for ensuring security, safety, and robustness of VEDLIoT solutions.

The report is organized as follows:

Chapter 2 provides an introduction, overview and summary of the structure and content of the remainder of the document. The second chapter is an introduction to the project and WP5's goals with reference to the tasks contained in the Work Package.

Chapter 3 describes the capabilities of the WebAssembly as a go-to technology to provide interoperability in the cloud-edge continuum, in contrast to other popular solutions aiming to solve similar problems. We discuss various implementations of Trusted Execution Environment concept and the place of WebAssembly in this landscape. Finally, we present our novel research on Wasm performance across different architectures and discuss possible technological limitations in its current state.

Chapter 4 presents the issue of Remote Attestation of IoT devices in the context of the well known Time-Of-Check to Time-Of-Use race condition. To address this issue, we propose the AutoCert mechanism, combining the Remote Attestation procedures with the classic Public Key Infrastructure authentication processes. We also discuss experimental results of the Proof-of-Concept implementation.

Chapter 5 presents the details of the implementation of SIRE, an infrastructure providing services crucial for a secure IoT system, such as remote attestation, auditability, membership management and others. As SIRE achieved its full functionality, we delve into the description of various modules and their responsibilities, concluding with the analysis of its performance and scalability.

Chapter 6 focuses on software testing, specifically addressing latest developments in the ML IoT space and describes the improvements in co-simulation capabilities of Renode, an open source simulator of IoT and embedded systems. We discuss the proposed automatic testing infrastructure and recent improvements in the simulation of a Machine Learning accelerator developed in the Work Package 4.

Chapter 7 discusses the issue of robustness in Machine Learning algorithms and reviews various techniques for its evaluation. It proposes a network block segmentation strategy that, when applied to the $alpha - beta - CROWN$ algorithm, provides better performance, result quality and lower resource cost than the classic approach.

Chapter 8 details the process of monitoring and mitigation strategies for run-time errors. The chapter divides many possible errors into groups based on their source, proposes solutions to combat errors from each group, and compares the efficiency of implemented strategies to the existing standards.

Finally, in Chapter 9, we sum up our significant achievements and outline our plans for future work and further development of proposed solutions.

2 Introduction

VEDLIoT (Very Efficient Deep Learning in IoT) is developing a framework for the Next Generation of IoT architecture, using deep learning algorithms distributed throughout the IoT-edge-cloud continuum. The proposed new platform, which includes innovative hardware accelerators and software toolchains, is expected to bring significant benefits to a large number of applications, including industrial robots, self-driving cars, smart homes, and predictive maintenance of industry and assisted living at home.

The growing criticality of automated systems depending on Artificial Intelligence requires us to push for quality of these systems. In Work Package 5 (WP5) we focus on several key aspects of these systems: security, safety, and robustness. This document follows up from the second deliverable, "Extended design and first implementation of Security, Safety and Robustness mechanisms and tools" (D5.2) [62] as it takes the implementation of these aspects and mechanisms to the next level.

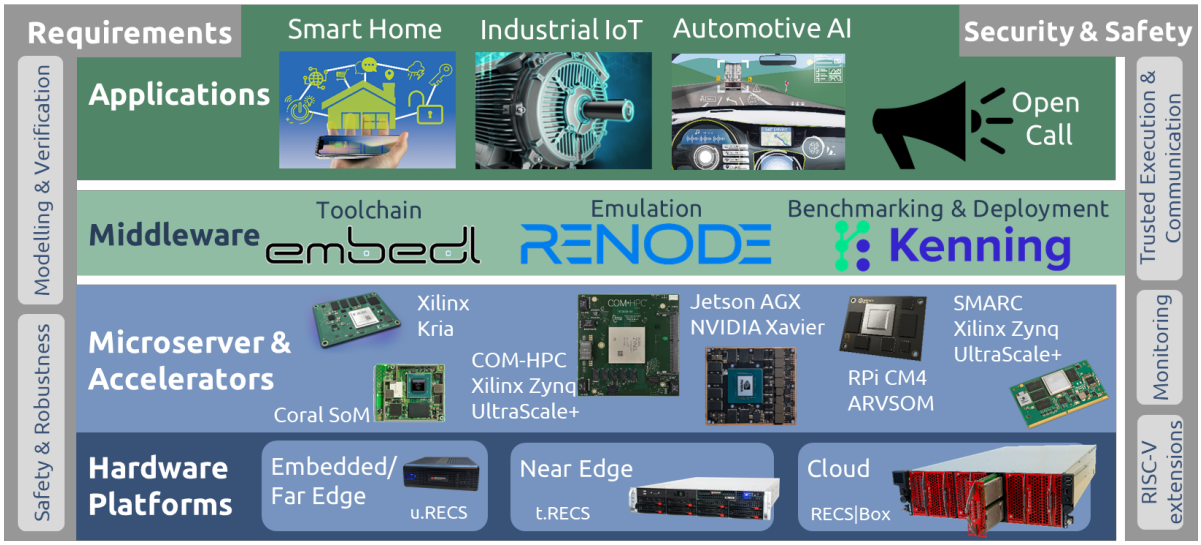


Figure 2.1. VEDLIoT abstracted overview.
WP5 components appear mostly on the right-hand side.

Task 5.1 End-to-end attestation of distributed trusted environments Duration: M1-M36

Building up on the work on attestation conducted in previous periods of the VEDLIoT project, in this deliverable we discuss two approaches to attestation.

The first approach, described in Chapter 4, aims to address the well-known TOCTOU problem. TOCTOU, or Time-Of-Check to Time-Of-Use, is a race condition observable in various kinds of system, in which there is a period of time between verifying access to a resource and actually using it. This period can be exploited by malicious parties to interfere with this resource access.

As TOCTOU is a valid problem in the attestation scenarios we analyze, we propose AutoCert, a mechanism described in Section 4.1 that aims to mitigate it using a well-known Public Key Infrastructure. In Section 4.2 we discuss our experimental results.

Another approach to the attestation problem, but also addressing membership management, auditability and coordination primitives, is the Trusted Verifier Service, or SIRE, described in Chapter 5. With most of the functionality of SIRE already in place, in Section 5.2 we present technical details of the implementation of various available modules. Later, in Section 5.5, we discuss the evaluation procedure and test results both for simple data reads/writes and for more complex join/attest operations. We conclude with the introduction of one of our goals - implementation of a SIRE-based application in the autonomous vehicle field (see Section 5.6).

Task 5.2 Security support for distributed execution and communication Duration: M4-M36

In T5.2 we continued our work on a unified, secure execution and communication environment that would span the entire cloud-edge continuum of the class of systems considered by VEDLIoT. In Chapter 3 we reintroduce the concept of WebAssembly as an execution platform, then in Chapter 3.3.2 we discuss our involvement in combining Wasm payloads with trusted execution environment, exemplified by Twine [84] and WaTZ [85] - trusted Wasm runtimes for Intel and Arm solutions.

We present performance analysis of Wasm applications in Section 3.4. We discuss two benchmarks: PolyBench/C (see Section 3.4.1) and SQLite-based benchmark suite (see Section 3.4.2).

We are still working on evolving Twine. This work is done in cooperation with Credora, an American industrial partner developing fintech applications using software derived from Twine. Thanks to this cooperation, we received a HiPEAC Tech Transfer Award in December 2022 [13]. HiPEAC looks for examples of technology research which have made it to the market, whether through licences, services or even the launch of a company. Winners receive a monetary award and get coverage in the HiPEACinfo magazine, and the HiPEAC website and social media accounts.

Task 5.3 Simulation platform for development and testing Duration: M7-M30

This task aims at building a platform that will enable extensive testing at every stage of the development of VEdLIoT to enable proper security mechanisms. The main goal of this task is to provide a tool to project participants and future users of VEDLIoT that will serve as a testing and simulation solution, speeding up the platform's development process by providing a reliable and deterministic testing environment, as described in the Chapter 6. Thorough testing of solutions like VEDLIoT is an extremely important issue as critical IoT applications need to satisfy high-level security requirements. The platform mentioned above is being built on top of the Renode Framework [21], an open source simulation environment developed by Antmicro, capable of simulating complex multi-node, heterogeneous, interconnected IoT systems. The tool will be able to simulate the platform developed as part of Task 4.7. Users will be able to use the same unmodified software that will be running on the target hardware, which reduces the overhead often associated with extensive software testing and, as a result, encourages the user to use the tool as it can be easily integrated into their existing workflows. Renode will provide additional tracing, inspection, and debugging capabilities that may not be available on the target platform. To improve development experience we have greatly extended our co-simulation capabilities, polishing support for currently available buses (see Chapter 6.1) and adding new fea-

tures like two-way communication for peripherals acting as both transaction recipients and initiators (see Chapter 6.1.3). All improvements to the Renode Framework will be released publicly on a permissive, open source license.

Task 5.4 Continuous integration workflow
Duration: M18-M36

With the aim of to prepare a system that can be used by project participants to test and verify the software for the SoC (System-on-Chip) created as part of Task 4.6 and Task 4.7, we have started our work described in Section 6.2.2. The system will be based on the platform developed in Task 5.3. It will provide continuous integration capabilities, ensuring the fitness of the software at each stage of development. The tests will range from verification of basic functionality to analysis of the entire system behavior, including remote attestation and encryption mechanisms. The system will allow project members to add their own tests to the suite and share them with other participants. Testing suites will be executed using a CI (Continuous Integration) system based on GitLab. The implementation of cloud-based testing provides significant improvements compared to local testing, like almost endless scalability, flexibility, and the ability to collaborate on the project regardless of the physical location. Tests will be executed in a deterministic manner, in a strictly controlled environment, for every version of the software, using unmodified binaries. Users will be able to provide input for sensors and other available interfaces, testing the system's behavior in synthetic, possibly hostile conditions, thus improving the robustness and security of the system.

Task 5.5 Safety and Robustness
Duration: M7-M36

In the course of VEDLIoT we conducted analyses of adversarial attacks on Deep Neural Networks (DNN). DNNs deployed in safety-critical applications need to be well understood in terms of their susceptibility to malicious inputs and decisions made in extreme scenarios. We analyzed the available solutions for verification of robustness of such networks in Chapter 7.

We also propose a network block segmentation approach, which, when applied to the $alpha - beta - CROWN$ algorithm, helps us significantly improve performance of computation. In Section 7.7 we present our experimental network and, in Section 7.8, we discuss the results of evaluation of different algorithms.

In the last technical Chapter 8 we discuss safety in the context of a real-life, safety-critical use case - Pedestrian Automatic Emergency Braking (PAEB) system, developed as part of T7.4. We discuss monitoring of ML systems via the quality of input and output data, and we present the categorization of run-time errors (see Section 8.1).

Finally, we analyze various error mitigation strategies in Section 8.3 and conclude by presenting current trends in safety standardization.

3 Common Layer for the Cloud-edge Continuum

VEDLIoT's WP5 aims at developing several system-level tools for adding dependability and security to the execution of machine-learning applications on a distributed environment. One particular aspect developed is the support for secure execution and communication of critical code on edge devices.

For that matter, we leverage hardware features for trusted execution environments combined with well-established dependability techniques to support the tools developed in other work packages (accelerator design, hardware, deep learning, and use cases). In terms of execution environment, VEDLIoT has provided a number of tools for supporting secure execution of code in untrusted (edge) devices.

In this chapter, we develop our main idea of using WebAssembly as the interoperable environment and trusted hardware as the means for achieving secure execution. This chapter is largely inspired by a scientific paper, published in the Workshop on Flexible Resource and Application Management on the Edge of the High Performance Distributed Computing (HPDC) conference in 2022 [76].

3.1 WebAssembly as a common layer

In the last decades, numerous Web applications have been developed to be accessed from anywhere, including personal computers and smartphones. Many of these programs were later moved to the cloud to be more practical or cheaper to maintain. Other applications were initially designed for the cloud for scalability and availability while relying on the cloud's naturally distributed and replicated nature. No matter the reason, *cloud computing* has become one of the main infrastructures supporting applications today.

The cloud was not always big as it is today. It started simpler, with a handful of providers and basic services such as virtual machines and virtual storage. Numerous cloud providers have come to exist to supply today's enormous cloud market demand. Nowadays, some applications are built to exploit the cloud as a heterogeneous environment. They can exploit it to obtain, for example, lower latency, more resilience, or legal compliance. With a growing number of multi-cloud applications, dealing with different cloud providers and technologies has become a frequent issue.

Telecommunication companies started to deploy a more distributed infrastructure, with smaller, cloud-like clusters closer to the consumers of network-based services, to improve the latency of their services. Local administrations and other infrastructure providers such as energy and transportation followed suit, deploying small groups of rather powerful computing devices close to the human activity they support. The use of these highly distributed devices has been collectively named *edge computing* [94].

To complete today's scenario, billions of sensing and actuating devices have been deployed, called the *Internet-of-things*, or IoT. These devices are often tiny, with limited processing capabilities. They execute simple tasks, such as sensing a temperature or turning on and off a light bulb. Yet, they are connected to the Internet, often coordinating their function using some edge device and connecting users through a cloud service.

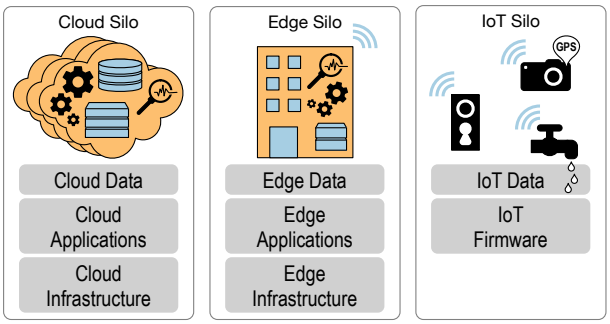


Figure 3.1. Independent cloud, edge, and IoT silos.

IoT, edge and cloud infrastructures form together what has been called the *cloud-edge continuum* [37]. This collective infrastructure is far from seamless today. They actually exist in separate silos, dominated by proprietary solutions, as shown in Figure 3.1. Developers of applications that span the whole continuum must implement specific solutions for each silo, often built using incompatible software components. The absence of a seamless environment makes it much harder to use the cloud-edge continuum. Yet, a good part of VEDLIoT's intentions is to help building distributed machine learning applications, partly running in the edge and cloud devices, and using data streams form the IoT.

Security has always been a component of applications shared among multiple users. Traditional security is used to deal with encryption, authentication, and access control, and many established tools exist. With the advent of the cloud, which is accessed through the Internet, security has become a fundamental component of all applications. Providers, developers, and users must be able to trust in the whole continuum — cloud, edge, and IoT — to ensure that their data is safe and their computations are correct.

We advocate that the technology provided by WebAssembly is adequate for implementing seamless applications across most hardware devices and software environments of the cloud-edge continuum, with the appropriate level of security. To support our claims, we present a comparison of WebAssembly running benchmark suites on two processor architectures. To the best of our knowledge, we were the first to compare WebAssembly performance on different CPU architectures.

Modern hardware allows running WebAssembly while achieving good performance and high levels of security. When paired with trusted computing, a technology that guarantees the confidentiality and integrity of secure applications, WebAssembly abstracts the complexity of software development while offering a trustworthy execution environment. Nonetheless, many pieces are still missing from a fully-fledged cloud-edge continuum. Consequently, we also shed some light on the work yet to be covered by the research and industrial community.

In the particular case of VEDLIoT, we have several use cases developed in Work Package WP7 that can benefit from a seamless common layer spanning over all the continuum. As an example, the Automotive AI Use Case relies on four types of processing nodes: a camera processing device, the vehicle central processing unit, a 5G base station edge processor, and a cloud server. It is already complex enough to simply deploy an application on such a different (and incompatible) set of devices. Programming them involves knowing and using several different programming environments,

which may even impose specific languages, increasing heterogeneity. With devices deployed on the field, such as the vehicle or the base station, security becomes a clear need. A common layer and the added security features as proposed here can clearly simplify the task of developing and deploying such applications.

In the following, we start by developing the current drawbacks of existing software architectures in more detail. We then present WebAssembly and its advantages for executing applications in the cloud-edge continuum. We complement our presentation with a preliminary performance comparison when executing selected applications using WebAssembly on two processor architectures to prove our claims of cloud to edge viability. We conclude with a few ideas for future work on the subject.

3.2 Building the cloud-edge continuum

A typical cloud environment is a rather complex system containing numerous (and different) hardware components. Such components are exploited using extensive collections of software, managed by large engineering teams, and shared by many tenants. Adding the edge (and IoT) to the picture pushes size and heterogeneity to another dimension. An ideal seamless cloud-edge continuum should offer a lightweight execution environment with a similar (or even identical) software and hardware interface, allowing unmodified code to be executed in any machine in the system.

Some initiatives already exist for a common environment for cloud, edge, and IoT silos. The Java Virtual Machine (JVM) [104] is one of the first practical common environment implementations that address the issue of applications running on heterogeneous underlying systems. To a considerable extent, the JVM is today one of the most comprehensive choices, with implementations for commodity servers to embedded devices. Still, the JVM supports very few programming languages and adds substantial performance overheads compared to the native execution of C programs. Java programs depend on a vast number of class libraries, imposing a large memory footprint for executing even the simplest of programs. Containers appeared more recently [32] as an alternative for running applications in heterogeneous environments. Still, containers are defined for specific architectures and a particular operating system interface. One needs to rely on recompilation to obtain containers that can run, for instance, on Intel and Arm devices (respectively popular as cloud and edge devices). WebAssembly has the generality of JVM and the lightness of containers, allowing to build multi-platform software that can execute with negligible performance losses and small memory footprints.

To automatically deploy applications in a distributed system, one has to deal with aspects such as admission control and resource management, as well as monitoring and optimising the use of the devices to compute and communicate. We are unaware of any practical, specific tool spanning the whole cloud-edge continuum, but we assume it would be straightforward to adapt many of the existing tools developed for the cloud [38], provided that the underlying systems become more homogeneous. Also, a few authors have already started working on models for integrating cloud and edge devices into a seamless system [37, 31, 33]. We do not address the issue in this chapter. Instead, we propose to use a homogeneous runtime model to close the gap at low-level.

As said, security has become an essential issue in cloud systems. Application users

need guarantees that their data's confidentiality and integrity are respected. These guarantees are hard to provide in a multi-tenant system, where co-tenants may abuse the system's vulnerabilities to uncover (or infer) someone else's application data. It is even more complex when the infrastructure provider is curious, as it has all the administrative power needed to inspect all contents in all physical machines. On the other hand, infrastructure providers wish to be protected from malicious tenants, who may want to exploit the infrastructure vulnerabilities for their own profit.

Edge computers are much more distributed when compared to the cloud. They are installed in user buildings, shared infrastructures, or even next to roads, making it impossible to maintain physical control over resources. Edge administrators have physical access to the edge devices they manage, with similar powers to cloud providers. On the other hand, users are in the proximity of the edge devices and may even physically abuse them. Edge-based infrastructure offers far fewer guarantees than the cloud.

Most recent versions of popular computer architectures include a form of a *Trusted Execution Environment* (TEE), a practical solution for establishing trust. Such TEEs allow code execution in a separated hardware section, where access from other software is architecturally impossible. A TEE can execute a program and protect its data so that a machine administrator cannot access it. Current hardware implementations may include an extra execution mode in the processor or even memory encryption for TEE data. The most popular implementation of TEE today is Intel's Secure Guard Extensions (SGX) [75], for which commercial cloud services such as Azure Confidential Computing [78] already exist. A similar solution is also necessary for edge deployments, where the most popular architecture is Arm, which in turn offers TrustZone [26] as a TEE. The existence of proprietary and incompatible solutions in the underlying hardware makes it harder to reuse trusted software from cloud to edge and vice versa.

Confidential containers could be a practical alternative for deploying applications on the cloud-edge continuum, as proposed by Scontain [27]. They are similar to traditional containers, except they run entirely inside a trusted environment. However, like other containers, they are platform-dependent. Also, they are costly in terms of the resources needed in many cases, as they may incorporate substantial amounts of operating system features.

Microsoft's Azure Sphere [79] follows the same idea but offers a unified programming model and support for certain trusted execution technologies. As a proprietary solution, it is heavily dependent on other Microsoft services. It offers a high-level interface but only supports a few programming languages.

By proposing WebAssembly as the execution model combined with trusted execution environments, we can offer a seamless portability base for running trusted applications. The same base can be used to deploy applications on edge or cloud devices with similar security guarantees. Besides, it has been shown that a WebAssembly TEE enables a double-sided sandbox [47], providing better security for the provider and the tenants.

Many different IoT infrastructures have been deployed and continuously generate data, feeding cloud applications worldwide. Components in the application chains (IoT to edge to cloud) may be updated independently, adding new functionalities and

removing vulnerabilities. Particularly in this application area, we observe the growing use of federated Machine Learning, where edge devices collaborate to build a model without revealing all details of each user's data, helping to maintain data privacy. Besides, attestation [83] plays a fundamental role in such a dynamic, distributed scenario. It allows for establishing trust in specific pieces of software, verifying their authenticity and integrity. Through remote attestation, one can ensure to be communicating with a specific, trusted (attested) program remotely. We believe that attestation plays an essential role in building a fully trusted environment for running cloud-edge continuum applications.

3.3 WebAssembly under the hood

This section describes how the cloud-edge continuum can leverage WebAssembly as a unifying technology and the key benefits over the current state-of-the-art solutions.

3.3.1 WebAssembly binary instruction format

WebAssembly (Wasm) [52, 11] is a novel and general-purpose virtual instruction set architecture (ISA). In contrast to previous efforts for platform-independent execution, such as Java from Oracle and Microsoft .NET, Wasm is developed by a consortium of technology companies from the beginning, such as Microsoft, Google and Mozilla, among others. While it was initially designed to increase the performance of Web applications, Wasm does not depend on any Web-related features and is increasingly used for building standalone applications. Wasm has many advantages to be used as a unified execution unit for the cloud-edge continuum. First, Wasm is a compilation target for a wide variety of programming languages, enabling developers to write applications using their favourite programming languages and deploy them across the continuum without adaptation. Second, contrary to Java and .NET, Wasm is compact, has minimal dependencies, and offers additional security benefits, such as sandboxing.

Wasm interacts with the underlying operating system thanks to the WebAssembly System Interface (WASI) [82], a specification standardising a POSIX-like interface. It has been designed with conciseness and portability in mind, enabling the platforms to implement the specifications with ease, ideal for constrained environments, such as IoT and edge devices, as well as TEEs. It currently offers a set of 46 functions, allowing applications to interact with files, networking, and many other operating system functions. Popular compilers for languages such as C and Rust seamlessly translate POSIX calls to WASI calls. Additionally, WASI follows the concept of capability-based security, a security model where each resource (*e.g.*, socket, file) access must be granted by the Wasm runtime, enabling establishing a sandbox. For example, WASI restricts the application to a subtree of the file system by introducing an abstraction layer between the Wasm program and the operating system interface.

Finally, Wasm runtimes can have different memory footprints depending on the execution model, such as interpretation, just-in-time compilation (JIT), or ahead-of-time compilation (AOT). As an example, WAMR [12] is a micro runtime aimed for edge devices that has a file size of 209 KiB when running AOT code and 230 KiB when interpreting, and 41 MiB when executing JIT code. A growing list of toolchains already supports Wasm as a compilation target for different source languages, including C, C++, and Rust. Examples are LLVM [66], an entire compilation infrastructure, and

Emscripten [116], a source-to-source compiler. Support for other programming languages, including C#, Go, Kotlin, Swift, and more, are under active development. For all these reasons, we believe Wasm to be an excellent choice for the binary architecture of the entire cloud-edge continuum.

3.3.2 Trusted execution environments

Trusted execution environments aim to provide safe and trustworthy code execution on (remote) untrusted hardware. Hardware manufacturers have provided TEE implementations more than a decade ago, each one of them offering different features and guarantees. The most influential TEEs that are currently marketed are Intel SGX [39], Arm TrustZone [26], and AMD Secure Encrypted Virtualization (SEV) [41]. These technologies enable processing data in isolated memory areas that cannot be accessed nor tampered with by more privileged software, such as the operating system or the hypervisor. Hence, cloud providers and edge device owners with management rights or even physical control cannot access the data and computation of a tenant, protecting the confidentiality and integrity of their applications.

Cloud providers, such as Microsoft Azure and Google Cloud, already market confidential computing, and we expect widespread adoption of these services due to the demand driven by the cloud-edge continuum [78, 49]. We observe that the rich ecosystem of trusted environments largely varies in terms of security, threat models, and implementation. However, defining a common basis for trusted execution and making it widely available in both cloud and edge environments is essential for the continuum and the industry in general. For that reason, Arm, Intel, Microsoft, and others created the Confidential Computing Consortium (CCC) [1], supporting open-source projects for trusted execution technology under the umbrella of the Linux Foundation. A unified abstraction for TEEs in the cloud-edge continuum must take support and shape from such ongoing efforts. For that reason, the CCC is involved in many projects, such as Enarx [2] and Veracruz [8], which aim to provide Wasm support in TEEs independently from hardware.

In our previous work, we proposed a few solutions to execute general-purpose Wasm applications within TEEs. We developed Twine [84] to bring a Wasm runtime into Intel SGX enclaves, leveraging WASI to interact with the TEE facilities and the untrusted operating system. More recently, we proposed WaTZ [85], a trusted runtime for Arm TrustZone with added remote attestation. The latter, an essential feature for providing trust for remote applications, is surprisingly missing in Arm's architecture. We believe that industrial versions of our prototypes will help pave the way to build distributed applications on the cloud-edge continuum that providers, developers, and users can safely trust.

3.3.3 Live migration of applications

Migration is another need of the cloud-edge continuum that can be helped with the homogeneity offered by . Migration may be needed for a variety of reasons. Some applications have strict latency constraints and may need to migrate to keep close to mobile users. Some applications temporarily require high processing power and may need to migrate to powerful cloud processors. Some applications may also move closer to where data is collected because of legal regulations, because users want to process their private data locally, or simply because it is faster to process it closer to their sources. Migration is much more of a challenge if the underlying environment

is as heterogeneous as the continuum.

Previous work covered the needs and solutions to hand off virtual machines [51], requiring transferring large amounts of data representing memory or even disk images. Transferring Intel SGX enclaves [17] is not only bound to a specific TEE technology but also depends on the application's help to provide its state since the operating system cannot access enclave memory. In contrast, Wasm offers a great environment to migrate running applications, thanks to its linear memory design and sandboxing mechanism. Indeed, Wasm's memory is stored in a contiguous memory segment, where references in code are relative to its starting address. Moving the linear memory to another address or machine does not involve any changes to the references in the application state. Furthermore, the opened resources are tracked by the runtime with WASI, enabling the reproduction of external dependencies. Jeong et al. [58] studied the live migration of applications loaded in browsers by replicating the linear memory. Nonetheless, migrations of Wasm applications with opened dependencies from cloud to edge machines are yet to be demonstrated. The process of migrating executing software depends on the underlying system and its dependencies (*e.g.*, opened files, sockets). We believe future work will enable applications to be seamlessly deployed and moved across the cloud-edge continuum, regardless of the device's processor architecture, TEE technologies, and operating systems.

3.3.4 Technological limitations and pitfalls

While we presented many advantages of Wasm, limitations also exist. One first challenge is simply compiling applications in Wasm. Even though compilers are mature enough to translate source code into Wasm bytecode (*e.g.*, LLVM), the support offered by WASI for surfacing system calls remains limited. Lifting this limitation by extending WASI to match POSIX fully would probably restrict the ability to execute Wasm applications in various environments, such as Web browsers or TEEs. Also, the behavior of WASI diverges from POSIX by adding sandboxing. One alternative is to avoid using WASI, as done by Emscripten, which compiles and defines the import section of Wasm applications with POSIX functions and system calls directly. While this helps run legacy Wasm programs with only a few changes on POSIX systems, it reduces portability for platforms without POSIX, such as other OSes (*e.g.*, Windows) and restricted environments (*e.g.*, TEEs or IoT devices).

Executing Wasm code imposes a performance overhead, as it happens with all intermediary representations. As we demonstrate in Section 3.4, Wasm programs may be up to $3\times$ slower when compared to their native version, depending on the type of workload. This is explained by many factors, such as increased register pressure, additional branch instructions, increased code size, stack overflow checks, and indirect call checks. While some of these issues can be compensated by allowing compilers to spend more time generating better code, other factors are a consequence of the design constraints of Wasm, which would require changes in the Wasm specifications to be solved at the cost of complicating the implementation of the compilers and the way Wasm operates [57].

Another less perceptible limitation is the size of the allocatable volatile memory for Wasm applications. Wasm uses linear memory to store the heap of an executing program. The linear memory is measured in pages, where each page has 65536 B (2^{16}). A linear memory instance can contain up to 65536 pages, for a total of 4 GiB (2^{32}). Be-

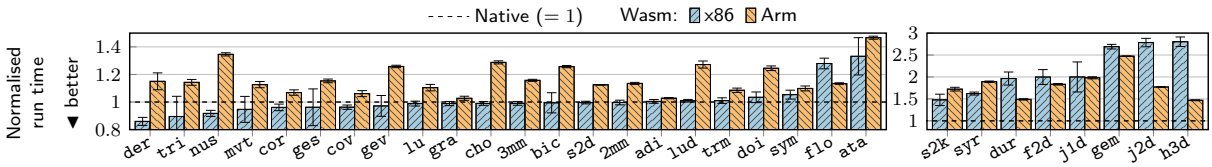


Figure 3.2. Relative performance of `Polybench/C` benchmarks.

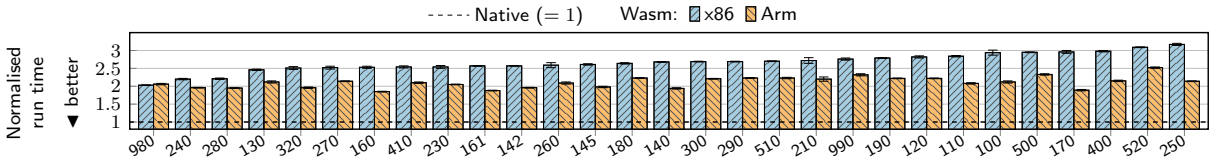


Figure 3.3. Relative performance of SQLite `Speedtest1` benchmarks.

sides, Wasm memory instructions' indices are 32-bit unsigned integers. While most software does not require more than 4 GiB of linear memory, this may restrict some applications of Wasm, such as training sizeable deep learning models or keeping large databases in memory. Fortunately, recent proposals [4] aim to extend this limitation by increasing the number of allocatable pages to 2^{48}, pushing the theoretical memory cap to 16 EiB (2^{64}).

Finally, Wasm is still young and improving through community proposals. Its second major version has been recently released [11], introducing many features, such as reference types, bulk memory, and SIMD instructions. Future contributors may propose Wasm and WASI extensions to relax the limitations or extend the capabilities of the specifications. For example, *wasi-nn* is a proposal to add a WASI module for machine learning to facilitate model inferences [9]. In conclusion, we expect that the current limitations of Wasm will recede with respect to software compilation and deployment for the cloud-edge continuum, thanks to advances in compilation toolchains, extensions of the specifications, and better support of Wasm for all types of devices and environments, including TEEs.

3.4 Performance

This section shows how Wasm scales across the continuum, going from cloud to edge devices using different hardware configurations. While recent research demonstrates the performance of Wasm in general, we are the first to compare two benchmark suites on different processor architectures. Our goal here is to show that Wasm is viable for running code on a variety of devices for similar workloads without suffering from significant performance overheads.

As an example of a cloud server, we use a Supermicro 5019S-M2, equipped with an Intel Xeon E3-1275 v6 (3.8 GHz). We settled for an off-the-shelf NXP MCIMX8M board as an edge device equipped with an Arm Cortex-A53 (1.5 GHz). While TEEs are not demonstrated in this work, these two platforms support trusted execution, namely Intel SGX for the former and Arm TrustZone for the latter. We already illustrated how Wasm could be embedded within SGX and TrustZone in our previous work [84, 85].

We opted for WAMR as a Wasm runtime for its small size and portability across operating systems and constrained environments, such as TEEs. The Wasm benchmarks are compiled into Wasm format using Clang, then compiled again ahead-of-time into

a native format using the compiler provided by WAMR (*i.e.*, `wamrc`). Time is measured using the POSIX function `clock` in all the benchmarks and averaged using the median.

3.4.1 PolyBench/C micro-benchmarks

PolyBench/C [5] is a CPU-bound benchmark suite comprised of various mathematical experiments and commonly used to evaluate the performance of Wasm applications and runtimes [57, 47, 84]. The name of the experiments has been abbreviated in this chapter for conciseness. We assessed 30 PolyBench/C experiments and compared the performance overheads Wasm introduced on x86 and Arm architectures relative to each of their native versions (plain x86-64 and Arm ELF binaries). As such, these measurements compare how Wasm applications perform depending on the deployment target (*i.e.*, cloud or edge machines).

In Figure 3.2, we observe a similar slowdown between the Wasm experiments on both architectures. Indeed, Wasm on x86 and Arm architectures both achieve a slowdown relative to the native of $1.3\times$. We identify and summarise the following groups based on the test performance results: *(1)* the run time of Wasm and native are similar (*e.g.*, `lu`, `gra`, `adi`), *(2)* the run time of Wasm is similar but slower than native (*e.g.*, `s2k`, `f2d`, `j1d`) *(3)* the run time of Wasm is faster than native (*e.g.*, `der`, `tri`, `nus`), and *(4)* the run time of x86 Wasm are significantly slower than Arm Wasm and native (*e.g.*, `j2d`, `h3d`).

Wasm is naturally slower than native code because of the increasing register pressure and code size and the presence of extra branch statements, as discussed in previous work [57]. In some rare cases, Wasm may be faster than native thanks to a reduced number of cache misses, as we observed in our previous work [84]. Finally, some workloads are not well optimised when compiled in Wasm and then recompiled ahead-of-time into native code, as can be observed for the Arm versions on the left-hand side of Figure 3.2.

3.4.2 SQLite macro-benchmarks

SQLite [59] is a widely used full-fledged embeddable database. Thanks to WASI, we showcase the versatility of Wasm by compiling and running SQLite outside of a browser. As such, we diverted the operating system calls made by the database engine to be handled by WAMR. For this purpose, we implemented a shim OS layer as the minimum set of POSIX functions in WASI necessary to support SQLite's in-memory databases. We assessed SQLite performance using its official benchmark suite (`Speedtest1` [7]), running 29 out of the available 32 tests, covering a large spectrum of scenarios (we excluded 3 experiments because of issues with our shim layer). Each `Speedtest1` experiment targets a single aspect of the database, *e.g.*, selection using joins or the update of indexed records.

Figure 3.3 presents our evaluation, where we compare the execution speed of Wasm, using both x86 and Arm architectures, and again normalise against the native run time. Overall, the Wasm slowdown relative to native is $2.7\times$ for x86 and $2.1\times$ for Arm. Most of the experiments (located on the right-hand side of the figure), which are slower, are related to inserting, updating, or deleting data (*i.e.*, 100-120, 180, 190, 230-250, 270-300, 400, 500). The remaining experiments are related to data reading (*i.e.*, 130-145, 160, 161, 260, 320, 410, 510, 520) and housekeeping (*i.e.*, 980, 990). Therefore, we correlate an increasing impact on the performance of write-intensive operations. Finally, when preparing the experiments presented here, we noticed a per-

formance improvement in WAMR's ahead-of-time compiler compared to our results from 2020. Indeed, in our previous work on Twine [84], we measured similar native performance on the same x86 hardware and a considerably worse performance when using WAMR (was $4.1\times$). This strengthens the perspective of using Wasm as a universal, lightweight, yet versatile bytecode to enable platform independence across the continuum.

3.5 Closing remarks concerning the common layer

We envision the cloud-edge continuum as an interoperable, scalable, and distributed system where software may be located to any peer, regardless of the underlying platform. This practice will transform the development life cycle of future applications, enabling developers to focus on the business value instead of dealing with the complexity of each different piece of infrastructure. Wasm is a perfect fit for that task, thanks to its abstraction from the operating system, device type, programming language, and the added security guarantees it can provide using TEEs.

We presented some performance measurements showing that Wasm is a viable alternative to native execution, with acceptable overheads. We covered many aspects of how Wasm can be successfully adopted for the cloud-edge continuum, such as trusted computing, which enforces the applications' confidentiality and integrity, and live migrations, diminishing the latency or increasing the computation power by relocating running software seamlessly.

The remaining challenges concern enhancing the interoperability with the existing programming languages towards Wasm while extending WASI to increase the capabilities of hosted applications. For example, recent initiatives are bringing neural networks [9] and parallelisation [10] into the WASI specifications. Building middleware software that connects the spectrum of the cloud-edge continuum, based on many factors (*e.g.*, latency, computation power) to ease the deployment and migration of Wasm applications, is yet another milestone to reduce the gap between the cloud and the edge worlds. We are confident that Wasm and trusted computing can serve as the foundation for software development for large-scale systems in the years to come.

4 TOCTOU-secure Remote Attestation and Certification for IoT

A key component in securing connected IoT systems is ensuring the integrity of the IoT software-state and detecting any change. This is typically achieved with Remote Attestation (RA), which aims at verifying the state of the software/memory of an untrusted attester (i.e., an IoT device) by allowing a trusted verifier to engage in a challenge-response-based exchange of proof. RA mechanisms rely on hardware/software/hybrid Root-of-Trust. As a result of said attestation, the attester is certified with a certain level of assurance guaranteeing software-state integrity that impacts trust decisions within networked systems. The attestation often results in software updates or issuing certificates indicating device assurance levels. The certificates include information like the assurance evidence, device IDs, assurance level indicating the trustworthiness of the device, etc. These certificates issued for IoT devices differ from conventional certificates due to the resource constraints of the device, dynamic operational environment, diversity in the supply chain, vulnerability management, etc.

In Deliverable D5.2, we briefly introduced the standard IETF RATS in the context of remote attestation schemes suitable for IoT and highlighted the TOCTOU challenge that the schemes face. In this chapter, we introduce and explain **Auto**mated digital **Cert**ification (AutoCert) to provide TOCTOU-security by combining Remote Attestation results about assurance of device health with standard Public Key Infrastructure (PKI) authentication processes.

In the context of RA and certificates that reflect the attested state of the device, the Time-Of-Check to Time-Of-Use (TOCTOU) race condition may take effect. The Time-Of-Check to Time-Of-Use invalidity is a highly contextual problem, existing in remote attestation, operating systems, certifications, etc., and remains possible in this case as well. Due to the dynamic nature of IoT systems, the software-state of the device may have changed in the delta time between the RA and the certificate issuance due to a software update, vulnerability exploitation, or software version update. Although potential solutions exist to prevent and resist TOCTOU attacks in Trusted Platform Module (TPM)-based remote attestation, a solution that provides a mechanism to validate the current software-state against the attested state and use an assurance certificate without invoking RA again, is missing and is critical in the IoT domain. However, a solution that provides a mechanism to validate the current software-state against the attested state in certificates without invoking RA again is also critical in the IoT domain.

4.1 AutoCert - Proposed Mechanism

The mechanism we called AutoCert is an automated procedure comprising of interactions among an **IoT owner**, **IoT devices** as a part of a networked system, a trusted third-party responsible for attesting the device's software-state, e.g., a Conformity Assessment Body (**CAB**), and a standard Certification Authority (**CA**) to enroll device certificates.

4.1.1 Pre-deployment

The manufacturer commissioned the IoT device with software, platform/device certificate, a dedicated TPM 2.0 chip, and a secure unique device identifier during device initialization. The platform certificate binds the TPM to the IoT device. The secure unique device identifier, i.e., *UDevID*, is a hardcoded identity like a device URI, EUI, or DevID playing a role in IoT device identification in local and global networks. The TPM's Root-of-Trust originates with a unique 2048-bit RSA key pair, known as the Endorsement Key (*EK*). The TPM restricts the use of the EK to a limited set of decryption operations as per the TCG rules, and it cannot be used directly for device authentication or digital signatures. Therefore, we generate a 2048-bit RSA key pair, the Attestation Key (AK), using the EK as a seed for attestation. The attestation certificate ($Cert_{AK}$) corresponding to the *AK* is also generated at this state by the IoT manufacturer. The IoT manufacturers and solution providers classify IoT devices into usage profiles based on their deployment scenario, e.g., smart home, automotive, industrial, critical infrastructure, smart grid, etc. In AutoCert, the IoT owner assigns a device_profile to the IoT device to enable security policies for devices within a network. This categorization assists CAs and CABs in conducting reasonable risk assessment and vulnerability management throughout the device lifecycle. The IoT device is configured to boot with trusted software that measures (i.e., calculates the hash) the next software to be run and stores this hash in a Platform Configuration Register using the TPM2_PCR_Extend function. This process continues through the OS kernel code resulting in a chain of measurement. In AutoCert, we propose configuring security-critical software, libraries, files, and executables as a part of this chain of measurements.

4.1.2 Implementation of Secure Boot into the Secure IoT Gateway IoT Bridge component

The Secure IoT Gateway, as described in Deliverable D5.1 [82], secures network connections between hosts by establishing VPN connections between the IoT Bridge and Local / Cluster Gateway, thus encrypting network traffic sent between hosts. IoT Bridges provide the VPN entrypoint for hosts by directly connecting the host via ethernet or Wi-Fi to an IoT Bridge. These Bridges are later deployed within a close range of the to-be-secured hosts, which raises the necessity to further secure the system beyond VPN encryption. It is possible that IoT systems or hosts are physically exposed or open to public access, like sensors or network cameras in buildings. Because the IoT Bridge is designed to be kept within range of the secured host, it has to be hardened towards physical access attacks.

Until now, the IoT Bridges are using SD-Cards or on-board eMMC as their boot drives, whose contents are unencrypted and not authenticated upon boot. By accessing the unencrypted filesystems on the boot drives, an attacker could run their own software on the IoT Bridge, thus bypassing the VPN tunnel or re-routing the traffic to another server. The private and public keys, which are used for authentication on the Network Cockpit backend, are also stored in the filesystem. They could be used by an attacker to imitate the device at the Network Cockpit side. It would also be possible to gain partial access to the VPN network created by the Secure IoT Gateway. This could be done by extracting the shared-key and the OpenVPN settings and parameters used for VPN encryption. So called "Zones" restrict the routing between VPN tunnels. All devices placed in the same Zone could be accessed by the hijacked VPN tunnel.

In order to circumvent the listed attack vectors, we are planning to integrate a Root-of-Trust inside the IoT Bridge. All known attack vectors depend on the unencrypted file system and a lack of boot ROM verification. By establishing a Root-of-Trust inside the SoC of the IoT Bridge, these attacks can be circumvented. We are focussing to integrate Secure Boot on the IoT Bridge, because the used ARM SoCs are equipped with eFuses to enable bootloader verification by the boot ROM.

4.1.3 Remote Attestation

AutoCert's remote attestation is built on the Challenge/Response Interaction Model from the RATS architecture. Before a device is attested (Fig. 4.1), the IoT owner is responsible for generating the reference values corresponding to the device software/s and securely transferring them to the verifier. We assume a confidential exchange of these values. Before the remote attestation begins, the IoT owner sends a signed request to the CAB with the *UDevID* and device_profile of the IoT device. The CAB sends a signed attestation request containing a random nonce N and a *PCRSelection* is sent to the IoT device. The TPM2_Quote function is used to generate the evidence. The cryptographically strong random nonce N uniquely distinguishes the evidence, determines its freshness, and prevents replay attacks. We propose the generation of an integrity key pair, IK, by the IoT device and sending it along with the evidence for the creation of an integrity_proof. The IK is an RSA key-pair, IK_{priv} and IK_{pub} generated with the TPM2_Create function using the *PCRSelection*. Since any change in the security-critical software on the device is recorded with an update to the PCR using TPM2_PCR_Extend, the use of this *PCRSelection* in creating the IK ensures that this key will not be valid if the software-state of the device changes.

A valid TPM-generated attestation key, the *AK*, is used to sign TPM-generated evidence. It serves as a way for third parties to validate keys and data generated by a specific TPM on an IoT device. On receiving the evidence, the CAB validates the accompanying signature and compares the evidence against reference values. Following the attestation result and using a suitable risk assessment mechanism (not discussed in this work), the attester's Assurance Level is calculated against the device_profile. The results of attestation and the Assurance Level are used by the CAB to ensure the software-state integrity.

TOCTOU and integrity_proof

The integrity key pair, IK, is proposed to address the TOCTOU race condition. The *PCRSelection* contains the measurements computed and stored during measured boot, representing the IoT device's software-state.

Using these PCRs in RA and generating the IK_{priv} and IK_{pub} key pair creates a dependence of the IK on the software-state of the device. As soon as the software-state changes due to a new vulnerability or malicious update, the IK is invalidated.

This forms the core of AutoCert procedures and is a part of the proof of the IoT device's software-state integrity, as it strictly locks the IK to a valid state of the device. We compute an integrity_proof by aggregating the value of *PCRSelection* used in evidence generation, i.e., *PCRIntegrity* and the IK_{pub}. The integrity_proof , Assurance Level, and *UDevID* are then shared with a trusted CA. The CA now possesses records of attested IoT devices against their *UDevID* and the assurance attributes. These attributes are integrated with the IoT profile of the standard X.509 certificate using

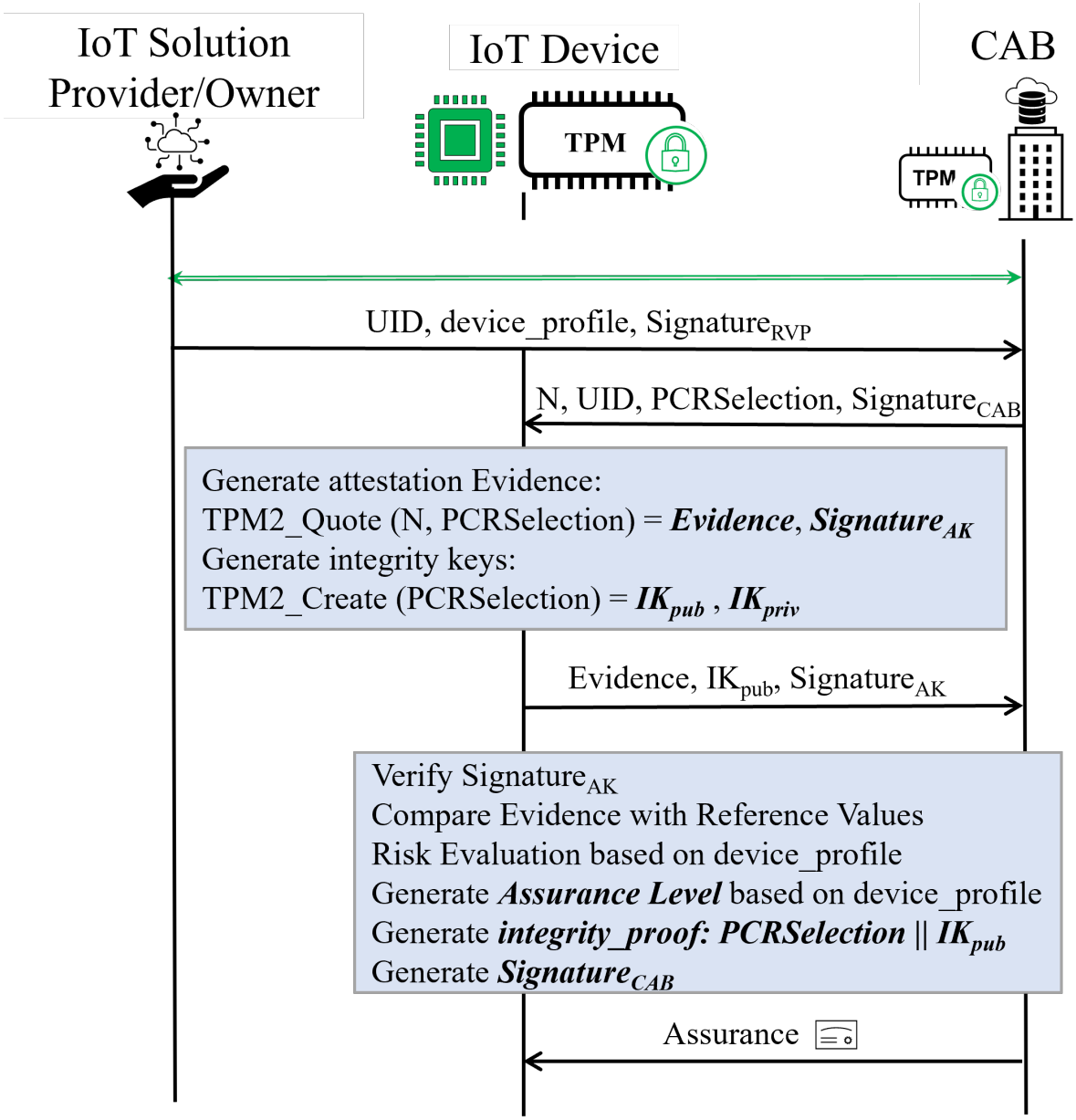


Figure 4.1. Remote attestation procedure

custom extensions. This certificate $Cert_{AC}$ reflects a CA-verified device identity (authentication) as well the CAB-attested software-state of the IoT device (assurance).

4.1.4 Verification for TOCTOU-security

The verification of this integrity_proof for TOCTOU-security applies to all IoT devices using X.509 certificates for authentication and establishing secure DTLS communication sessions with clients.

To achieve assurance of the IoT device's software-state, the client performs two levels of integrity checks, as presented in the Figure 4.2.

The first level of integrity check includes verifying the Assurance Level stated in the $Cert_{AC}$. This Assurance Level would form the basis of network access policies or authorization to access system resources.

However, as stated earlier, it is possible that the IoT device's software-state changes

after the remote attestation process, or $Cert_{AC}$ enrollment. This can happen due to malware or vulnerabilities in existing software. This scenario presents itself as an instance of a TOCTOU attack, and checking the Assurance Level is insufficient in security-critical cases.

To eliminate this TOCTOU condition, AutoCert facilitates another level of integrity verification. To perform this Level 2 integrity check, AutoCert introduces a lightweight service to ensure that the integrity_proof is valid. The verification process includes sending a random challenge by the client to the IoT device after signing it using the IK_{pub} from integrity_proof in the $Cert_{AC}$. Since the integrity_proof is locked to the state of the IoT device attested by CAB, it can only be decrypted by the IoT device if it possesses IK_{priv}, hence guaranteeing proof of possession. The IoT device decrypts the challenge, includes the current value of the *PCRIntegrity*, and signs it. The challenge ensures the freshness of this message exchange. The current value of the PCR concatenated with the challenge is received by the client, which verifies it against the PCR values from the integrity_proof , i.e., the *PCRIntegrity* confirming that no changes have occurred concerning the software-state since attestation.

4.2 Implementation and Experimental Evaluation

As a Proof-of-Concept (PoC), we implemented the AutoCert setup with an attestation service on the IoT device, which is invoked when it receives an attest request. We also implemented an integrity verification service corresponding to the two levels of integrity checks. The experiments are performed using the OPTIGA TPM Evaluation Kit comprising of a Quad Core 1.2 GHz, 64-bit Raspberry Pi 3 with 1 GB RAM and an Iridium board with OPTIGA SLM 9670 TPM 2.0. We choose TPM SLM 9670 for this evaluation since it is specially designed for use automotive/industrial applications. The following set of experiments aims to measure the system-wide execution time of the proposed mechanism during different phases. We measured the Round Trip Time (RTT) as the time elapsed from the start of each AutoCert phase until the completion of the phase. We measured the phases using a system clock in nanoseconds and iterated the experiments 5-10 times to ensure statistical accuracy.

Phase 1 of AutoCert begins with a request to the CAB to initiate AutoCert remote attestation with the IoT device. The RTT of this phase is 28800 ms. This phase is expected to execute during device assembly after the unique device keys are integrated into the hardware, and device software is installed. This does not interrupt runtime services like mutual authentication, where excessive delays disrupt services, timeout, or cancellation of operations.

Phase 2 of AutoCert is the certificate enrollment. On receiving a certificate enrollment request from the IoT device, the CA checks for assurance attributes received from the CAB, associated with the *UDevID* of the IoT device, and enrolls the certificate, including the assurance attributes. The enrollment of $Cert_{AC}$ with assurance attributes takes 7104 ms. This measurement merely gives an estimate of the generation of a certificate with additional extensions. In actual events, certificate issuance and enrollment time also vary depending on the computational capabilities of the CA and network capacity.

Phase 3 of AutoCert provides 2 levels of assurance to the communicating devices. The proposed Level 1 integrity check attains a basic level of assurance. This begins by ver-

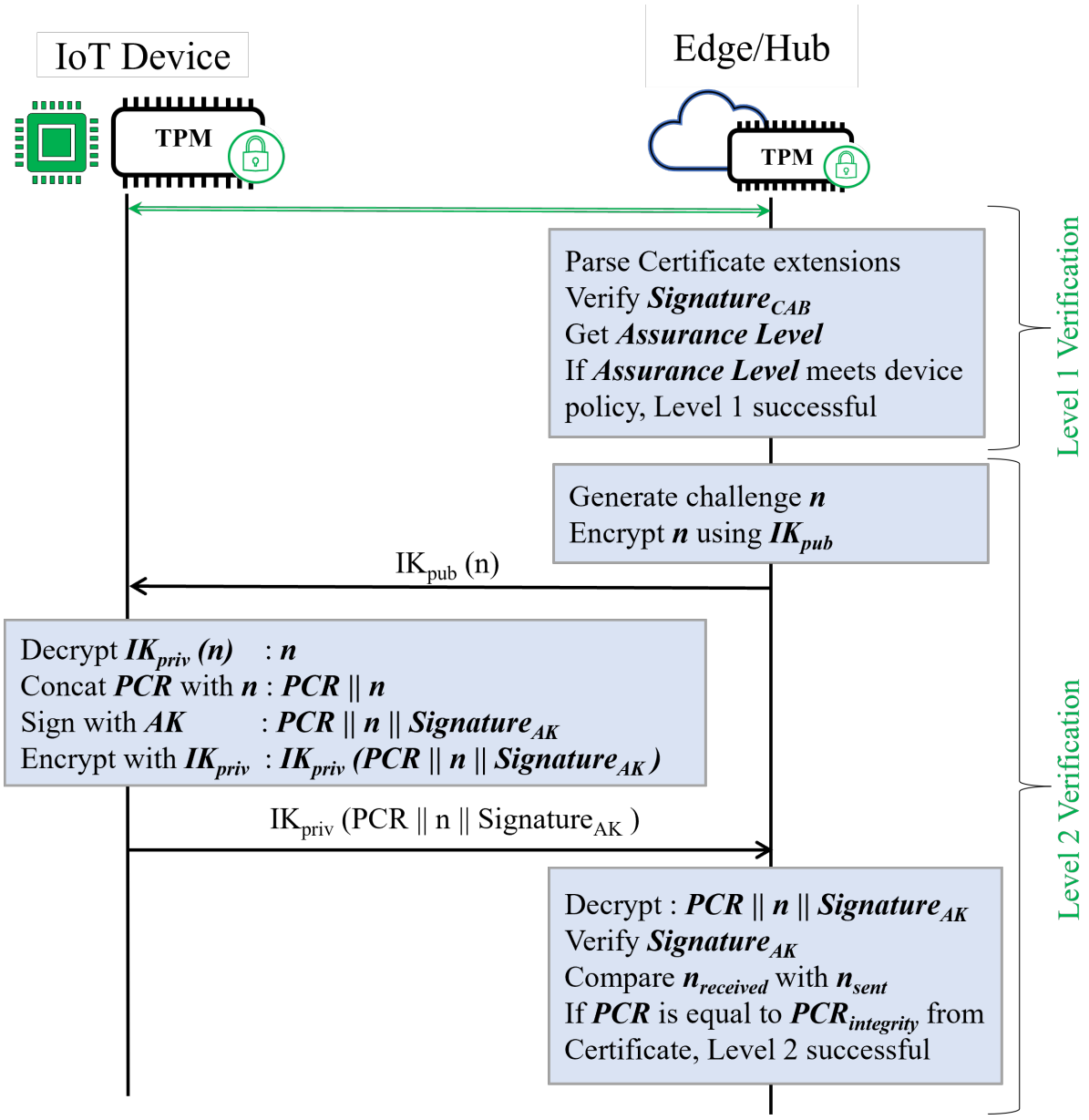


Figure 4.2. Verification procedure

ifying the signature and the Assurance Level from the $Cert_{AC}$. An extended TOCTOU-security of assurance is provided in Level 2. The Level 1 verification steps are executed in 0.7 ms, and the RTT for Level 2 verification, including minor network delays between the two involved entities, is 4746 ms. The majority of the execution time during Level 2 verification can be traced to the creation and loading of the encryption key. As these operations depend on the implementation of TPM specifications and adjacent function libraries, it is reasonable to state here that the RTT for Level 2 verification is justified considering the hardware security guarantees provided by the TPM.

4.3 AutoCert - Conclusion

This chapter presented AutoCert, addressing TOCTOU-security in Integrity Certificates corresponding to software-state assurance in IoT devices and providing a standardized mechanism to distribute Integrity Certificates. AutoCert's remote attesta-

tion is based on IETF RATS relying on TPM2.0 for evidence generation. We have proposed the integration of the AutoCert mechanisms into existing standards to facilitate its adoption in the emerging PKI for IoT. The complete set of results and discussions are published in a scientific journal [63].

4.4 Future Work

In the current approach, the remote attestation is built on the challenge/response Interaction Model from the RATS architecture, which is not primarily developed for low-power IoT devices. In future, we plan to extend this work and develop a highly optimized remote attestation and certification mechanism that fits low-end IoT devices. The planned work will also encompass lightweight attributes of the certificate taking advantage of our recent work on concise X.509 certificate for IoT [74], the optimization of certification process, and the certificate usage. In the current work, the risk evaluation of individual software was considered out-of-scope. A potential future direction for this research is the study of vulnerability assessment and risk evaluation of individual software, and its integration with our TruCert solution, which can help certify IoT devices with different assurance levels.

In the future, we also plan to focus on dissemination of our trusted execution, remote attestation, and automated certification of IoT work in relevant venues. We plan to push this work as a potential solution for the implementation of EU Cybersecurity Act for IoT. Towards this end, we have submitted a presentation request in a highly selective conference on EU Cybersecurity Act aimed at industry and public organizations: The International Conference on the EU Cybersecurity Act 2023. Our proposal [15] is already accepted, and we will prepare and present this work in this conference. RISE is also a member of Stakeholder Cybersecurity Certification Group (SCCG), the highest-level external group helping ENISA and EU on the implementation of EU Cybersecurity Act; we plan to disseminate our work via SCCG.

5 Trusted Verifier Service

In this chapter we will describe further the implementation of the Tru**s**ted Veri**f**ie**r** S**e**rvice (SIRE), a replicated infrastructure that supports multiple functionalities necessary to an IoT System. In Deliverable 5.1 [86], we outlined the main features of SIRE on a conceptual level, while in Deliverable 5.2 we gave further insight into the system's architecture and a few implementation details. In this deliverable, we will give further insight into the implementation of all of the SIRE's features, which is currently in a fully functional state. We will also provide a performance evaluation of SIRE and an analysis of its results.

5.1 Recalling SIRE

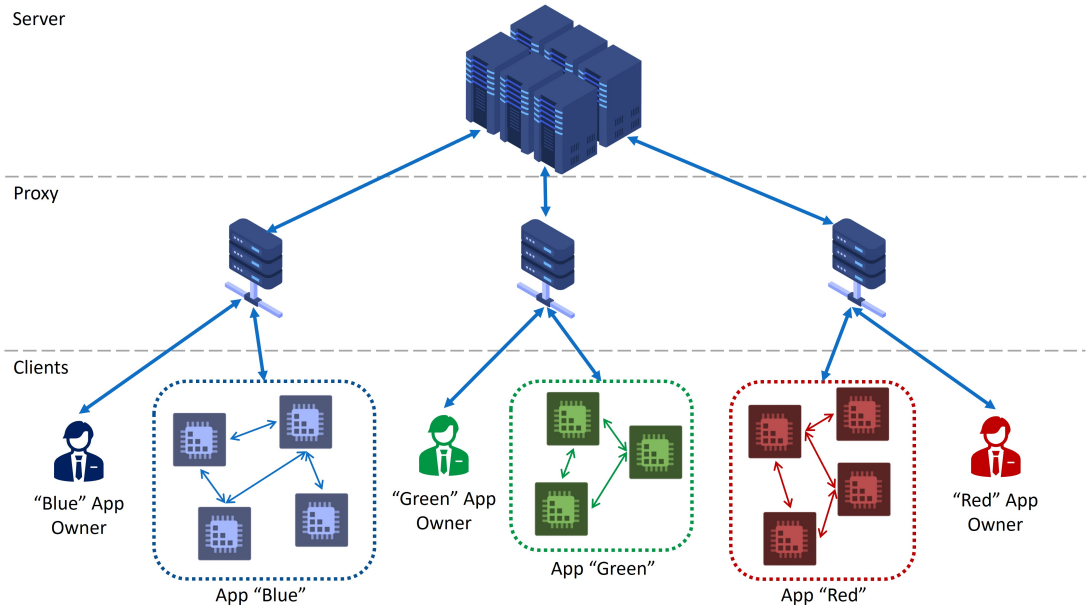


Figure 5.1. Overview of SIRE.

In this section, we will give a brief summary of the design of SIRE described in the previous deliverable.

SIRE is an infrastructure that supports remote attestation, application membership management, auditable integrity-protected log, and coordination primitives. It considers that each device belongs to one application, with the possibility of having multiple applications running simultaneously. Each of these applications can have different requirements, specifications, and contexts (e.g., the same SIRE deployment can manage a set of devices that belong to a smart home application while having another set that belongs to a smart industries application). A good example of this kind of application would be the Automotive AI Use Case, where there are many migrating components that would need to be accounted for and certified to ensure proper and secure operation of the critical car subsystem.

The SIRE system is composed of three main entities, the SIRE server, the proxy, and the devices, as illustrated in Figure 5.1.

5.2 Implementation Architecture

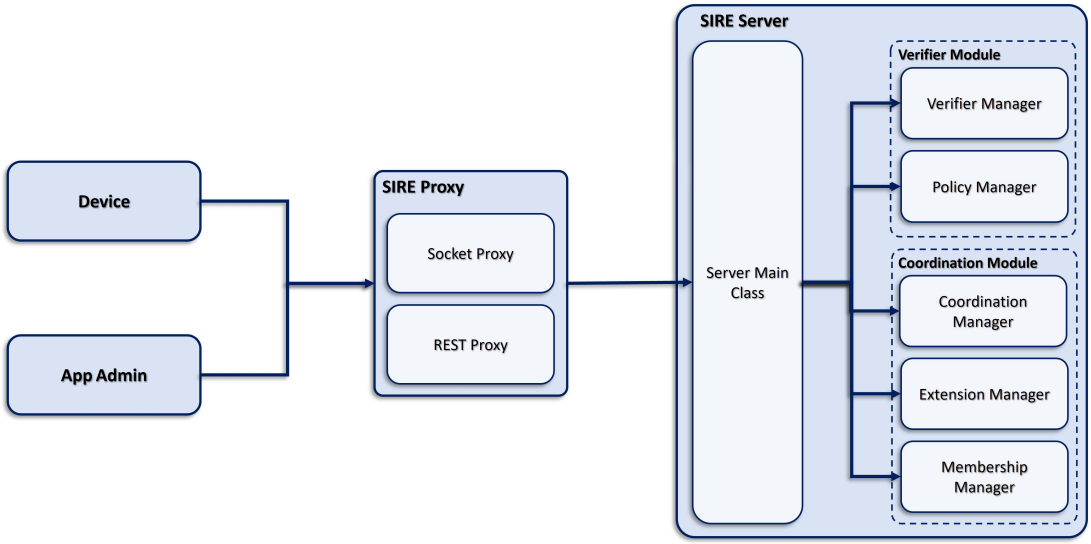


Figure 5.2. SIRE's implementation architecture.

SIRE is implemented in Java over the BFT-SMaRt library, and its implementation architecture is described in Figure 5.2. Since SIRE is a system with multiple functionalities, we divided its implementation into multiple modules, each responsible for a different feature. These modules do not function independently and will interact with one another whenever needed. For example, the Verifier Manager will call the Policy Manager, which in turn can call the Extension Manager to execute the policy script. SIRE's implementation is structured as follows:

- **SIRE Server:**
  - **Server Main Class** – Handles requests and relegates them to the appropriate managers. Contains the attestation protocol.
  - **Verification Manager** – Verifies the attester's evidence.
  - **Coordination Manager** – Manages the key-value store of the coordination kernel.
  - **Membership Manager** – Holds the state of the membership of each application; responsible for pruning invalid and timed-out devices as needed.
  - **Extension Manager** – Compiles, stores and executes Groovy [25] extensions of each application.
  - **Policy Manager** – Compiles, stores, and evaluates the policies of each application.
- **Proxy:**
  - **Socket Proxy** – Accepts communications through sockets with Protobuf [50] serialization.
  - **REST Proxy** – Accepts communications through REST requests with JSON [3] serialization mapped with Spring [106]; holds a web interface.

In the following sections, we will detail the implementation of these modules and their interactions.

5.3 SIRE Server Implementation

As mentioned before, SIRE's server is replicated on top of BFT-SMaRt [36], which grants it Byzantine fault-tolerance. Since SIRE possesses many features, it is important, for the sake of clarity and good programming practices, to separate them into multiple modules, with each being responsible for one of the functionalities.

Server Main Class

All requests sent to the server are handled by its Main Class which relays them to the proper classes. This class also contains the attestation protocol that will be further detailed and explained in the following section.

5.3.1 Verifier Module

In the case of SIRE, devices will act as attesters and the server as the verifier. The main class of the server is responsible for executing the multiple steps that compose the attestation protocol, relaying the verification and policy process to their respective entities. The process of verifying these properties and their correctness is performed by the Verification Manager. This process is usually done using a set of rules in a policy which is used to evaluate the evidence and, therefore, the attester's correctness. These policies will be created, stored, and evaluated in the Policy Manager, as we will see later in this chapter. Using both the Verification and Policy Manager, SIRE can evaluate a device's correctness and whether it can join the system.

Attestation Protocol

The main class of the server contains the entirety of the attestation protocol except for the verification of the evidence and policy evaluation.

We will be using signature algorithms to ensure that the messages of the remote attestation protocol have not been tampered with. Since SIRE is replicated, an adequate signing algorithm is needed, and as such, SIRE uses Schnorr signatures [92, 93] with Lagrange Interpolation. This signature algorithm uses the properties of elliptic curves to compute and verify signatures. Schnorr uses random nonces in its implementation, which have a high performance cost when generated in runtime. To avoid this performance issue, the nonces are pre-generated and stored in text files, which the system will access during startup and load a random batch of those pre-generated nonces. During runtime, each time a nonce is needed to sign data, one will be picked at random from the batch loaded during system startup. The elliptic curve used for all operations involving Schnorr signatures was the secp256r1 curve [6].

Every attester is assumed to have an asymmetric key pair and the verifier's public key, while on the verifier's side, it is expected to have an asymmetric key pair and the policy of each existent application. These keys can be hardcoded or obtained from a Public Key Infrastructure (PKI). As described in Figure 5.3, SIRE's attestation protocol is structured in the following way:

- **Message 0 (attester→verifier):** The attester signs its public key using Schnorr and sends it to the verifier.

- **Message 1 (attester←verifier):** When the verifier receives Message 0, it verifies the signature, and if it is correct, it will generate a timestamp to be returned to the attester. This timestamp is generated through BFT-SMaRt, which possesses leader-generated timestamps, making it so that every replica will have the exact same one. After generating it, it will be stored associated with this device's id and used when processing Message 2. This timestamp will be signed using Schnorr together with the attester's public key, using the verifier's key, and sent to the attester.

- **Message 2 (attester→verifier):** The attester verifies the signature of Message 1's content (attester's public key and timestamp), and if it is valid, it will generate the evidence of its hardware and software properties. The format and contents of this evidence are specific to each application. After the evidence is generated, Message 2 will be prepared, composed of the application's id, the timestamp that the verifier sent in Message 1, the evidence, and the attester's public key. All of this will be signed and sent to the verifier.

- **Message 3 (attester←verifier):** After receiving Message 2, the verifier checks the signature to be sure the message has not been tampered with. If it is valid, it will first check if the timestamp sent by the attester is the same as it has stored and if it is within the established time-bound specified by the application administrator. This verification ensures that the evidence is only valid for a short time. If this verification is successful, the verifier will then proceed to evaluate the evidence using the policy from the corresponding application. The proper policy is obtained using the application's id sent in Message 2. If the evidence is validated, the attester is now assumed as correct and will be added to the application's membership which can trigger the execution of an extension in the Extension Manager. After adding the attester to the membership, the verifier will send the attester a message composed of a timestamp of the time of its attestation, the attester's public key, and a hash of Message 2, which will all be signed. If the evidence is invalid, an empty message will be sent to signal the attester that it has not been correctly attested. When the device receives the third message, it will check the hash's signature and validity as an assurance of it being attested.

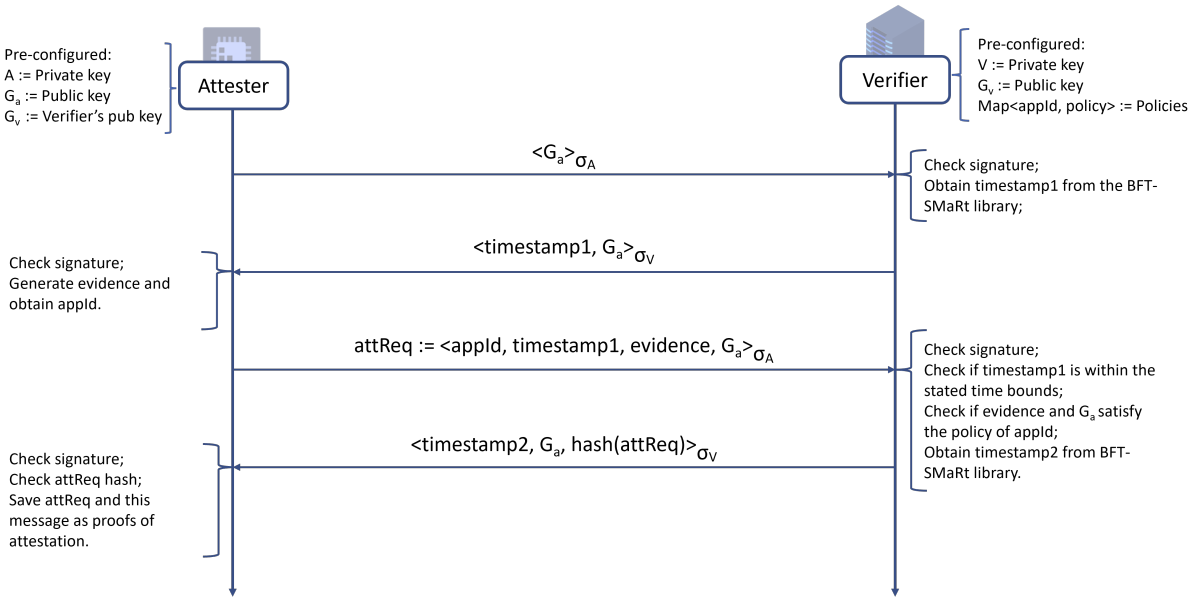


Figure 5.3. SIRE's attestation protocol.

Timestamps are used to limit the interval of time in which the evidence is valid. By assuring the freshness of the evidence, we can limit the possibility of attackers pretending to be a correct device by replaying messages to successfully perform attestation and be added to the membership. This way, we can prevent devices that are not in the system from being assumed as such. For example, if an attacker saves Message 0 and Message 2 sent by device A, that, since then, has left the system, it can execute an attestation on behalf of device A, adding it to the membership of its application. By allowing this, we would undermine the membership management provided by SIRE. However, we should note that our scheme only works if clocks are loosely synchronized and under periods of synchrony.

Verification Manager

The Verification Manager is responsible for performing the verification process of the device's evidence. The evidence's composition is specific to each application, but an example of an evidence format, which is the one used in Intel SGX RA [55] and WaTZ [85], is: anchor, claim, attestation protocol version and public session key. We used this evidence format for evaluation and testing purposes, but SIRE can support any evidence format. The evaluation of the evidence is done using a set of rules defined in the policy, which is stored and evaluated by the Policy Manager. Each application has a corresponding policy which is expected to be implemented by the application administrators.

Policy Manager

The Policy Manager has the responsibility of storing and evaluating policies. These policies are to be implemented by the application administrators and will be specifically used with devices belonging to that application. These policies can be implemented in two different ways:

- **Logical Expressions**, which are parsed and evaluated with the given values. Simpler to implement for users but harder to support for the attestation ser-

vice. An example of this is Microsoft Azure attestation's policies [30].

- **Scripts**, which are populated and executed with the given values. Harder to implement for users but easier to support for the attestation service. An example of this is Intel SGX Remote Attestation's policies [55].

The interface that application administrators can use to manage their application's policy is described in Table 5.1 and can be accessed through the proxy, as we will see in Section 5.4. Each application has only a single policy. Currently, SIRE supports only policies in script format, executed in the Extension Manager of the coordination kernel, simply associated with the proper extension key. This will be further detailed in Section 5.3.2.

Function	Description
setPolicy(appId, policy)	Sets policy from app with Application ID *appId*
deletePolicy(appId)	Deletes policy from app with Application ID *appId*
getPolicy(appId)	Gets policy from app with Application ID *appId*

Table 5.1. Policy Manager Interface

5.3.2 Coordinator Module

The Coordinator Module is responsible for storing a coordinated Key-Value (KV) store with a ZooKeeper-like [53] interface, managing the applications' membership and their extensions. The coordination property is achieved through the ordered execution of operations granted by the BFT-SMaRt library [36]. This module is divided into three classes: (1) Coordination Manager, which manages the KV store and calls the Extension Manager when needed; (2) Extension Manager, which compiles, stores, and executes the application's extensions, including the script policies from the Verifier Module; and (3) Membership Manager, which stores, manages and maintains the membership of each application, including checking if a device is still valid and pruning invalid devices.

Coordination Manager

The Coordination Manager maintains the KV store, which can be accessed by both devices and application administrators, to store any data they might need to coordinate. The key-value store has a similar interface to ZooKeeper [53], as can be seen in Table 5.2, distinguishing itself by having generic values, as opposed to ZooKeeper, which only stores strings.

The keys are represented in Strings as they can be compared and are more intelligible compared to more generic types, such as byte arrays. The generic values allow users to store any data they might need without any restrictions other than having to be serializable into byte arrays. This was necessary due to the serialization method SIRE uses, Protobuf [50], which does not support generic types, requiring the object to be deserialized before being stored, and serialized after each access, adding unnecessary computations. The values are stored as they are sent, meaning that SIRE will not perform any additional encryption, leaving that responsibility to clients. Whenever an operation is called on the KV store, the Coordination Manager will call the Extension Manager to execute the corresponding extension if existent. Besides being used for the attestation protocol, timestamps can also be used for coordination

between devices. These timestamps are also stored on the KV store to be accessible by another device.

Function	Description
put(appId, key, value)	Adds a new entry to storage
delete(appId, key)	Deletes an entry from storage
getData(appId, key)	Gets data associated with key from storage
getList(appId)	Gets list of all entries
cas(appId, key, oldValue, newValue)	If the value of a key is equal to oldValue, replace it with newValue.

Table 5.2. Coordination Manager KV store interface.

Extension Manager

The Extension Manager is responsible for compiling, storing, and executing extensions. Extensions are small pieces of custom code that will be executed when certain criteria are met, on the server side, making the coordination kernel more versatile and efficient without sacrificing its simplicity [42].

The extensions should be implemented in the form of Groovy [25] classes by the application administrators. Since they are assumed to be trusted clients of the system, no measures are taken to assure the security of their code. We chose Groovy as the programming language for the extensions because it is already included in JDK and can be directly run on the JVM.

The extension is sent to SIRE by the application administrator in the form of a string of text, which in turn is compiled by the extension manager present on the server and instantiated. These will be run whenever an operation from the membership (*join, leave, ping* and *getView*) or KV store (*put, get, getList, delete* and *cas*) interface is called. The interface that application administrators can use to manage their application's extensions is described in Table 5.3. This interface can be accessed through the proxy, as seen in Section 5.4. The extension is required to have a method called $executeExtension$, which takes an $ExtParams$ object as an argument. This $ExtParams$ is just an encapsulator for the parameters that the operations might receive (key, newValue, and oldValue).

When the extension is first introduced in the system, it should be associated with a key in the form of $appId$ + $extType$ + key, with the possibility of $extType$ and key being null. The $extType$ parameter corresponds to the type of operation, and the key parameter is the key associated with the operation that will trigger the execution (e.g., an extension associated with application $app1$, operation type $extPut$ and key $exampleKey$ will be executed whenever $put(app1, exampleKey, exampleValue)$ is called). Since some parameters can be null, the extension manager will first check for extensions matching $appId$ + $extType$ + key, then for any matching $appId$ + $extType$, and at last for $appId$ only.

In the case of the script policies mentioned in Section 5.3.1, these will be treated as a normal extension associated with a key in the format $appId$ + $ext\_attestation$. The result of the execution of the extension is returned to the Policy Manager and treated accordingly. This script policy is expected to have a method called $verifyEvidence$,

which takes an *Evidence* object as an argument and returns a boolean. In the following example, the script policy defines the reference values for the hardware and software claims which it compares with the one contained in the evidence. It also defines the endorsed keys, which are the public keys it expects devices to have. At last, it defines which version of the device's attestation protocol should be running.

```
package sire.attestation

def verifyEvidence(Evidence e) {
    def refValues = ["measure1".bytes, "measure2".bytes]
    def isClaimValid = false
    for(value in refValues) {
        if(value == e.getClaim())
            isClaimValid = true
    }
    def keys = [[3, -27, -103, 52, -58, -46, 91, -103, -14,
                 0, 65, 73, -91, 31, -42, -97, 77, 19, -55,
                 8, 125, -9, -82, -117, -70, 102,-110, 88,
                 -121, -76, -88, 44, -75] as byte[]]
    def isKeyValid = false
    for(key in keys) {
        if(key == e.getPubKey())
            isKeyValid = true
    }
    def expectedVersion = "1.0"
    return isClaimValid && isKeyValid
        && expectedVersion.equals(e.getVersion())
}
```

Function	Description
addExtension(appId, type, key, code)	Adds extension for the app with Application ID *appId* to be executed when an operation of a given type is called with the given key. The given type and key can be null. The code will be stored associated with extensionKey (appId + type + key, in this order).
removeExtension(appId, type, key)	Removes extension associated with extensionKey (appId + type + key, in this order). The given type/key can be null.
getExtension(appId, type, key)	Gets the code of the extension associated with extensionKey (appId + type + key). The given type/key can be null.

Table 5.3. Extension manager interface.

Membership Manager

The Membership Manager is responsible for storing and maintaining the member state of each application. Each application membership will store the id of the member devices and the device's state. The device's state is composed of two main fields: the device's last ping and the device's attestation certificate. The Membership Manager's interface is described in Table 5.4.

The devices' last ping is used to detect possibly crashed devices, preventing them from wasting system resources. Since SIRE is distributed and each replica holds the membership state, it will use the leader-generated timestamp from BFT-SMaRt to update the device's last ping to avoid possible inconsistencies. To prevent the devices from sending unnecessary pings and wasting resources, each interaction it has with the system will update its last ping. If a device does not interact or ping the system for a certain amount of time, it is assumed as crashed and is removed from the application's membership, preventing it from communicating with SIRE before being attested again.

The attestation certificate is provided by the Verification Manager when the attestation protocol is executed successfully. When a device calls the join operation to become a member of an application, it will also trigger the attestation protocol, with it only being added to the membership if it is attested successfully. When an attestation certificate expires, the device has to attest again to be able to communicate with SIRE.

To avoid SIRE from constantly checking the membership for invalid devices, these will only be pruned when they are accessed, for example, when a View operation is called or when the invalid device attempts to interact with the system. Since a device's membership state might not be accessed for long periods, SIRE still employs a garbage collecting mechanism by executing a periodic prune to the entire membership, preventing timed-out devices from wasting resources.

Function	Description
join(appId, deviceId)	Finish the attestation protocol and join the application's membership
leave(appId, deviceId)	Leave the system
ping(appId, deviceId)	Assures the system that the device is still running
getView(appId)	Get current membership of system from the app with Application ID *appId*

Table 5.4. Membership interface

5.4 Proxy Implementation

As described in the previous deliverable, the proxy is used to relay messages between the server and devices as well as between the server and the application administrators. The proxy is divided into two entities, the Socket Proxy and the REST Proxy, which mostly differ in the method of communication they use: sockets using Protobuf [50] and REST requests using JSON [3], respectively. Even though the REST Proxy is more oriented to application administrators since it has a user-friendly web interface for their use, it also accepts requests from devices.

5.4.1 Socket Proxy

The Socket Proxy receives the requests through sockets using Protobuf [50]. Each time a client connects to SIRE, the Socket Proxy creates a socket and thread for it. This thread remains active until it leaves the system.

Each time the Socket Proxy receives a request, it is relayed to the server replicas using the BFT-SMaRt's client (called Service Proxy). If the request has a return value, the

Socket Proxy encapsulates it in a Protobuf message before returning it to the client. These message formats are used both in communications between Clients and Proxy and between Proxy and server.

The Protobuf messages have the following format:

- **ProxyMessage**
  - *operation* – The type of the operation. Can be any of the ones described in the APIs defined in the modules in Section 5.3.
  - *evidence* – Evidence for the attestation protocol.
  - *timestamp* – Timestamp for the attestation protocol.
  - *pubKey* – Device's public key for the attestation protocol.
  - *signature* – Signature for the messages of the attestation protocol.
  - *key* – For operations on the KV store interface of the Coordinator Module.
  - *value* – For operations on the KV store interface of the Coordinator Module, as described in Section 5.3.2. Also acts as the *newValue* for the *cas* operation.
  - *oldValue* – Value to be compared with the current one in the KV store for the *cas* operation.
  - *deviceId* – Identifier of the device that sent the request.
  - *appId* – Identifier of the application of which the sender of the request is part.
  - *code* – Code of the extension for the Extension Manager, as described in Section 5.3.2, as well as the policy for the Policy Manager, as described in Section 5.3.1.
- **ProxyResponse**
  - *responseType* – The operation this message is a response to. Can be one of the following: $mapGet$, $mapList$, $view$, $extensionGet$, and $policyGet$.
  - *list* – The list of all the entries of an application's KV store of the Coordination Manager, as described in Section 5.3.2, in response to a $getList$ request.
  - *value* – Value returned from a $mapGet$ request to an application's KV store of the Coordination Manager, as described in Section 5.3.2.
  - *members* – List of all the members of an application's membership from the Membership Manager, as described in Section 5.3.2. Contains each of the device's id, last ping, certificate and expiration time.
  - *extPolicy* – Code of extension returned from an extGet request to the Extension Manager, as described in Section 5.3.2, as well as the definition of the policy from the Policy Manager, as described in Section 5.3.1.
  - *sign* – Signature for the messages of the attestation protocol.
  - *timestamp* – Timestamp for the attestation protocol and coordination between devices.

Each time a new connection is established, i.e., a new device interacts with the system, the Socket Proxy creates a new thread that is dedicated entirely to this device's requests. After a device leaves the system (or is removed due to being invalid), the thread is closed.

5.4.2 REST Proxy

The REST Proxy receives requests through a REST interface with contents in JSON [3]. These requests are encapsulated into a $ProxyMessage$, as described in the previous section, and relayed to the server replicas using the BFT-SMaRt's Service Proxy, which processes in the same way it does with requests coming from the Socket Proxy. Encapsulating the result of the request into a $ProxyResponse$ is also not needed as it is simply sent in the response body in JSON format. In the case of application administrators, they need to be authenticated to have permission to access the REST interface.

The requests are mapped to URLs using Spring [106] in the following way:

- https://sireurl.com/resource/secondary_resource/_accessed?parameter=value

The resource on the URL can be one of the following:

- $extension$ to access the Extension Manager. POST request to add an extension, DELETE request to remove an extension, GET request to fetch an extension.
- $policy$ to access the Policy Manager. POST request to set the policy, DELETE request to remove the policy, GET request to fetch the policy.
- $membership$ to access the Membership Manager as well as the Verifier Module in the case of a $join$. POST request to join the system or update the timestamp, GET request to fetch the membership, DELETE request to leave the system.
- map to access the KV store from the Coordinator Module. POST request with just one value in the body for put, two values for cas; GET request with an application ID and key in the body for $getData$, just the application ID for $getList$; DELETE request for $delete$.

The parameters are mostly IDs that specify the application (in the case of application administrator operations) or the device (in the case of device operations). The other parameters are sent in the body of the request (extension's code, map values, etc.), for example, if an application administrator wants to add an extension to its application called "App1", it sends a POST request to the URL "https:/sireurl.com/extension?key=App1" with the code of the extension in the body of the request.

Web Interface

The web interface is simply a more user-friendly way of accessing the REST interface and as such, is only to be used by application administrators.

First, the application administrators are presented with a login page to authenticate themselves with a username and password. Then, a list of their applications is shown, from which they can pick one to configure. After choosing an application, they are

presented with a page with the multiple operations they can call (create, delete and get extensions; set, remove and get the policy; and view the state of the app's membership and devices).

The multiple screens of the web interface can be seen in Figure 5.4, where the top left side is the login page, the bottom left side is the list of applications, and the right side is the operation page. The operation page is divided into three sections, Extensions, Policy, and Membership, where each operation has the corresponding text boxes to fill in with the arguments needed. The result of the execution of an operation is shown directly under it, as can be seen in Figure 5.5.

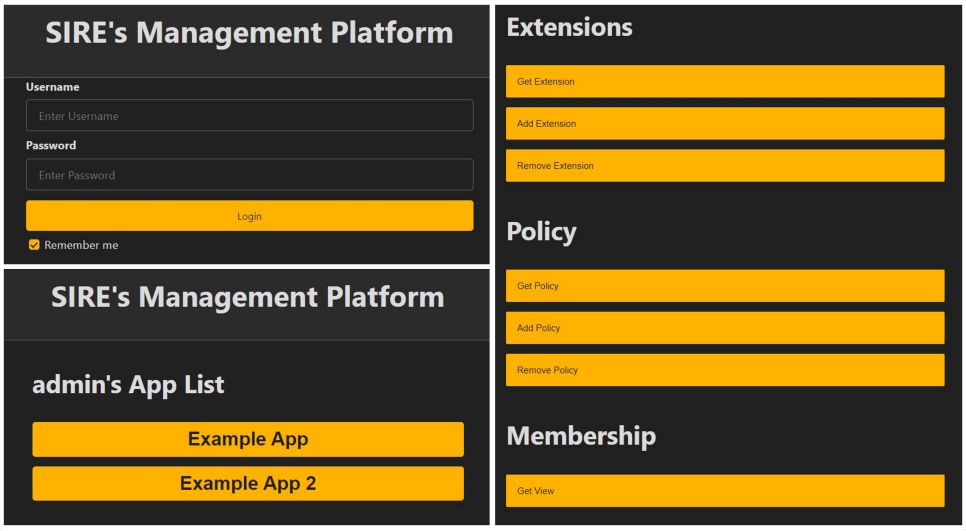


Figure 5.4. SIRE's web interface.

Figure 5.5. example of operation in SIRE's web interface.

5.5 Evaluation

In this chapter, we address the evaluation of SIRE. Firstly, we describe and discuss the evaluation of the throughput and latency of get, a read operation, and put, a write operation, performed on the Coordination Manager's KV store. After that, we discuss the evaluation of the $join$ operation, which includes the attestation protocol. With this evaluation, we want to answer the following questions:

- What is SIRE's overall performance?

- What is the performance of SIRE's attestation protocol?
- How scalable is SIRE?

All tests were executed in machines with the following characteristics: Dell PowerEdge R410 with two quad-core Intel Xeon E5520 (2.27 GHz) CPUs; two hardware threads per core; 32 GB of RAM; Ubuntu 20.04 operating system with OpenJDK RE 17.0.3. The machines are connected through gigabit Ethernet.

5.5.1 Read and Write data – *get* & *put*

To evaluate the performance of the get operation, we used 4 server replicas in 4 different machines and 11 client machines with a maximum of 3000 clients spread across them. We assume a worst-case scenario in which devices run their own proxies, imposing a higher load on SIRE servers. One additional machine was used to just gather the measurements without executing any operations in order to avoid the measuring process from affecting the results. The get operation was evaluated using a key with 100 bytes that obtained another 100 bytes in return. To minimize the impact the clients had on the performance, we used pre-processed get requests, which allowed us to reduce the processing done by the clients.

We performed this evaluation using rounds, having the number of clients increase incrementally from 1 to 3000. These clients were distributed equally between each machine. In each round, every client executed 10 million operations.

In each round, measurements of the throughput are taken around every two seconds by the leader replica that counts the number of requests in that period of time. This number is then divided by two to obtain the throughput and stored, updating the max throughput accordingly. At the end of the round, we perform an average of all these throughput measurements. For the latency, we measure it for each request in the client, store it and update the max latency accordingly. At the end of the round, we perform an average of all the latency measurements.

With this setup, we obtained an average throughput of $130K$ operations per second across all rounds and an average max throughput of $140K$ op/s. The best performance obtained was in the round with 800 clients, in which we measured an average throughput of $169K$ op/s and a maximum throughput of $172K$ op/s.

Regarding latency, we obtained an average of 3.6 ms across all rounds. The lowest latency was obtained on the round with 10 clients, with it being on average 0.6 ms, which is to be expected as there was a lower load, and requests could be attended to more quickly.

When it comes to the relation between latency and throughput, as can be seen in Figure 5.6, the latency started increasing when the average throughput reached $140K$ op/s with 1 ms of average latency, going to 4.5 ms with an average throughput of $169K$ op/s (800 clients). After that, the performance declined with each round, having a lower throughput and a higher latency than in previous rounds, due to saturation and a bit of memory thrashing due to the high number of clients in BFT-SMaRt. In the last round, with 3000 clients, we had an average throughput of $139K$ op/s and an average latency of 3.6 ms.

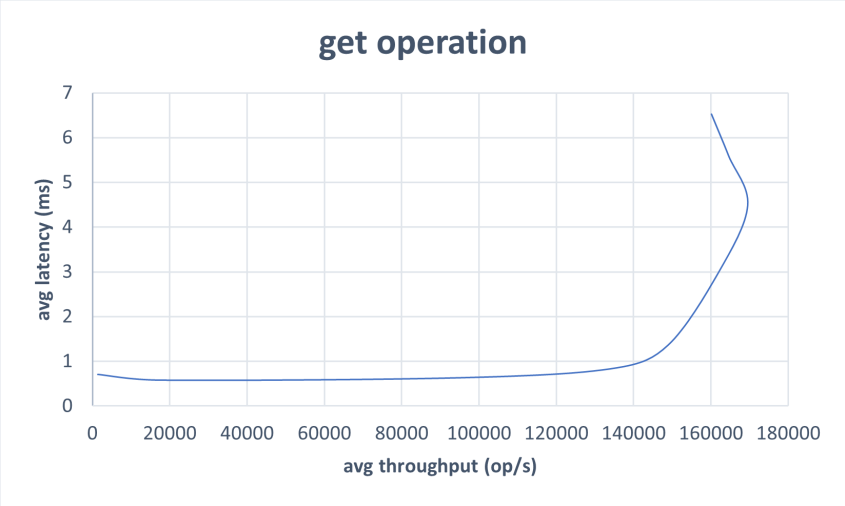


Figure 5.6. Performance evaluation of the get operation, using 11 client machines and a maximum of 3000 clients. The graph presents a relation of throughput and latency in function of an increasing number of simulated devices interacting with the system.

To evaluate the performance of the put operation, we had a similar setup that only differed in the maximum number of clients, which was 6000, due to the write client being less computationally heavy. We also assumed the same worst-case scenario in which the clients run their own proxies to impose a higher load on SIRE servers. The number of rounds was incrementally increased to accommodate the higher maximum number of clients from 1 to 6000. To take measurements, we employed the same method that was used for the get operation.

With this setup, we obtained an average throughput of $26K$ operations per second across all rounds and an average max throughput of $27K$ op/s. The best performance obtained was in the round with 3000 clients, in which we observed an average throughput of $34.3K$ op/s and a maximum throughput of $35.3K$ op/s.

As for latency, we obtained an average of 36.5 ms across all rounds. The lowest latency was obtained in the round with 10 clients, with it being on average 4.6 ms.

When it comes to the relation between latency and throughput, as can be seen in Figure 5.7, the latency increased at a steady pace until the average throughput reached $20K$ op/s with an average latency of 9.7 ms (200 clients round). From that point, the latency increased at a higher rate compared to the throughput, becoming even higher when the throughput reached around $30K$ op/s with a latency of 25.8 ms (800 clients round). From that point, the curve became almost a vertical slope due to saturation. In the last round, with 6000 clients, we had an average throughput of $34K$ op/s and an average latency of 59.7 ms.

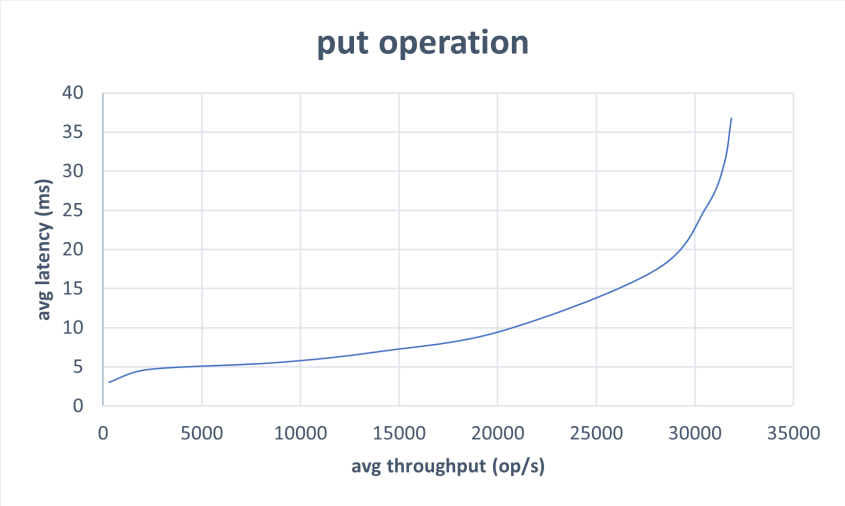


Figure 5.7. Performance evaluation of the put operation, using 11 client machines and a maximum of 6000 clients. The graph presents a relation of throughput and latency in function of an increasing number of simulated devices interacting with the system.

The difference in performance between the get and put operations is significant, with the throughput being almost 5 times higher on the get and the latency being almost 3 times lower. Such disparity is due to the get being executed in an unordered way by the server replicas in a single round trip, while the put must be ordered across all replicas, requiring the execution of the BFT-SMaRt consensus protocol to order every batch of operations. This protocol requires three additional communication steps and several cryptographic operations.

5.5.2 Attestation protocol – *join/attest*

In the case of the $join/attest$ operation, we used a similar setup with the same maximum number of clients and rounds as the put operation described in Section 5.5.1. To take measurements, we employed the same method used for the get and put operations described in Section 5.5.1. In this case, the average throughput was mostly constant through each round, around 160 op/s, with only the average latency increasing along with the number of clients, going from 25 ms with one client to about 72 s with 6000 clients. The reason for the constant throughput lies in the signing algorithm we used, Schnorr [92, 93], more specifically, the verification of signatures, which performs operations on elliptic points that are very computationally heavy, being the bottleneck of the attestation protocol. We performed a microbenchmark to evaluate the performance of the Schnorr [92, 93] signatures. On average, it took 5.435 ms for the verification, which means it can perform around 184 signatures per second, thus explaining the values we obtained for the attestation protocol.

Despite these performance values being significantly lower than the ones obtained for the get and put operations, these are still satisfying results given that the attestation protocol should only be executed ever so often and not something devices will constantly be running, like the get and put.

Figure 5.8. Performance evaluation of the $attest/join$ operation, using 11 client machines and a maximum of 6000 clients. The graph presents a relation of throughput and latency in function of an increasing number of simulated devices interacting with the system.

5.6 Wrapping-up SIRE and looking ahead

In this chapter, we detailed SIRE's implementation, which is now in a functional state, containing the majority of the intended features. Development is moving forward as planned, having only a few deviations from the original design and idea, which were needed to deal with challenges that arose during implementation. Although SIRE is in a functional state, there are still improvements and further developments to be done. Currently, our focus is on demonstrating how SIRE could be used as a solution in real-world scenarios.

For this purpose, we want to implement an application for the autonomous vehicle field where SIRE could be used to coordinate the movements of vehicles in order to solve problems such as intersection management [73]. By exploiting SIRE as a coordination service, we could solve these problems with the added security benefits of remote attestation.

We also intend to implement an application for Byzantine fault-tolerant machine learning. Nowadays, most machine learning implementations are distributed. Most of these rely on a core component, a parameter server, which is in charge of updating the parameter vector while workers perform the actual update estimation. Although the implementations are distributed, the workers cannot tolerate computation errors or attackers trying to compromise the system. In this scenario, SIRE can act as a parameter server, storing and aggregating parameter vectors in a Byzantine fault-tolerant way.

6 Co-simulation improvements and testing in Renode

VEDLIoT aims to improve the state-of-the-art for two areas of engineering that are notoriously difficult to test: IoT and Machine Learning. During the course of the project, we have been working on supporting the development of RISC-V-based ML accelerators using Renode [21] - an open source simulation framework by Antmicro. We aimed to support developers at all stages of development, starting with accelerator hardware design, with Renode's co-simulation capabilities.

The initial phases of development were focused mainly on CFUs - Custom Function Units [19], which are AI accelerators for Field-Programmable Gate Arrays, tightly integrated with RISC-V CPUs via the custom instructions mechanism [90]. The Renode simulation framework provides support for CFUs through Verilator [105], a fast simulator that converts Verilog (a hardware description language) to a cycle-accurate model in C++. More technical information regarding our work with CFUs and a description of Renode's use in Google's CFU Playground [48] can be found in deliverable D5.2 [62] for the VEDLIoT project. Here we present other improvements to the co-simulation capabilities of Renode, aiming to improve the support for the ML accelerator developed as part of Work Package 4.

We also present the recent developments in building a testing infrastructure for machine learning processing as well as describe the used technologies, their potential, and their issues. This chapter also includes a description of the developed solutions' workflow.

6.1 Improvements in the Renode co-simulation framework

Renode, in its nature, is a high-level functional simulator. It does not intend to be cycle accurate and focuses on fast prototyping, inspectability, and ease of integration with other tools. However, during the years of Renode development, we have identified the need for a more detailed approach in selected parts of the system.

This level of precision can be achieved via co-simulation with Verilator.

The very high-level flow of a co-simulated scenario can be summarized in a few steps:

1. CPU issues a bus transaction or a co-simulated instruction,
2. Verilator integration layer receives the operation and sends it to the verilated block,
3. The bus implementation provided by Renode translates the operation into a sequence of signal changes,
4. Meanwhile, the peripheral's clock signal is being toggled to drive the logic forward.

While previous deliverables focused on co-simulation with CFUs, Renode enables more scenarios including verilated peripherals. The framework also allows for work with memory-mapped peripherals. Currently, several buses for connecting peripherals are supported:

- AXI4,
- AXI4-Lite,
- APB3,
- Wishbone.

Precision is a key aspect of co-simulation with verilated blocks. To increase the precision, our work focused on introducing better alignment between the integration layer and the specifications of the supported buses, with the most emphasis on AXI4-Lite. The results of this work were published on the Renode repository [23], and in a dedicated repository listing sample Renode-Verilator integrations [24].

6.1.1 Improving transaction accuracy

In order to achieve this, firstly, we have focused on improving the accuracy of transactions between a peripheral and the CPU from the perspective of their conformance with the specification. Buses are, in short, a set of standardized signals constituting a communication system for data transfer between components. The specification defines these signals' behavior, relationships, and constraints.

In the case of AXI4-Lite, a sample write transaction can be divided into the following steps:

1. The initiator sets the address of the transaction via a handshake mechanism
2. Similarly, the initiator writes data
3. Lastly, the transaction recipient responds with a transaction completion status

The handshake mechanism is used to ensure that all relevant values (address and data) are transferred when both the initiator and recipient are in a correct state. A simplified handshake process for setting the address (`awaddr` signal) can look as follows:

1. the initiator sets the address to the `awaddr` signal
2. the initiator informs the peripheral that the `awaddr` value is correct by setting `awvalid`
3. the peripheral marks that it's ready to read out the `awaddr` value by setting `awready`
4. on the next rising clock edge, when both `awready` and `awvalid` are asserted, the handshake and the transfer are complete

During the latest period of VEDLIoT, we have worked on improving the accuracy of these procedures, striving to emulate the synchronous nature of signal transition in an imperative, sequential code while also keeping the implementation clear and easy to follow.

6.1.2 Trace generation and model evaluation procedures

Secondly, we have also introduced improvements in the fields of trace generation and model evaluation procedures. While trace generation is a feature of Verilator itself, the programmer gets to control the moment of evaluation and generation of trace steps.

When co-simulating with Renode, users give up a certain level of control over the simplicity of use - they only need to connect signal names in order to get a fully functional simulation. This simplicity comes at the cost of not being able to mark moments in which traces are generated explicitly. Users must rely on the integration layer's internal implementation in Renode.

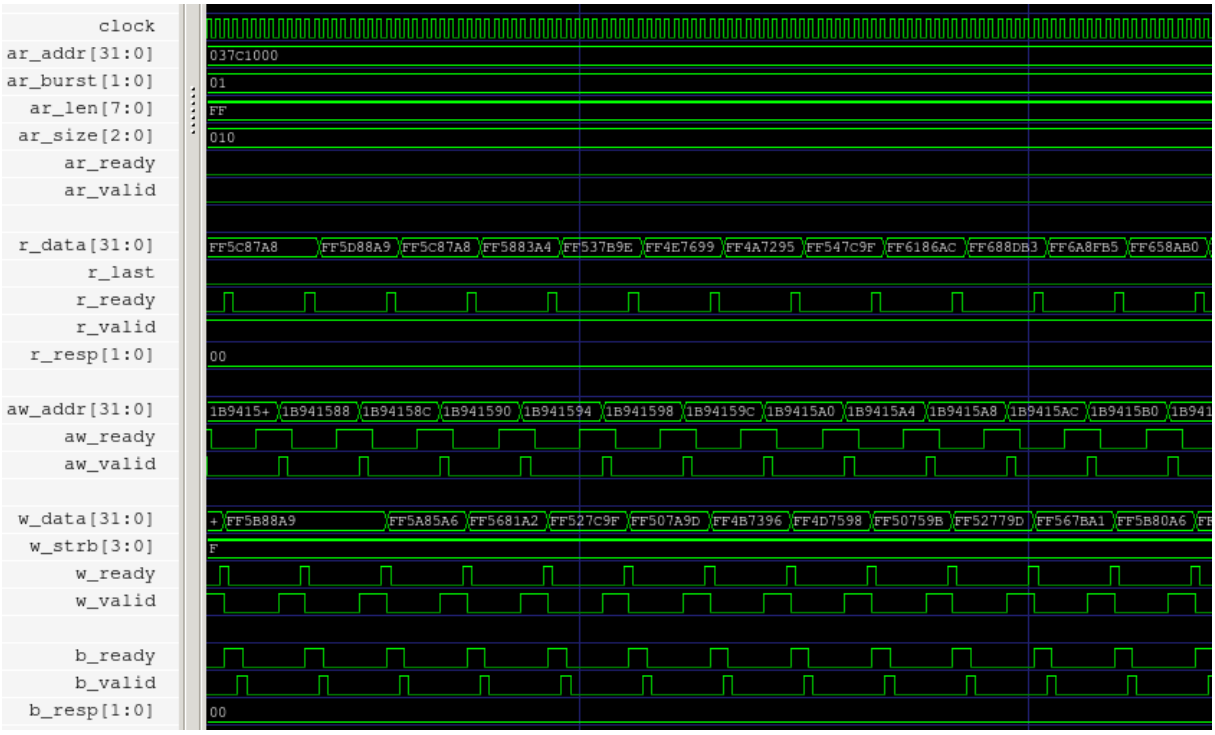


Figure 6.1. Sample waveform generated by Renode with Verilator

The initial approach in the Renode co-simulation library was to generate traces on every signal change. This approach had many benefits - it clearly showed the causality of changes and their precise order in time. It also reflected the implementation details of specific buses in Renode.

However, with the quasi-synchronous nature of HDL programming, these traces were not what hardware engineers expected. The biggest downside of generated traces was unbalanced clock signals, which typically indicate serious errors.

To make traces more understandable to their typical users, all trace generation is now being done only during clock signal changes, after the signal is changed and the model is evaluated.

After these adjustments, traces, as seen in Figure 6.1, are easier to follow for hardware engineers and provide a more accurate view of the verilated block's behavior.

6.1.3 Peripheral as transaction initiator and recipient

Then, we have focused on improving flows where a peripheral converted via Verilator acts as both the initiator and the recipient of a transaction. While it is typical for a peripheral to be the recipient, a situation when it is an initiator is common in scenarios involving Direct Memory Access (DMA) engines. In order to reflect the behavior of hardware in such cases, where each signal of every bus is evaluated simultaneously, we needed to adjust the behavior of Renode, where the buses were checked separately. At first, we introduced a mechanism that has allowed us to freeze the signal status of a bus when suspending its analysis (possibly mid-transaction), thus preventing its use by the peripheral. We were then able to grant control to another bus and resume signal status and analysis for the initial bus when needed. This flow enabled us to simulate the environment accurately, but unfortunately, the simulation speed was significantly low.

In an effort to vastly improve performance, ensure fine-grained control over processing stages, as well as represent the hardware description and behavior as closely as possible, we have divided the flow into the following stages:

1. Run the `pre-pos-edge` stage for all buses, preparing them for the upcoming clock rising edge
2. Set the clock signal high for all interfaces
3. Evaluate model
4. Run the `pos-edge` stage for all buses, allowing them to react to their state after the tick
5. Set the clock signal low for all interfaces
6. Evaluate model
7. Run the `neg-edge` stage for all buses

This improved scenario allowed us to reduce the runtime of some of our scenarios from over a minute to a few seconds. As future work, we plan to prepare a more thorough evaluation of the implemented changes.

6.2 Testing infrastructure for ML processing

In this section, we have described the developments in building testing infrastructure for ML processing required to verify the software created for the SoC used in the VEDLIoT project. The proposed infrastructure is based on the Google Cloud Platform and utilizes Renode Framework to simulate the target hardware and test the developed solution.

6.2.1 Difficulties of ML workflows

ML is a powerful tool that gives the user the to analyze vast amounts of data and make predictions or decisions based on that data. It has undoubtedly revolutionized the field of computer science, but the process of implementing an ML solution is often lengthy, time-consuming, and resource-heavy in general because ML algorithms

require a large number of resources in order to run effectively. The high resource consumption of ML solutions lies mainly in the complexity of the algorithms themselves. Many ML algorithms are highly sophisticated, with multiple layers and many parameters. These algorithms can take a long time both to train and run, especially when working with complex models, large datasets or different execution parameters.

The training process involves adjusting the algorithm's parameters so that it can accurately make predictions or decisions based on the data. This process can be computationally intensive, requiring a significant amount of processing power and memory, which comes at a cost. In order to make accurate predictions or decisions, ML algorithms need to be trained on a large and diverse dataset. Resource consumption of machine learning processes also highly depends on the amount of data used to train the algorithms. Additionally, machine learning algorithms often require significant amounts of memory in order to store the model parameters and intermediate results during the training process. This can be especially true for deep learning algorithms, a machine learning algorithm that uses multiple layers of interconnected nodes to analyze data. These layers can be vast, requiring a significant amount of memory to store.

6.2.2 Testing in the cloud

Cloud-based testing refers to testing software applications and systems using cloud computing environments. Using this technology allows users to replace testing software applications and systems on a local machine or on-premises with the cloud's computational power. This approach can pose significant advantages compared to local testing, the most evident one being far greater scalability. Cloud testing allows for creating a testing environment that can be easily scaled up or down as required by the current computational needs. This is particularly useful for testing applications that are expected to handle a large number of test cases or a sudden increase in traffic. With local testing, the computational resources are significantly less adaptable, especially when it comes to scaling down. What is more, cloud testing can be more cost-effective than local testing because it eliminates the need to purchase and maintain expensive hardware and software licenses, as cloud providers typically charge on a pay-as-you-go basis, which means that users only pay for the resources they actually use.

What is also important when it comes to projects like VEDLIoT is the superior accessibility of cloud-based testing solutions. This is particularly useful for teams that are distributed across different locations. With on-premise testing, all team members must either be in the exact location to collaborate and work on the testing process or use custom remote access solutions. While very well known and often used, these seldom provide collaboration features expected from a modern system.

Cloud-based testing additionally allows users to create testing environments that closely mimic production environments. Combining the cloud-based testing approach with Renode Framework will enable users to work with a highly accurate and controllable environment, which can be easily modified or replicated.

6.2.3 Proposed cloud-based testing infrastructure solution

The process of appropriate testing of complex systems and platforms like those developed in VEDLIoT is a process requiring a lot of resources and know-how of the

software we want to test. The aim of Task 5.4 was the development of a reliable system that project participants could use to test and verify the software for the SoC created as part of Task 4.6 and Task 4.7. To address this issue, we have developed a dedicated environment that will allow the user to successfully perform testing of their solutions with only a basic understanding of the testing infrastructure itself. The constructed system is based on the platform developed in Task 5.3. It will provide continuous integration capabilities, ensuring the software's fitness at each stage of development.

The infrastructure was built using the Google Cloud Platform (GCP) to ensure availability and enable users to benefit from testing in the Renode framework. The selected platform allows for easy scalability of used resources, helping us balance the need for computational resources with the costs of operation.

On a designated GCP runner, we have deployed a GitLab instance with a Docker image registry and the ability to spawn CI runners. The GitLab infrastructure is private to the project members but allows for easy publication of test results without exposing details of underlying tests. This is particularly helpful if tests were to use non-public data or ML models.

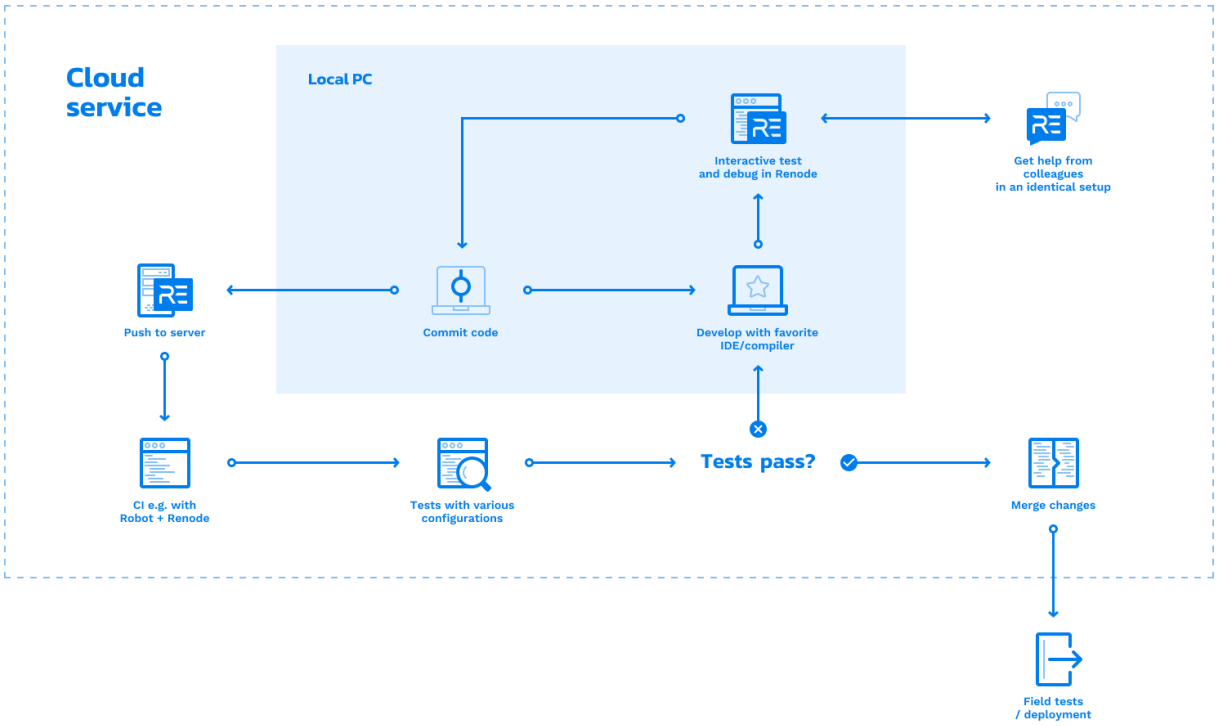


Figure 6.2. Renode cloud-based testing workflow

All of these elements lead us to the complete testing workflow, as presented in Figure 6.2. We envision the ideal workflow as starting with local testing with Renode - developers would use their local installations to verify the parts of their system they are currently working on.

When they are satisfied with their results and push the code upstream, the CI environment takes over and runs the developed payload against a range of tests. These tests can be easily reproduced locally if needed but can also benefit from the scalability of the cloud setup to run more test cases than users would be willing to verify

on their own machines.

6.3 Future work on the testing workflow

Recent significant improvements in the co-simulation capabilities of Renode allow us to present this feature as a viable alternative to full RTL simulation. In the next stage of the project, we plan to focus on the usability aspect and present users with an easy-to-reproduce testing setup while still improving the support for various buses and co-simulation scenarios. The work on this setup has already started.

The workflow of the testing solution starts in a repository containing all of the components necessary to begin testing in Renode. The centerpiece of this testing environment is a package containing Renode Framework, which will allow the simulation of all the elements of the target hardware, like CPU or all the necessary peripherals, as well as simulated hardware surroundings.

The repository also comes with a preinstalled Verilator for cycle-accurate simulation that can be used alongside Renode. It also contains a customizable test that will run on each `git push` to ensure the correctness of the changes with each commit and quickly identify potential problems. The tests are being built using the Robot Framework, which allows the user to write tests using easy-to-read syntax and generate reports from a test suite. Additionally, the repository contains a REPL (Renode Platform) file for the platform developed as part of Work Package 4 and sample tests to base upon.

After the testing repository reaches the required level of maturity, users will be able to fork it and update test cases with their own conditions, input data, emulated application, and co-simulated payload. This will follow the idea of ”Renode issue reproduction template” [22], which is used by the Renode community to report issues and suggest improvements.

6.3.1 Providing ML with data

In the traditional testing setup, the user cannot reliably control the physical environment of the tested hardware, but thanks to Renode’s support for strictly controlled external input, a completely simulated environment becomes a feasible alternative. Renode allows users to provide input to a wide variety of sensors like temperature, acceleration, gyro, or cameras. With the latest addition of support for time-synchronized multi-sensor data input [20], Renode becomes invaluable in testing various scenarios against ML payloads.

These features will be integrated with the testing environment, allowing users to easily generate the most complex setups to improve their ML solutions further.

7 DNN robustness evaluation

The most popular algorithms, deep neural networks (DNN), have improved their ability to handle a broad range of challenging ML tasks, from computer vision to language processing. However, the behaviors of DNNs are difficult to analyze because of their non-linear and large-scale nature. As a result, they are often deployed as black-box models without formal robustness and safety assurances. In particular, neural networks (NNs) are not robust to input perturbations, which makes them highly vulnerable to adversarial attacks in run-time settings. For example, in the context of image classification in a self-driving car, the neural vision of a well-trained NN can be easily fooled if a slight perturbation is applied to the same image it was trained and tested with without changing the context of the image. In this case, the car could confidently perceive a stop traffic sign image as a speed limit traffic sign in the autonomous driving environment, which could lead to catastrophic consequences [81].

This chapter presents a literature review about evaluating the robustness of DNNs. The main techniques, such as Branch and Bound and incomplete verifiers, are discussed. Then, a new approach is conceptually described, and experiments and results are shown to validate the proposal. In the end, we demonstrate that the new technique is more scalable and requires fewer iterations for robustness verification.

7.1 Evaluating safety and robustness for neural networks

The deployment of NNs in safety-critical settings is constrained by several limitations [103]. Therefore, it is pivotal to have a formal understanding or knowledge of the areas where NNs may be likely to wane in their decision-making under extreme scenarios, as well as the validation of their tendencies under potentially malicious inputs. However, several tools have been devised by researchers aiming to provide a formal safety and robustness guarantee for DNNs, which includes tools such as Satisfiability Modulo Theory (SMT) [87], mixed-integer linear programming [100, 65, 35, 70], convex relaxations and duality theory [91, 110, 91, 45], Abstract Interpretation [46, 113, 80, 16], and interval-based methods [117, 108, 107], DeepPoly [97], Bounded-Block Poly (BBPloy) [119]. All of these studies strive to analyze and verify the behaviors of DNNs under extreme worst-case scenarios when the input is/are defined within a certain range of perturbations. In practice, NN safety and robustness verification is the process of demonstrating desirable linear correlations between its inputs specified within a certain perturbation set and its output by optimizing a set of linear constraints imposed on the network [16]. In addition, the scalability and flexibility of NN analyzers are urgently needed as NNs get deeper and bigger in design. Their scalability and flexibility will give us a quicker and easier way to formally provide a guarantee that the network is safe and robust for all inputs within a certain perturbation range.

Three dimensions may be used to evaluate the effectiveness of safety and robustness verification algorithms for neural networks. The first dimension measures the tightness of the verification constraints, the second measures the computational complexity, and the third measures how effectively it cooperates with other DL models (like different state-of-the-art activation functions, such as ReLU, Tanh, Leaky ReLU, Sigmoid, Parameterised ReLU, and Exponential Linear Unit). There is a disagreement

between these dimensions. For instance, the verification algorithm's prudence and computational complexity are generally in conflict with one another. Each of the NN certification algorithms has a particular domain in which they may perform better, which is the basis for their significant advantage. For instance, less conservative algorithms are required for applications such as reachability assessment for robustness verification [112, 113, 102, 68, 96, 95], but fast-running algorithms are required for DL robustification [69, 40, 72, 43].

However, the generation of tighter linear bounds for DNNs in a formal verification fashion is done by techniques such as Satisfiability Modulo Theory (SMT) solver algorithms [34, 44, 87, 60, 61], and integer programming algorithms [100, 70, 18, 96]. Nonetheless, the intricacy of these algorithms in a conundrum scenario is directly proportional to the size of the network. This means that the bigger the size of the NN, the more complex these algorithms become and, as a result, the more difficult it is to provide a formal robustness guarantee for the network in question. In their work, "Lomuscio, Alessio", and "Lalit Maganti" [70] leveraged mixed-integer programming algorithms to perform exact robustness verification of a non-linear DNN, which they claimed reduced the computational burden and complexity of providing formal verification for the network. There is no doubt that this made a significant contribution to scalability optimization. However, in terms of generating tighter linear bounds on the ReLU-neuron while optimizing scalability, our network segmentation strategy outperformed their approach, and even computationally, our approach requires less computational resources than their approach.

According to what we currently know, the most accurate and sophisticated robustness verification tool that can provide a provable formal guarantee to a DNN is $alpha-beta-CROWN$ [29], which is a combination of $alpha-CROWN$ [114] and $beta-CROWN$ [109] incomplete NN verifiers. It is an award-winning NN verification tool in the 2021 and 2022 International Verification of Neural Networks Competition with the maximum overall score, outperforming several other NN verification tools. The foundation of $alpha-beta-CROWN$ was derived from the traditional bound propagation algorithm known as "$CROWN$" [117] and the "branch and bound" (BaB) [56] technique, and it scales to reasonably large convolutional neural networks (CNNs) [71] and can effectively work on GPUs. Thanks to the adaptable "autoLIRPA" library created by Wang et al. [109], $alpha-beta-CROWN$ can verify a wide variety of NN designs, including CNN, ResNet, GoogleNet, AlexNet, etc., and multiple activation functions. However, $alpha-beta-CROWN$ is expensive for verifying the robustness of a large network, even on a single image. Even with GPUs, it takes a significant amount of computational resources and time.

We suggest a network block segmentation approach, which we applied to the $alpha-beta-CROWN$ algorithm to improve the scalability of the verification tool and to reduce the need for high computational power in $alpha-beta-CROWN$. Interestingly, compared to applying the $alpha-beta-CROWN$ algorithm to the whole network at once, our method delivers superior verification scores and generates tighter bounds for ReLU-neurons in each segment or block.

In the sections that follow, we present four methods available in the literature that can be used for evaluating robustness. Following, we evaluate all the methods in comparison, using ResNet-50 (a 50 layers deep convolutional neural network) as a baseline. Finally, we show and discuss the results for robustness levels of our model

using three benchmark datasets.

7.2 Branch and Bound (BaB)

BaB is a robustness verification process that divides a non-linear optimization problem into two sub-problems while simultaneously relaxing each ReLU-convex neuron's vertex using an incomplete verifier, such as those specified in [100], [70], [18], and [96]. There are two steps in the BaB process: the first is the **"branching step"**, and it involves splitting each unstable ReLU-neuron into two sub-problems by adding split constraints $(z_1 > 0)$ or $z_1 \leq 0$ to the optimization problem. However, these additional constraints make the bounds tighter because they convert the unstable ReLU into a stable ReLU. The second step is the **"bounding step"**, in which we use an incomplete verifier to compute the lower bound for each sub-problem.

Figure 7.1 shows how branching constraints could be made more efficient with the use of complete verifiers. After the first split, the active and inactive zones have lower bounds of -2.5 and $+0.8$, respectively, compared to the original lower bound of -4.0. The result of 0.8 indicates that the lower bound of the branching step's additional constraint $z1leq0$ is greater than zero and that the sub-problem has been solved and verified. However, for the split constraint where $(z1 > 0)$, the sub-problem was not solved because it's lower bound of -2.5 (Figure $5b$) falls inside the optimization problem's negative axis. As a result, we must split the ReLU-neuron iteratively until all the sub-problems are verified within that region.

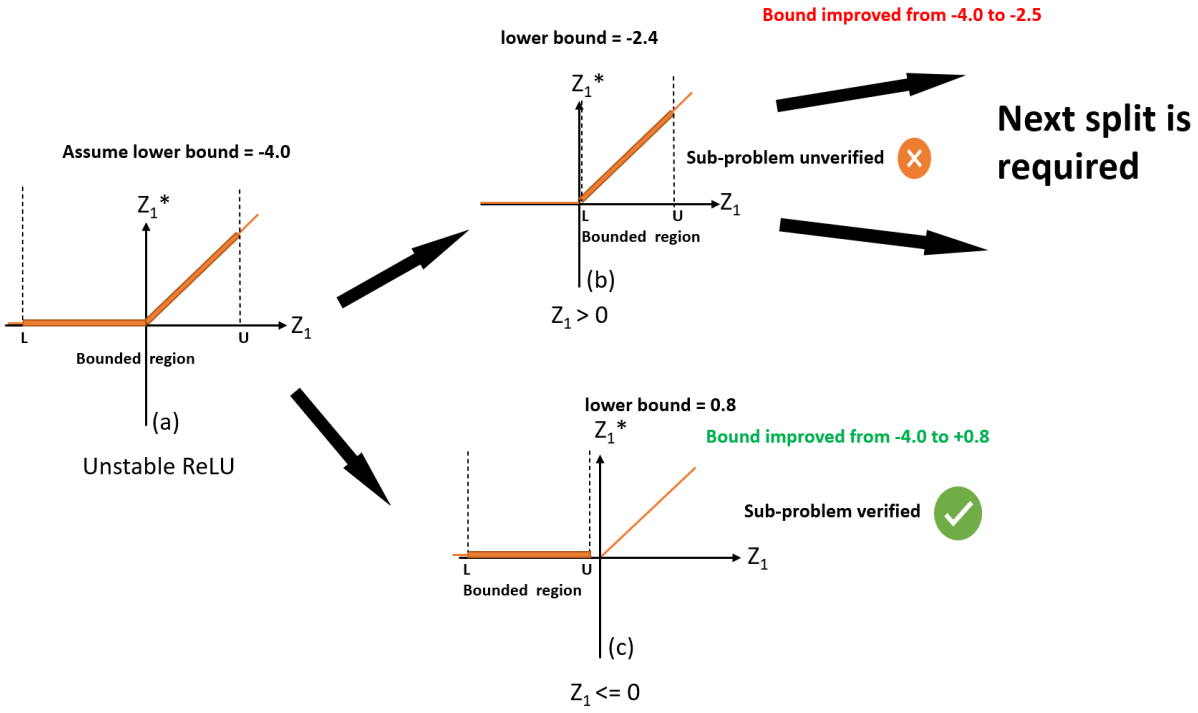


Figure 7.1. Illustration of bounding step in BaB process concerning a ReLU-activation

7.3 CROWN incomplete verifier

The traditional $CROWN$ verifier was the most commonly used incomplete verification tool for complementing the complete verification process in adversarial settings.

"$CROWN$" is essentially a linear solver algorithm [99]. It is a broad framework designed to verify and guarantee the robustness of DNNs with non-linear activation functions for specific classes in a dataset. It can handle advanced activation functions such as ReLU, sigmoid, and tanh because it has the power to create a linear bound for a given activation function using both linear and quadratic requirements. However, $CROWN$ incomplete verifier is too weak to satisfy the split constraint imposed by the BaB method because $CROWN$ operates entirely on bound propagation, similar to the back-propagation [111] in the normal NN. Although it is quite effective in the incomplete verification process, it is too weak in the complete verification process.

In an incomplete verification process using $CROWN$, if the requirement that $y^*_{CROWN} > 0 \implies y^* > 0$ is satisfied, the network robustness can be guaranteed for all inputs. In a bound propagation, $CROWN$ computes the network's output $y(x)$ by linearizing the input of the ReLU activation to determine the lower bound of each ReLU-neuron. This entails relaxing the convexity so that the network's x and $y(x)$ have a linear relationship. Hence, $y(x)$ is computed in equation 7.1 and is expressed as:

$$y(x) = W^3 ReLU(W^2 ReLU(W^1 x)) \tag{7.1}$$

Where W is the weight of the network, and the ReLU non-linear activation is given by equation 7.2:

$$ReLU(z) = Max(0, z) \tag{7.2}$$

Figure 7.2 shows how the ReLU-neurons have three possible outcomes depending on the constraints on their inputs.

Case 1***:*** The outcome of the ReLU-neuron will become linear if the output of the σ between $Z^{(i)}$ and $Z^{*(i)}$ (bounded region) are within the active region (i.e., $L \geq 0$), which makes all the values positive, in which case no linearization is required because the activation is already linear, as shown in Figure 7.2a.

Case 2***:*** The associated neuron can be deleted if the output of σ between $Z^{(i)}$ and $Z^{*(i)}$ (bounded region) falls within the inactive region (i.e., $U \leq O$), as illustrated in Figure 7.2b, because the weights $W^{(i)}$ linked to that neuron are dropped during back-propagation [111].

Case 3***:*** The entire output of the ReLU-neuron becomes convex/non-linear if the output of σ is between $Z^{(i)}$ and $Z^{*(i)}$ (bounded region) falls within both active and inactive region (i.e., $L \leq 0$ and $U \geq 0$) as depicted in figure 7.2c. As a result, the σ must be relaxed to create a bounded region whereby the network will be stable, allowing its robustness properties to be evaluated.

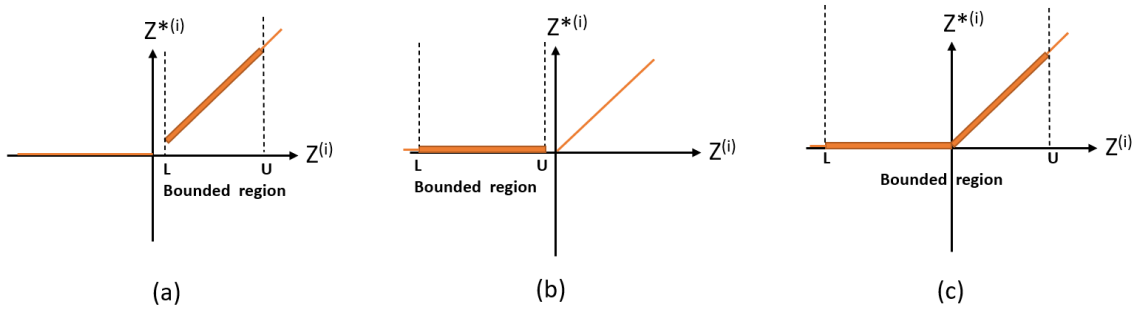


Figure 7.2. Illustration of the possible bound region of a ReLU-neuron

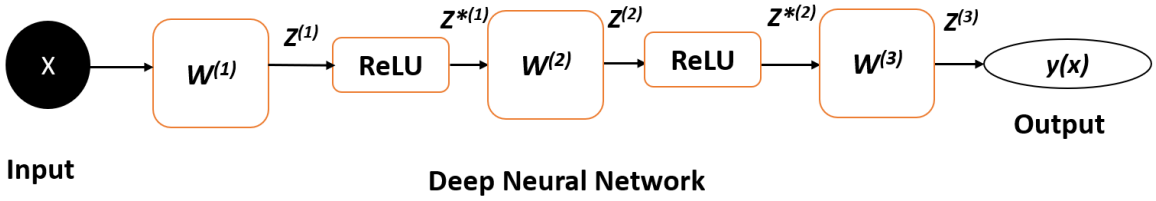


Figure 7.3. Illustration of a block diagram of DNN

"$CROWN$" attempts to relax the convexity of the activation function by defining the lower and upper bounds for each output ReLU-neurons in the network using two linear functions on the activation function, as shown in Figure 8. However, since $CROWN$ lacks the ability to generate tighter linear bounds, the actual robustness level of the model cannot always be determined. In some cases, it cannot give the exact robustness level of the model because of its loose bounds. In such scenarios, the model could be robust, but we might come to the opposite conclusion by looking at the bounds produced by $CROWN$.

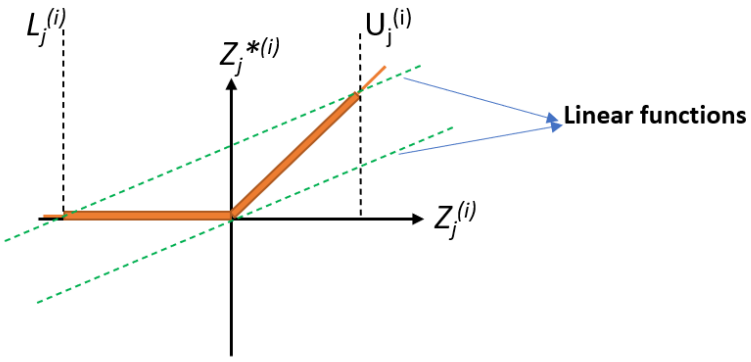


Figure 7.4. Illustration of convex relaxation introduced by the tradition CROWN incomplete verifier

As the Figure 7.4 illustrates, for j^{th} ReLU-neuron in the layer i, the equation 7.3 holds:

$$z_j^{*(i)} = ReLU(z_j^{(i)}) \tag{7.3}$$

Assuming the inputs of $ReLU(z)$ are bounded, we have; $L_j^{(i)} \leq z^{(i)} \leq U_j^{(i)}$ and if unstable, we have $L_j^{(i)} \leq 0 \leq U_j^{(i)}$, which could be otherwise refers to as pre- and post-activation bounds.

Through backward-bound propagation, convex relaxation aims to establish a linear relationship between each hidden neuron in the network and the output. The inequality solver may be driven from Figure 7.3 such that

$$y(x) = z^{(3)} \tag{7.4}$$

Therefore, $z^{(3)} = W^{(3)T} z^{*(2)}$ which is an inequality for $y(x)$ with respect to $z^{*(2)}$, from where we represents as $y(x) = W^{(3)T} D^{(2)} z^{(2)}$ by replacing the ReLU-activation with a diagonal matrix $D^{(2)}$ such that the inequality still holds. This paves the way for us to establish a linear relationship between $z^{*(2)}$ and $z^{(2)}$. The $D^{(2)}$ is designed in such a way that the lower and upper bounds can be easily maintained for each ReLU-neuron.

$CROWN$ establishes a linear relationship between the output $y(x)$ and the input x and computes the lower bounds of the final output neurons. It cannot optimize the neuron-split linear constraint introduced by **BaB** because it generates certain slack bounds that prevent the convex from being relaxed appropriately. Therefore, it could not provide an accurate robustness guarantee to the model for all inputs. Given this limitation, Xu et al. [114] proposed the cutting-edge $alpha - CROWN$ incomplete verifier, which generates tighter linear lower bounds than the $CROWN$ algorithm, and this is confirmed in our experiment.

7.4 Alpha-CROWN incomplete verifier

This $alpha - CROWN$ algorithm focuses more on computing the linear lower bound of the ReLU-neurons, based on the simple fact that if the requirement $y(x) > 0$ for $x \in C$ is satisfied, the robustness of the model could equally be verified and guaranteed. The fundamental technical insight of this approach is that it generates several lower bounds, as shown in Figure 7.5 by the dot lines. As a result, there are numerous alternatives for choosing the lower bound for each ReLU-neuron, which means we could choose any lower bound as long as its slope falls between 0 and 1. This indicates that $alpha - CROWN$ is an adjustable bound generator, which generates multiple linear lower bounds with values between 0 and 1. When optimizing the split constraints imposed by BaB, we choose the solution with the tighter linear bound so long it is not a negative value. Equation 7.5 demonstrates how the lower bound could be adjusted by values between 0 and 1 through optimizing the α.

$$z_j^{*(i)} \geq \alpha_j^{(i)} z_j^{(i)} (0 \leq \alpha_j^{(i)} \leq 1) \tag{7.5}$$

Where, $0 \leq \alpha_j^{(i)} \leq 1$ is adjustable. Furthermore, the lower bound of the $alpha - CROWN$ verifier could be computed as:

$$y^*_{CROWN} = Min\ \alpha^T_{CROWN} x + Const_{CROWN} \qquad where \quad x \in C \tag{7.6}$$

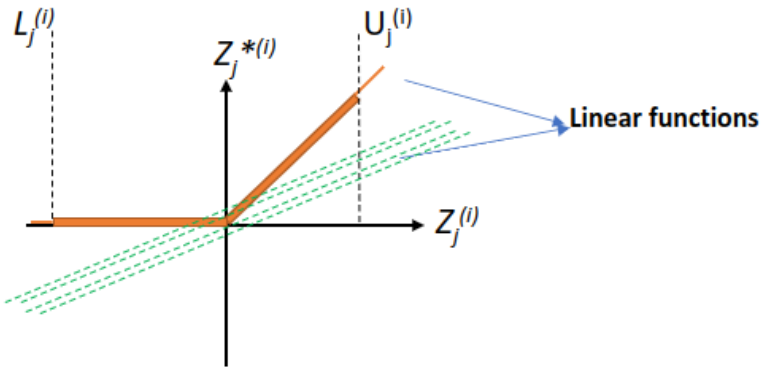


Figure 7.5. Illustration of convex relaxation introduced by the alpha-CROWN incomplete verifier

Where α is the slope of the lower bound, between 0 and 1. α is an adjustable term, which can be adjusted by values between 0 and 1. Our goal is to optimize α to find the best possible linear lower bound for each output ReLU-neuron. We choose the value of α in gradient ascent and apply a "closed form" solution [77] to get its inner minimization for the purpose of optimization.

However, the $alpha - CROWN$ algorithm has an additional advantage in that it tightens not only the lower bounds but also the ReLU-neurons' pre-activation $(z^{*(i)})$ and post-activation $(z^{*(i)})$ values. This helps enable a stronger robustness guarantee and a better verification score than the $CROWN$ and other existing incomplete verifiers in BaB's complete verification settings.

Although $alpha - CROWN$ is a powerful verifier, it often took several iterations of splitting the ReLU activation in BaB settings before a verification property could be guaranteed to hold. When $z_1 \leq 0$ during backward bound propagation, it takes time for $alpha - CROWN$ to address the split constraint. It is often quick and effective to optimize the split constraint when $z_1 > 0$. Because of these limitations in alpha-CROWN, Wang et al. [109] proposed beta-CROWN, an advanced incomplete NN verifier quicker than alpha-CROWN at solving both sub-problems in BaB settings.

7.5 Beta-CROWN incomplete Verifier

$Beta - CROWN$ is a bound propagation-based method that can completely encapsulate neuron split constraints via an optimizable feature (beta), which is a "Lagrangian multiplier" [28] built from either primaeval or dual dimensions. The "Lagrangian multiplier" always conditioned the split constraint $z > 0$ to be either 0 or 1, making $beta - CROWN$ faster and producing better bounds than $alpha - CROWN$ verifiers with neuron-split constraints while remaining as efficient and parallelizable as $CROWN$ on GPUs [109]. The lagrangian multiplier in beta-CROWN is an adjustable parameter that can generate multiple lower bounds for the ReLU-neurons between 0 and 1, similar to what we described in alpha-CROWN. The lower bounds become equally tighter from 0 to 1 as the value of beta increases. We took into account all of the positive split constraints while bound back-propagating through the network from the output to the input layer, and we then used the Lagrangian multiplier to provide various lower bounds, and we chose the one with the closest value to the ground truth. The rationale behind the $beta - CROWN$ can be expressed as equation 7.7:

$$Min\ [x \in C, z^{(2)} \in < 0] y(x) \geq Max[\beta \geq 0] Min[x \in C] W^{(3)T} D^{(2)} + \beta^T S^{(2)} z^{(2)} + const. \quad (7.7)$$

7.6 Alpha-beta-CROWN

The most recent, fastest, and most accurate NN verification algorithm, $alpha - beta - CROWN$, use potent GPUs to ensure formal verification of the model. As far as we know, it is the only verifier that can effectively handle the BaB's split restriction on high-dimensional NNs like ResNet50, VGG19, GoogleNet, InceptionV3, etc. However, in order to function efficiently on these networks, a lot of GPU is needed. The $alpha - beta - CROWN$ combines the alpha and the beta parameters in a traditional $CROWN$ verification. Its application in a BaB setting produces a complete verification algorithm that can tell us "Yes" the network is robust or "No" the network is not robust. The advantage of the $alpha - beta - CROWN$ method is that it adds a double extra constraint to the network that, if solved, creates tighter bounds and makes the ReLU-neuron split linear optimization problem work better, which helps transform the unstable ReLU-neurons in the network into stable neurons. "Branch and bound" can be expressed with the equation 7.8:

$$Min[x \in C, z \in Z]\ y(x) \geq Max[\beta \geq 0] Min[x \in C] (\alpha + P\beta)^T x + \alpha^T \beta + C \quad (7.8)$$

Where $x \in C$ and P be the probability of β and z is all the split constraints.

The $alpha - beta - CROWN$ [109] is computationally expensive for verifying the robustness of large networks. Regardless of whether we use a GPU to scale up or accelerate the verification process, it still requires a significant amount of memory space and computation time. In order to respond to the high computational demand posed by the $alpha - beta - CROWN$ algorithm, we suggest a network block segmentation approach in which we divide the network into three different segments. Instead of applying the $alpha - beta - CROWN$ algorithm to the entire network (i.e., from the first hidden layers to the output layer), we applied it to segments and mapped their outputs together to compute the final overall output of the network. This improved the verification tool's scalability and minimized the requirement for the heavy computational demands imposed by the $alpha - beta - CROWN$. Our approach generates tighter bounds for ReLU-neurons in each segment or block and, interestingly, delivers superior verification scores compared to applying the $alpha - beta - CROWN$ algorithm to the entire network at once. This allowed us to outperform the approach suggested in [109] and achieve a higher overall verification score for the entire network. Our network segmentation is expressed with the equation 7.9:

$$\sum_{i=1}^{n} Min\ y(x) \geq Max[\beta_n \geq 0] Min[x \in C] (\alpha_n + P\beta_n)^T x + \alpha_n^T \beta_n + C \quad where\ \ x \in C \quad (7.9)$$

Then the overall output is computed as:

$$\frac{Min\ y_1(x1) + Min\ y_2(x2) + Min\ y_3(x3) + - - - + Min\ y_n(xn)}{N - c} \tag{7.10}$$

Where N is the number of segments, and c is a small constant value.

Figure 7.6 shows how we divided a NN into three parts. This lets us make a summary of the entire network, which we then use during bound back-propagation on the network to solve the "ReLU-neuron split" constraints that the $alpha - beta - CROWN$ imposes on the network. Technically, the ReLU activation is the ending point for every hidden layer in a NN. Initially, we attempted an intuitive segmentation method [64] by splitting the network such that each segment ends with a ReLU activation. However, we notice that doing so requires extra computation during the bound back-propagation, and instead, we choose to divide the network into segments, such that each ends with an affine layer, as illustrated by Figure 6. In comparison to the earlier technique, the latter requires less computation and yields a better verification score.

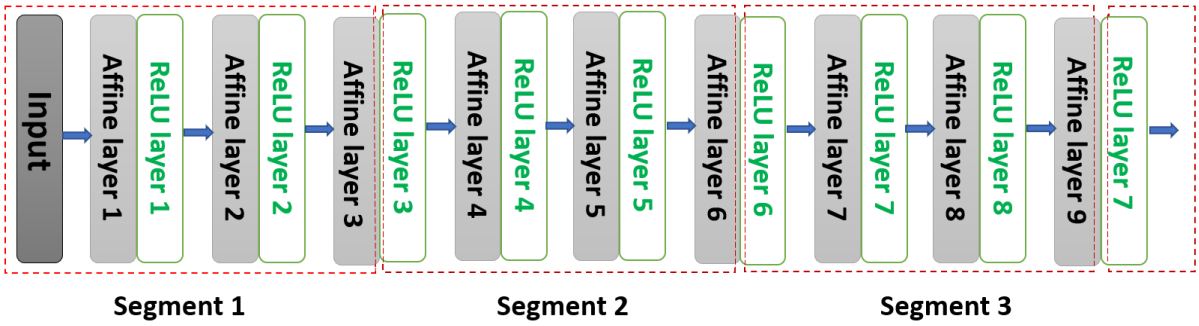


Figure 7.6. Illustration of the designed neural network segmentation technique for network summarization inbound propagation

Moreover, segmentation of a residual network such as ResNet50 [115] requires extra care due to its skip connection, which is its most pivotal feature. As presented by the curving line in Figure 7.7, it shows how it allows the network layer to establish "shortcut connections" with a previous layer. Accordingly, an "intrinsic" residual block [115] that may be leveraged to divide the network is made up of a group of layers that are present between a layer and its skip-connected layer. However, it could be feasible to strike a better balance between speed and accuracy if the number of segments or the size of each segment could be selected with a more flexible approach. We have not looked into this yet, so we will leave it for our future work.

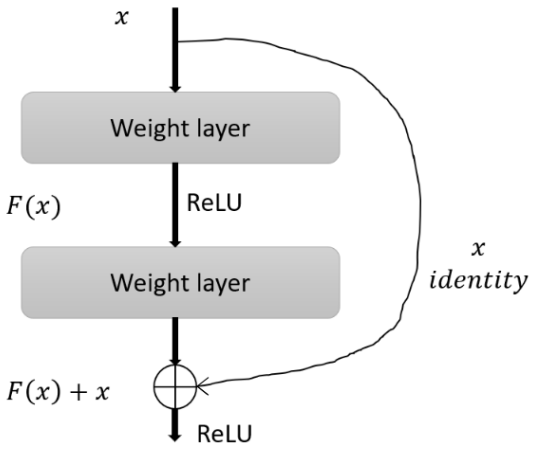


Figure 7.7. Illustration of a residual learning block

7.7 Experimental validation network and datasets

We developed our model from scratch using the ResNet50 DL architecture as a baseline. It is pivotal to emphasize that we are not leveraging a transfer learning model [54] for our evaluation. Rather, we are employing a comparable architecture that gives us the freedom to choose the network's hyperparameters and the number of hidden layers. Thus, we initially created our network using ResNet50's default hyperparameter settings. However, we performed some hyperparameter tuning [118] thanks to a Python open source library called **"OPTUNA"** that can be found in the PyTorch framework [88]. The hyperparameter optimization enabled us to select the best hyperparameters that provided the model with the highest validation accuracy score and the least amount of loss. After the hyperparameter optimization process, we designed our final ResNet50 DL model, which we used to evaluate our network segmentation algorithm.

Concerning the robustness verification, we verified and assessed the robustness level of our built ResNet50 DL model using the following benchmark datasets: **CIFAR10** [89], **MINST** [67], and **GTSRB** [101].

The CIFAR10 dataset consists of $60,000$ images divided into 10 distinct classes. Each image in this dataset has a pixel size of 32×32 and three colored channels (red, green, and blue). It is a balanced dataset with $6,000$ images in each class.

The MNIST dataset contains $70,000$ images, $60,000$ of which are from the training set and the remaining from the test set. It has a total of 10 classes, where each class consists of $7,000$ images with a pixel size of 28×28. MNIST is a greyscale handwritten digit image dataset with numbers ranging from $0-9$. It is extrapolated from two NIST databases, where 50% of the graphics were created by the staff of the Census Bureau and the other 50% by high school students.

GTSRB is a dataset containing images of German traffic signs and can be used to train DL image classification models for autonomous vehicles. These are common traffic signs used across the member states of the European Union except for a few member states. The dataset comprises $12,630$ test images and $39,209$ training images from 43 classes of traffic signs (speed limit, crossing, stop, traffic signals, etc.). The number of data points in each class varies, resulting in some classes having a small number of

images while others have a huge number. With a file size of around 314.36 MB, downloading the dataset does not require a lot of time or memory space. It has two distinct folders: train and test. The train folder is divided into sub-folders, each representing a particular class, and each class has a variety of traffic sign images. It was first published at IJCNN 2011 [98]. It should be emphasized that each of these datasets represents a multi-class image classification benchmark that has been used in a separate image classification use case. For example, the GTSRB is used in the field of autonomous and advanced driver assistance systems.

7.8 Result and Discussions

Our ResNet50 DL model's resilience is quantified and verified against the L_∞ norm onslaught, which is defined by a fixed ϵ of adversarial perturbation using the three datasets described in Section III (a). Traditionally, each x_0 is described by the value of q_1, representing the luminance of each pixel it is composed of. However, following the L_∞ norm adversarial onslaught with epsilon set to 0.001, each pixel corresponds to an intensity interval $[q_i - \epsilon, q_i + \epsilon]$.

Therefore, in each segment, we added an adversarial disturbance ϵ to the pixels of x_0. Then, we used the $alpha-beta-CROWN$ verification algorithm, in which our task was to verify that each segment in the designed ResNet-50 model could produce output that could classify all perturbed inputs with the same class as the original input. Also, our task was to verify whether each segment could satisfy the condition described by $y(x) > 0, \forall x \in C$, where C is a set of perturbation. However, if all of these two requirements are satisfied by the model's overall output, we conclude that the model's robustness is verified for a certain amount of pixel distortion ϵ on the input image. The output of every segment in the network is mapped to produce the model's overall output, as described in equation 7.10. Furthermore, the adversarial tolerance of the model was also determined by the magnitude of the perturbation size ϵ. In this study, we evaluate the model's resilience with various epsilon values, and at a point where the robustness requirements are no longer met by the model, we consider that epsilon value as the model's adversarial tolerance point.

In order to evaluate our approach, firstly, we used the $alpha-beta-CROWN$, $alpha-CROWN$, $beta-CROWN$, and $CROWN$ algorithms over the mentioned data sets. Table 7.1 shows the results of applying those algorithms to the entire model. On the other hand, Table 7.2 presents the results obtained from the network segmentation approach.

We assessed the resilience of our network using the three benchmark datasets. Furthermore, we utilized a L_∞ norm as our L_p norm and a perturbation size ϵ of 0.001 for evaluation.

The conventional $CROWN$ algorithm's poor verification score may be attributed to its inability to perform well in both approaches, as demonstrated in tables 7.1 and 7.2. This demonstrates that the $CROWN$ cannot handle the neuron-split constraint in a BaB context. Looking at the same tables, the $alpha-CROWN$ and $beta-CROWN$ algorithms did not perform well individually in either approach. However, their verification scores were somehow optimized compared to the traditional $CROWN$ algorithm. Furthermore, as shown by their number of branches in the tables, the $CROWN$, $alpha-CROWN$, and $beta-CROWN$ algorithms required a significant number of it-

erations before their individual verification could hold in a BaB setting.

The $alpha - beta - CROWN$ algorithm combination achieves a higher verification performance score than the other algorithms when employed separately. Nevertheless, when we tested our model using the aforementioned three benchmark datasets, our network segmentation strategy beat the standard application of $alpha - beta - CROWN$. Table 7.2 shows that, compared to the alpha-beta-CROWN standard used in a BaB setting, our technique is more scalable and requires fewer iterations for a verification to hold.

	CIFAR10			MNIST			GTSRB		
Algorithms	time (s)	branches	%score	time (s)	branches	%score	time (s)	branches	%score
$CROWN$	5525.7	509606	**10.4**	2369	74209	**18.0**	3578	64209	**16.9**
$\alpha - CROWN$	5707.68	67933	**24.8**	1800.56	22339	**49.6**	2156	11339	**36.0**
$\beta - CROWN$	4340	48023	**49.8**	4107.70	6107.70	**55.3**	4001.70	14933	**59.30**
$\alpha - \beta - CROWN$	212.3	9400	**77.4**	3180	4507.11	**65.8**	1011	8933	**70.80**

Table 7.1. A comparison of techniques for NN robustness verification without the use of our network segmentation method.

	CIFAR10			MNIST			GTSRB		
Algorithms	time (s)	branches	%score	time (s)	branches	%score	time (s)	branches	%score
$CROWN$	2720	909606	**11.8**	1064	60401	**20.0**	1074	81094	**17.4**
$\alpha - CROWN$	1612	40136	**29.1**	1601	13737	**51.4**	1043	26217	**37.1**
$\beta - CROWN$	1120	61014	**51.7**	3413.50	7900	**60.4**	5141	5037	**67.10**
$\alpha - \beta - CROWN$	114.3	7203	**79.9**	2319.23	3740	**73.20**	986	7139	**76.13**

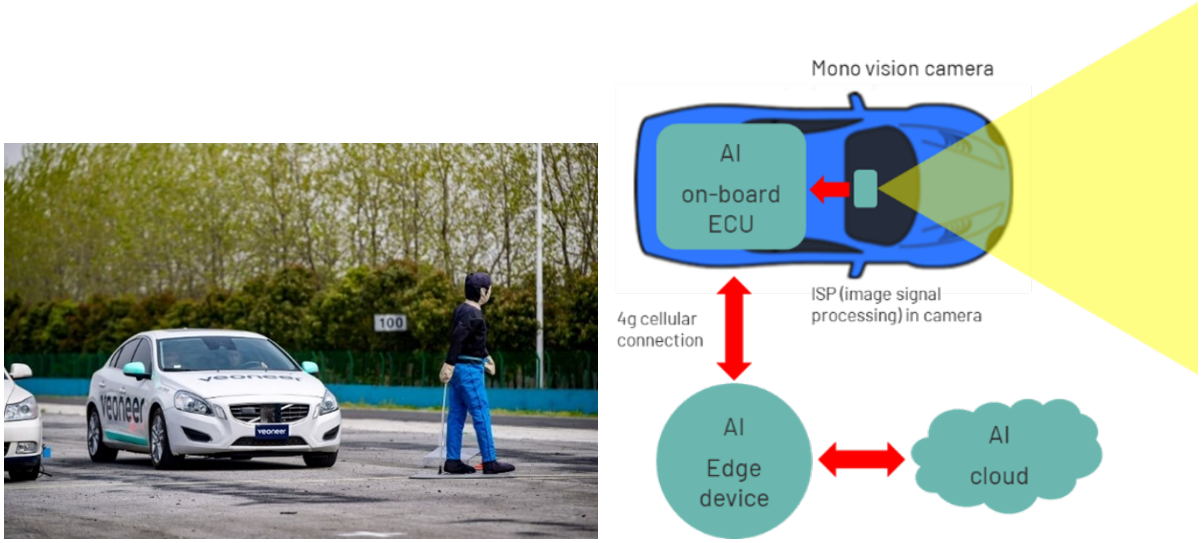
Table 7.2. A comparison of techniques for NN robustness verification using our network segmentation method.

8 Monitoring and mitigation strategies of run-time errors

Issues concerning the use of AI in the context of road safety have been becoming increasingly important in the automotive industry in the recent years. This importance has also been taken into account as part of the VEDLIoT project, in which the automotive use case is one of the most important ones, developed as part of T7.4. This section concerns the monitoring and mitigation of run-time errors in a system using AI/ML models applied to the use case of Pedestrian Automatic Emergency Braking (Pedestrian AEB), see D7.2, Section 4. Below, we describe how we have defined categories for run-time errors and how examples of run-time errors related to the pedestrian detection system can be used to illustrate other potential run-rime errors. Then, we discuss existing strategies for mitigation of such errors and propose possible improvements. Finally, we go over the achieved safety of the system in relation to existing standards.

As stated in Deliverable D5.2 [62], we have identified the need of a monitoring solution capable of evaluating the performance of AI/ML models, in- and output data quality, communication robustness and capacity in real-time.

According to its description, Task 5.5 was intended to develop adaptation or reconfiguration mechanisms, which can arguably bring the system back to some safe state, to react to unforeseen situations. A safe state will mean different things depending on the selected application. In this chapter, we are using the VEDLIoT automotive use-case of Pedestrian AEB as an example. The safe state could mean that we warn the driver when the automatic system is not operating to its full performance and the driver has to take control. In this document we try to reason of how to extend the system intended function so that this transfer of control is less necessary.



(a) Testing of an AEB function *(b) Distributed processing of sensor data*

Figure 8.1. The automotive use case

The Automotive use-case of Pedestrian AEB is illustrated in Figure 8.1 (see D7.2 for

further explanation). We start by identifying possible run-time errors, their detectability, and possible mitigation solutions.

8.1 Categories of run-time errors

The categorization of the run-time errors relates to whether the error occurs due to factors that are internal to the vehicle or external. These two categories are then further divided depending on whether it is possible to mitigate the error, which yields four categories, see Table 8.1.

Table 8.1. Run-time error categories

	External	**Internal**
Mitigable	Section 8.2.1	Section 8.2.3
Non-Mitigable	Section 8.2.2	Section 8.2.4

External is defined as something that depends on the environment, e.g., the surroundings and other vehicle systems. Internal is defined as the subsystems mounted in a vehicle, including sensors, central processing unit, communication devices, and power supplies. Mitigation is defined as an activity that reduces the impact of the run-time error so that the intended system function is still maintained to a level that does not require any user involvement. Error detection and identification is the first necessary step, but appropriate countermeasures or alternative processes adapted to the scenario are also required. This will be discussed in Section 8.3.

8.2 Example system with run-time errors grouped in categories

In this section, the relationship between a vehicle and its environment will be explored given the use case for detection of a pedestrian in order to initiate automatic emergency braking, as said before, developed in Work Package 7.

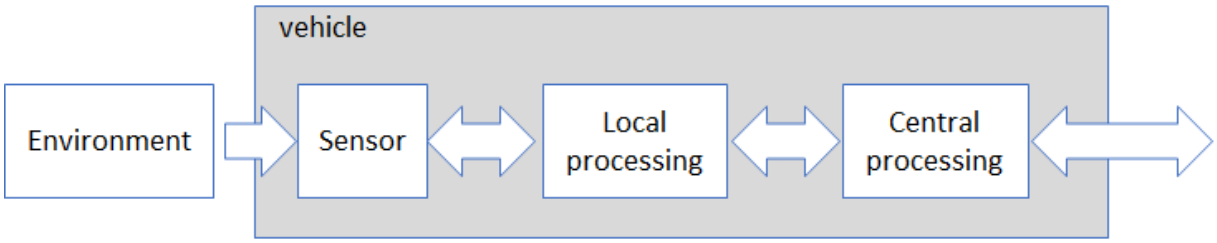


Figure 8.2. Interaction between a vehicle and its environment

Run-time error categories

- E External
- I Internal
- M Mitigable
- N-M Non-Mitigable

Environment

- E/N-M Out of the ODD
- E/N-M Unknown context

- E/N-M Weather
- E/N-M Unknown objects
- E/N-M Light conditions
- E/N-M Interference
- E/N-M Unexpected scenario
- E/M Localization error
- I+E/M Hard maneuvers creating unexpected dynamics

Sensor connects External to Internal

- I/N-M Electric failure
- I/M Calibration loss/error
- I+E/N-M Small crash impact
- I /M Supply voltage variations
- E/N-M Blockage (full, partial)
- I /N-M FOV boundary performance
- I+E /M+N-M Low SNR (dirt, aging of lens or radome)
- I/N-M Point cloud correlation (information leakage)

Internal Connection #1

- I/M Comm data crosstalk
- I/N-M? Timing/clock error due to (capacitive) loading
- I/M EMC

Local processing

- I/N-M Electric failure
- I/N-M Processing parameter loss/error
- I/N-M Memory overwritten
- I/N-M Insufficient data precision, rounding errors
- I/N-M Insufficient data dynamic range
- I/N-M Unexpected processing time (interrupt handling)
- I/N-M Incorrect output data (specification error)
- I/N-M Incorrect re-programming

Internal Connection #2

- I /M Communication data crosstalk
- I/N-M? Timing/clock error due to (capacitive) loading
- I/M EMC
- I/N-M Scheduling/addressing/router error

Central processing

- I /N-M Electric failure
- I/N-M Processing parameter loss/error
- I/N-M Memory overwritten
- I/N-M OS issues (content switching, interrupts, priorities)
- I/N-M Not enough resource allocation
- I/N-M Resource allocation delay
- I/N-M Incorrect output data (specification error)

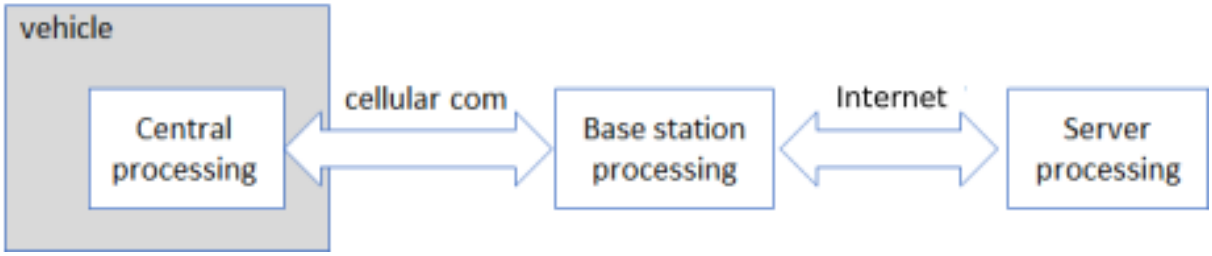


Figure 8.3. Interaction between a vehicle and the cellular communications infrastructure

Cellular com connects Internal to External

- E/N-M Not enough resource allocation
- E/M Resource allocation delay
- E/N-M Low SNR
- E/N-M Retransmission required
- E+I/N-M Handover
- E/N-M Interference

Base station processing

- E/N-M Electric failure
- E/N-M OS issues (content switching, interrupts, priorities)
- E/N-M Processing code implementation on OS (Insufficient data precision, rounding errors, word lengths)
- E/N-M Interfaces (data format, protocol)

Internet

- E/N-M Not enough bandwidth
- E/N-M Resource allocation delay
- E/N-M Retransmission required
- E/N-M Addressing issues (routers, lost packets)

Server processing

- Out of scope for this report.

The following sections aim to describe the reasoning for the categorization of the automotive use-case different run-time errors.

8.2.1 External, Mitigable

Ego localization can be calculated based on multiple sources like Global Navigation Satellite System (GNSS), sensor Simultaneous Localization and Mapping (SLAM), and vehicle Inertia Measurement Systems (IMS). Vehicle dynamic behavior can be well modeled, so it is likely that any instantaneous localization error is detected. The only time this is not true is when the system has just started unless there is some memory from when it was turned off.

As with the localization problem, good vehicle modeling may help counteract the effects of hard maneuvers. IMS information will also support mitigation of this potential error. In cellular communication, data transfer resources have to be requested and acknowledged before transmission, which is handled by a base station. The vehi-

cle does not have information about a possible resource allocation delay, which could generate processing errors. This could be mitigated if a model for this delay could be created based on known parameters like physical location, time of day, traffic density, and more. Including this model will allow for adapted processing mitigating the effects of allocation delays.

8.2.2 External, Non-Mitigable

For many Non-Mitigable external errors, we have unforeseen scenarios like unexpected objects in the field-of-view of the sensors. For ADAS functions, this could be an elephant on the road. From a functional standpoint, we may not be able to classify the object but may still determine that it is blocking our travel path. The intended function may be compromised, but the system will detect an anomaly and warn the driver.

Operating out of the Operational Design Domain (ODD) with respect to harsh weather, bad lighting, or unknown context, i.e., not knowing the physical position of the vehicle, will sometimes be difficult to identify. The used sensor types will behave differently with respect to these factors, and it may not be obvious that the ODD border has been crossed. Due to this, it will be hard to mitigate the problem, but it may be possible to generate a driver warning based on uncertainty criteria.

Many types of sensor interference may be identified based on entropy calculations. With AI, minor interference like single-pixel modification may cause missed detections or wrong classification. This is hard to detect and must be considered non-mitigable.

Sensor performance may be compromised due to small crashes, for example, in parking areas. The sensor data may look reasonable but introduce bias due to unexpected field of view. Based on long-time statistical data, it is possible to identify the error, but during the initial phase after system startup (leaving the parking lot, for example), this is an undetected fault.

Most current sensor systems are capable of detecting full or partial hard blockage of the sensor field-of-view (FOV). This sensor cannot deliver the expected information, and the full system cannot operate with full performance when this issue occurs, so the driver must be informed about cleaning the sensor surface.

Low sensor Signal-to-noise ratio (SNR) can be caused by multiple physical effects. Dirt, snow, and rain on the sensor facia or aging of the facia material are typical reasons. As for the blockage, the problem may be detected, and the driver may be warned, but the system will be non-operational.

Multiple communication parameters may affect the vehicle system's performance. They may be divided into physical problems and communication system design problems. Low SNR is a physical problem, and although it could be the result of bad system design, it is assumed not to be in this case. The low SNR may require the retransmission of data packets, and the scheduling and protocol for this is part of the communication system design.

When the vehicle is traveling, it will need to change base stations and thereby require handover and the effects of that are dependent on the communication system design. If the base station does not have sufficient communication resources (due to,

for example, limited RF bandwidth), data transfer will be delayed or lost. All these errors are detectable but cannot be mitigated in real-time. A warning to the driver would most likely be the appropriate action.

Interference of the communication link will require a change of resource allocation, using a different time-frequency slot for data transmission. If the interference is continuous and wideband, there may not be any available resources to counteract the error. Errors due to interference may be mitigable up to a certain level, but a warning to the driver is the only option above this level.

Electric and software failures in the base station may be hard to detect and mitigate by the vehicle system. Software routines adapted to anomaly detection may have to be included in any vehicle system software running on the base station processing devices. Even if the base station has multiple redundant processing units, it may not be possible to identify a problem from the outside, i.e., the vehicle system, and demand a switch to any backup hardware or software within a maximum time limit. The base station operates as a black box with special resources available to external users.

Suppose data should be transferred from the base station to any server on the Internet. In that case, additional insecurities arise since other non-related users with unknown priorities may affect the communication links. Data packets may be lost or delayed by how the route between the base station and the server is selected. Neither of these problems is guaranteed to be detected within a limited time frame.

8.2.3 Internal, Mitigable

As described in Section 8.2.1, vehicle dynamics modelling may help detect and mitigate errors due to hard maneuvers. IMS information will also support mitigation of this potential error.

System calibration information created either during production or as part of runtime statistics may be compromised or lost. The easiest solution is to have multiple copies of this information stored at different local locations that can be accessed through multiple independent communication channels.

Any state-of-the-art hardware system has supply included quality supervision functions. Depending on the type of problem it may in some cases be possible to prioritize the quality of energy supply for some system functions.

As described in Section 8.2.2 low sensor SNR can be caused by multiple physical effects. By detecting dirt, snow, and rain on a sensor facia it may be possible to adapt the system function for the current scenario. This could be by relying more on data from some types of sensors. This may keep the intended function at a reasonable level.

Electromagnetic interference (EMI) and electromagnetic compatibility (EMC) errors which have not been handled during the system design phase and may be caused by uncontrolled external systems may be hard to protect from. The mitigation possibility is to use alternative communication routes.

8.2.4 Internal, Non-Mitigable

A well-defined system can detect an electric failure, but it cannot keep the intended function operating on the faulty hardware.

Low-speed vehicle crashes may generate errors in sensors directly but also in the communication and power supply links. If the system is damaged when turned off it may, in some cases, be hard to detect the error at system startup. Based on long time statistical data it is possible to identify the error (see Section 8.2.2).

At sensor FOV boundaries, the performance drops off rapidly. The effect with single-point targets is well understood and modeled, but with real-life distributed targets, the effect is much more complicated. The solution would be to use only information within the designed FOV or multiple sensors with partially overlapping FOVs. When relying on a single sensor, this is a non-mitigable error.

Timing errors on the clock or data lines within the system, caused by, for example, capacitive loading, may distort the communicated information. The error may be detected, but it is hard to remove the error without physical manipulation.

The same is valid for processing parameter errors as for system calibration information (see Section 8.2.3). This information is also created during development/production and may be compromised or lost. The easiest solution is to have multiple copies of this information stored at different local locations that can be accessed through multiple independent communication channels.

Overwritten memory containing either data or program code will cause severe system errors. The system may hang completely, but it will restart if it is well-defined with watch-dog functionality. This will cause moments of undefined system operation, which may jeopardize the intended function.

Processing errors caused by combinations of insufficient data dynamic range, low data precision, and perhaps mathematical rounding errors may generate erroneous results that do not fulfill the intended function requirements.

Interrupt handling could be a source of problems due to its possible randomness, which may cause occasional long processing cycles where data is not produced with the expected time synchronization.

Although mainly a design/specification problem, incorrect output data may be generated due to calculation overflow or saturated data. Depending on the specification agreement between subsystems on how to report special cases of data, this may be a source of run-time error.

Many automotive systems operate with over-the-air updates, which will require the re-programming of subsystems in the vehicle. An incorrect or erroneous software update may cause total system failure and fewer detectable errors.

Sensor data is sent to a vehicle central processing unit and from there through a wireless communication channel (5G) to a base station edge processing device. All of these are controlled by their individual OSs and processing switching and priorities. A delay in time or insufficient resource allocation may create an error in the intended automotive function. Other problems, like the ones related to communication scheduling or addressing through routers and switches, may be detected using time stamps.

8.3 Mitigation options/strategies

The most common mitigation solution is based on redundancy, where the same system function can run on multiple local or off-board platforms. These platforms do not have to be dedicated to the intended function but may be allocated for the function in case of failure in the primary platform. All platforms should have individual power supplies and multiple interchangeable communication interfaces. Redundancy may be applied in different ways, as shown in in Figure 8.4.

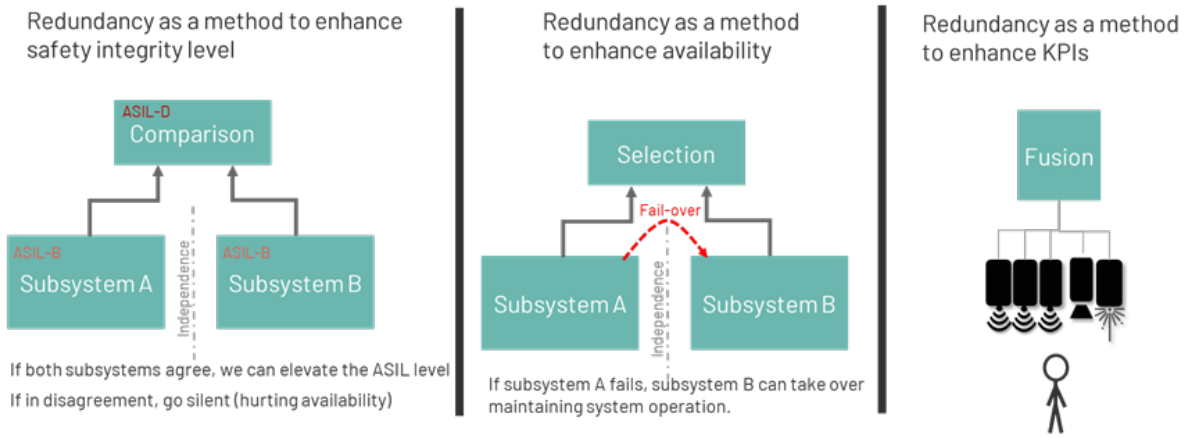


Figure 8.4. Redundancy applied to enhance safety, availability, and fusion results.

To avoid a central fault supervisor, the error detection functionality should be running on all platforms while supervising its own and all other platforms. Figure 8.5 illustrates the effect for the automotive use case with at least three separated processing platforms. The arrows only indicate the additional communication needed for real-time monitoring. This could generate an immense overhead in processing and data transfer. The correct trade-off must be defined for the possible improvement in the Safety of the Intended Function.

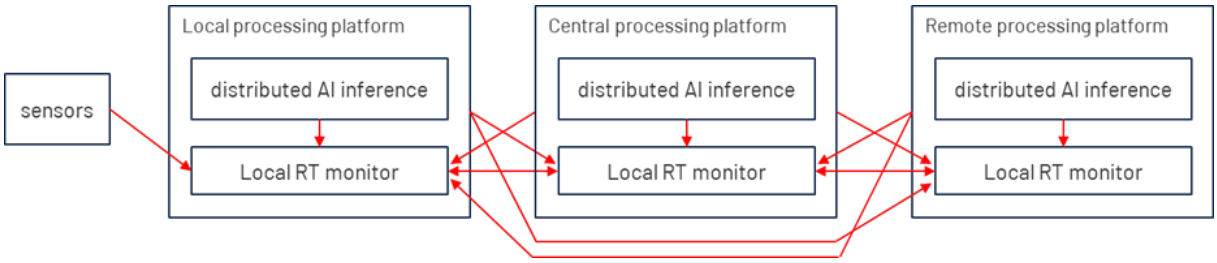


Figure 8.5. Error detection distributed between separate processing platforms

The first step to mitigation is error detection. Some possible error detection solutions have been described in Section 8.2 and subsections. They include hardware detectors, statistical analysis, and behavior modelling. The software implementations can be based on AI as well as deterministic algorithms. In our example of Pedestrian AEB, we assume an AI implementation distributed over many processing nodes. We have identified the need for a separate real-time monitor, which is also distributed over multiple hardware processing nodes.

Based on ISO26262 requirements, most modules have performance/error reporting

functions, but this is insufficient when considering SOTIF (ISO 21448). We have to analyze data between modules and also inject data into modules and evaluate the response. This is illustrated in Figure 8.6.

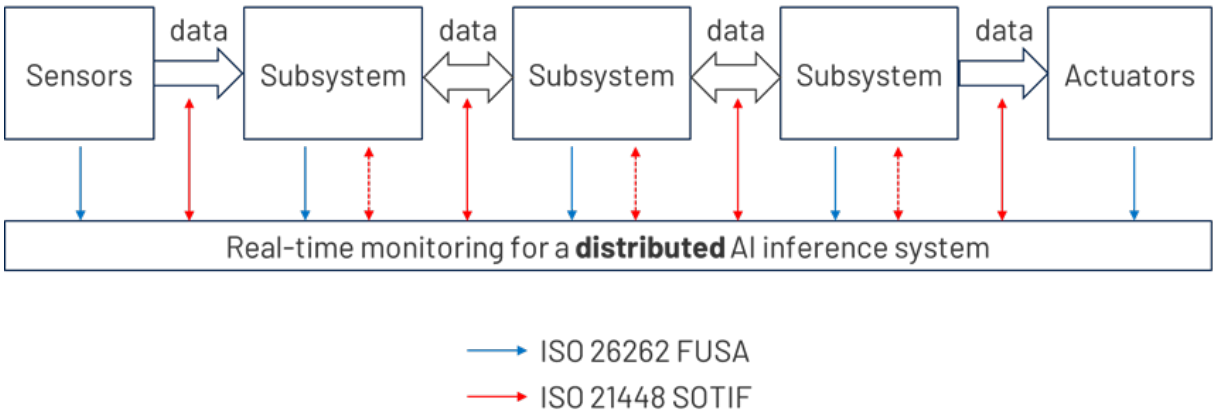


Figure 8.6. Application of FUSA and SOTIF for monitoring a distributed AI inference system

8.4 Possible ways of increasing the mitigation level

The automotive use-case application is designed to be reconfigurable. This is primarily done to guarantee sufficiently low response latency (breaking for a pedestrian). How to implement a reconfigurable VEDLIoT AI system is investigated in other VEDLIoT work packages. This reconfigurability could also be used to change the processing based on other errors detected by the real-time monitor. This may require a more advanced real-time monitor capable of solving updated resource allocations due to detected errors. This will be further explored in Task 5.5 during the rest of the VEDLIoT project.

8.5 Achieved system reliability and safety

The full system safety case argumentation is important. An evaluation of this has been carried out by Veoneer. It was presented at the Scandinavian Conference on Systems and Software Safety Conference in Gothenburg in December 2022 [14]. The automotive use case has been used as an example. The presentation is part of this deliverable as a separate document. A state diagram with the likelihood of transferring between states is a good tool to evaluate the safety level of any system. An example of an automated driving process is shown in Figure 8.7. Without this, it will be hard to determine the improvements created by additional functions (redundant or complementary). The critical paths may be identified, and the focus should be on minimizing the likelihood of entering these paths.

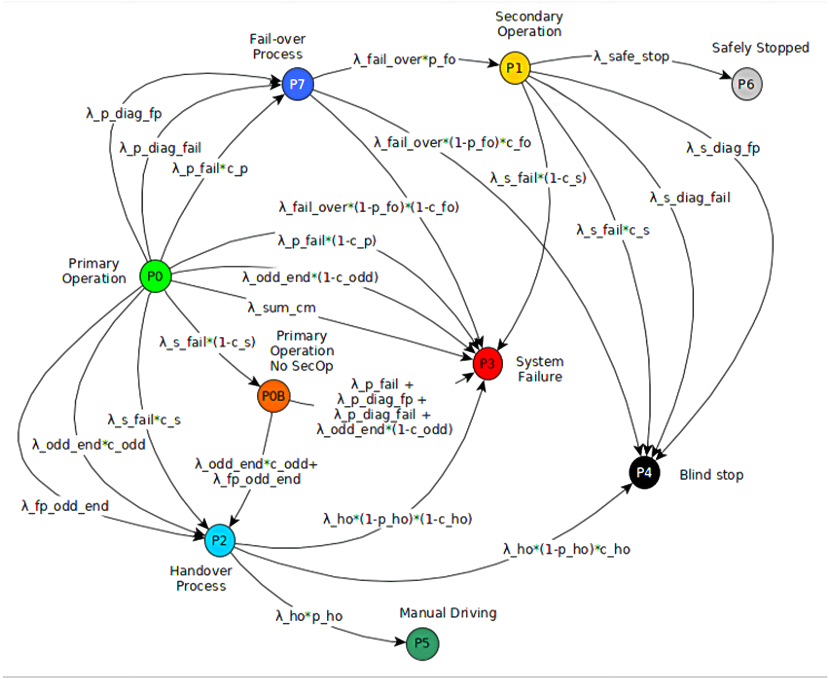


Figure 8.7. Description of an automated driving process, courtesy Thorbjörn Jermander, Veoneer

Figure 8.8 shows the probability of ending up in a certain state. In this example, the second operation state probability is very low, which is inefficient since we are not using this extra function very often. This could be managed by a better real-time monitor as described in Section 8.4.

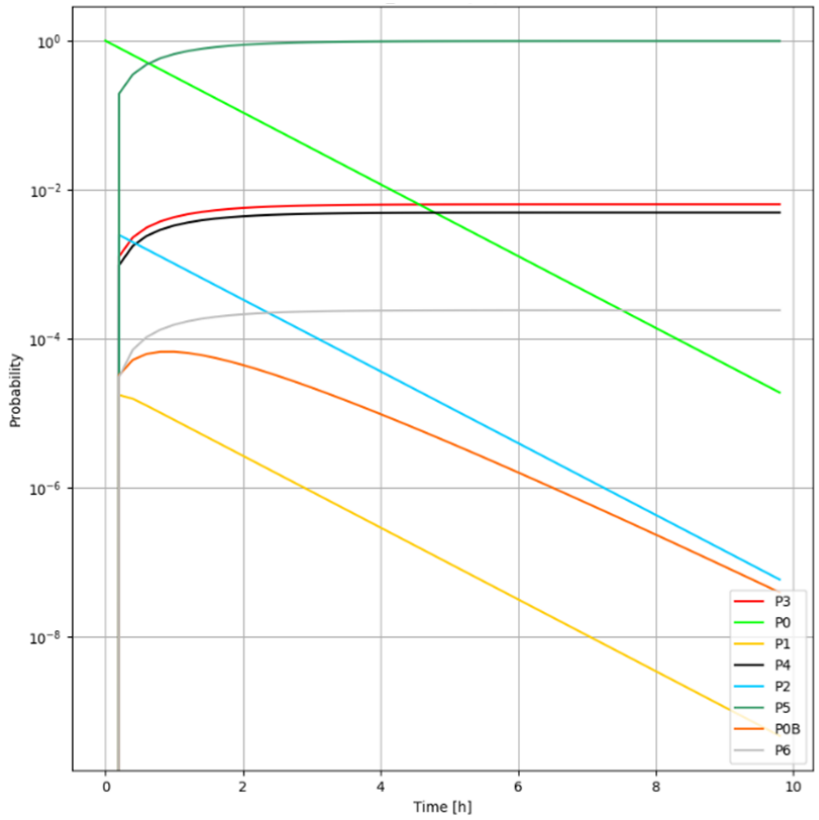


Figure 8.8. Probabilities of being in a certain state as a function of time.

This tool and other similar do already exist and are being further developed by industry, academia, and regulators. We will follow this development as described in Section 8.6, but we will not do additional work within WP5 of VEDLIoT.

8.6 The relation to existing standards, conclusions and outlooks

International groups are looking at improvements in safety argumentation and proposals for new safety standards. This includes Safety Strategy, ADS/ADS Feature description, Safety Criteria Definitions, Vehicle-level Safety requirements, and Safety Validation & Assessment. Veoneer is part of workgroups within this important area and will report the progress in the final VEDLIoT WP5 deliverable.

Our work, as presented at the Scandinavian Conference on Systems and Software Safety, has led us to several conclusions regarding safety for the VEDLIoT project, and some insights for the future outlook in this area. First of all, a well-defined and agreed-upon holistic system design approach is essential for ensuring the safety of IoT solutions. What is more, new methods, tools and standards need to be adopted in order to compile a well-operating, effective environment in terms of safety. New international specifications, e.g. ISO TS 5083, are of great importance as generic references, and they need to continue being developed in order to deal with the challenges this area presents. Another means that has potential for achieving safety targets are incremental safety assessments, proven to be extremely effective independent confirmation measures for maturity monitoring. These conclusions will help drive our future work in the project, with emphasis on errors likely to be included in the distributed DL model inference.

9 Overall Achievements and Future Work

During this period, our work on Tasks 5.1, 5.2, 5.3 and 5.5 of the VEDLIoT project continued. The Task 5.4 was started and will continue in the next period. This deliverable document is organized into 9 chapters, 6 of which describe technical progress.

In Chapter 2, we have described the structure and progress in work packages on a per-task basis. The following paragraphs discuss the highlights of our most recent work and the future plans aimed at supporting the VEDLIoT project goals.

In Chapter 3, two lines of work are still in progress concerning Task 5.2 (Security support for distributed execution and communication). The first line concerns communication, with the upcoming development of an MQTT-based trusted communication bus for attested WebAssembly processes. The second concerns the further evolution of our WebAssembly trusted execution run-time for Intel processors, with increased TRL in support of an industrial use case outside VEDLIoT.

In Chapter 4, we presented the collective work of partners regarding the time-of-check to time-of-use challenges that remote attestation and certification face in the context of IoT. Firstly, we have given an overview of the seriousness and specificity of TOCTOU problems for IoT devices resulting from resource constraints of such devices, their operational environment, supply chain, vulnerability management, and others. Then, we highlighted the importance of developing a solution capable of software validation appropriate for IoT devices and described AutoCert as a proposed mechanism. Our future work in this area will involve developing a highly-optimized remote attestation and certification mechanism for low-end IoT devices and dissemination of the developments discussed in this project via multiple outlets. We also aim at pushing this work as a potential solution for the implementation of the EU Cybersecurity Act for IoT.

Implementation of SIRE is further discussed in Chapter 5, building on the information provided in Deliverable 5.1 [86] and Deliverable 5.2 [62]. We delved into the full scope of SIRE's features, now operating in its fully functional state, its implementation architecture, the details of SIRE server and proxy implementations, as well as the process we have used to evaluate the service. The future work planned in this area will involve using SIRE in a real-life scenario, for coordinating autonomous vehicles movement, with an added benefit of Remote Attestation. An application for Byzantine fault-tolerant ML using SIRE as a parameter server is also planned.

In Chapter 6, we have presented the improvements to Antmicro's open source simulation framework - Renode. We also provided an overview of testing performed within the said framework and described a cloud-based testing platform developed as a part of the project. The next step involves preparing a reproducible setup to let other partners easily use the CI infrastructure created in T5.4. It will, in spirit, follow the "Renode issue reproduction template" [22] and will allow users to create test cases for the accelerator SoC and run them in Renode without additional technical work. We also described further development in the space of co-simulation in Renode, allowing users to create complex setups in which the majority of the system is simulated in a functional fashion, with certain parts represented as cycle-accurate verilated models. In the next period we will continue to improve the co-simulation

support, as well as expand the support for WP4 accelerator SoC - by adding new and improving already available LiteX peripherals.

In Chapter 7, we have described the methods of evaluating the robustness of Deep Neural Networks and presented a new solution for that problem utilizing novel approach to the $alpha - beta - CROWN$ algorithm, significantly improving the solution's scalability and shortening the verification process. In safe AI, evaluating the robustness of deep learning models is essential. Many of the methodologies used by researchers and industry specialists to gauge the robustness of their models are focused on identifying constraints that they believe, if satisfied, could indicate robustness or a lack thereof. In the project scope, we are investigating new ways of robustness quantification. Future research trajectories that could produce remarkable outcomes include:

- **Adversarial vulnerabilities - Investigating the impact of adversarial attacks on deep learning models:** This approach can aid researchers in identifying the weaknesses of a model and where it is most susceptible to failure.
- **Feature replacement:** Another potential avenue for research in this field could be the concept of substituting under-performing segments within the network after assessing their robustness levels. This could be done before quantifying the overall robustness level of the model, which could aid in constructing robust models for specific problems.
- **Ensemble robustness quantification:** By combining different techniques, we may be able to give some formal verification to the behaviors of the models before their deployment in critical real-world systems.

Chapter 8 described the strategies for monitoring and mitigating run-time errors based on the use case of Pedestrian Automatic Emergency Braking (Pedestrian AEB). This chapter categorized the errors into groups and discussed the ways of mitigating errors in every group and methods of increasing the mitigation level. Lastly, the chapter compared the achieved system reliability and safety level in relation to the existing standards.

10 References

[1] Confidential computing consortium. https://confidentialcomputing.io/projects/.

[2] Enarx. https://enarx.dev.

[3] Introducing json. https://www.json.org/json-en.html.

[4] memory64 proposal. https://github.com/WebAssembly/memory64.

[5] PolyBench/C. http://polybench.sf.net.

[6] Secp256r1. https://neuromancer.sk/std/secg/secp256r1.

[7] SQLite, speedtest1. https://sqlite.org/cpu.html.

[8] Veracruz. https://veracruz-project.com.

[9] Wasi-nn proposal. https://github.com/WebAssembly/wasi-nn.

[10] Wasi-parallel proposal. https://github.com/WebAssembly/wasi-parallel.

[11] WebAssembly core specification. https://www.w3.org/TR/wasm-core-2/.

[12] WebAssembly micro runtime. https://github.com/bytecodealliance/wasm-micro-runtime.

[13] Hipeac tech transfer awards 2022 winners announced. https://www.hipeac.net/news/7008/hipeac-tech-transfer-awards-2022-winners-announced/, 2022.

[14] Scandinavian conference on systems and software safety. https://www.saferesearch.com/events/scandinavian-conference-system-and-software-safety-2022, 2022.

[15] An automated and continuous cybersecurity re-certification solution for iot (i22d). https://eucyberact.org/session/an-automated-and-continuous-cybersecurity-re-certification-solution-for-iot-i22d/, 2023.

[16] Aws Albarghouthi et al. Introduction to neural network verification. *Foundations and Trends® in Programming Languages*, 7(1–2):1–157, 2021.

[17] Fritz Alder, Arseny Kurnikov, Andrew Paverd, and N. Asokan. Migrating SGX enclaves with persistent state. In *48th IFIP International Conference on Dependable Systems and Networks*, DSN '18, pages 195–206. IEEE, 2018.

[18] Ross Anderson, Joey Huchette, Will Ma, Christian Tjandraatmadja, and Juan Pablo Vielma. Strong mixed-integer programming formulations for trained neural networks. *Mathematical Programming*, 183(1):3–39, 2020.

[19] Tim Ansell, Tim Callahan, Jan Gray, Karol Gugala, Maciej Kurc, Guy Lemieux, Charles Papon, Zdenek Prikryl, and Tim Vogt. Draft proposed risc-v composable custom extensions specification. https://github.com/grayresearch/CFU/blob/main/spec/spec.pdf.

[20] Antmicro. Providing synchronized multi-sensor data in renode with resd. https://antmicro.com/blog/2022/12/synchronized-multi-sensor-data-in-renode-with-resd/. Accessed on: Jan. 14, 2023.

[21] Antmicro. Renode. https://renode.io. Accessed on: Jan. 14, 2023.

[22] Antmicro. Renode issue reproduction template. https://github.com/renode/renode-issue-reproduction-template. Accessed on: Jan. 11, 2023.

[23] Antmicro. Renode repository. https://github.com/antmicro/renode-verilator-integration/. Accessed on: Jan. 14, 2023.

[24] Antmicro. Renode-verilator integration examples. https://github.com/antmicro/renode-verilator-integration/. Accessed on: Jan. 14, 2023.

[25] Apache. Apache groovy). https://groovy-lang.org/objectorientation.html, 2022.

[26] Arm. Introducing Arm TrustZone, 2019.

[27] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *12th Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 689–703, Savannah, GA, November 2016. USENIX.

[28] Ivo Babuška. The finite element method with lagrangian multipliers. *Numerische Mathematik*, 20(3):179–192, 1973.

[29] Stanley Bak, Changliu Liu, and Taylor Johnson. The second international verification of neural networks competition (vnn-comp 2021): Summary and results. *arXiv preprint arXiv:2109.00498*, 2021.

[30] M. Baldwin. How to author an azure attestation policy. https://docs.microsoft.com/en-us/azure/attestation/author-sign-policy, Mar 2021.

[31] Daniel Balouek-Thomert, Eduard Gibert Renart, Ali Reza Zamani, Anthony Simonet, and Manish Parashar. Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows. *The International Journal of High Performance Computing Applications*, 33(6):1159–1174, 2019.

[32] Gaurav Banga and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *3rd Symposium on Operating Systems Design and Implementation*, OSDI '99, New Orleans, LA, February 1999. USENIX, USENIX Association.

[33] L. Baresi, D. F. Mendonça, M. Garriga, S. Guinea, and G. Quattrocchi. A unified model for the mobile-edge-cloud continuum. *ACM Trans. Internet Technol.*, 19(2), apr 2019.

[34] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of model checking*, pages 305–343. Springer, 2018.

[35] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya Nori, and Antonio Criminisi. Measuring neural net robustness with constraints. *Advances in neural information processing systems*, 29, 2016.

[36] Alysson Bessani, Joao Sousa, and Eduardo Alchieri. State machine replication for the masses with BFT-SMaRt. In *Proc. of the IEEE/IFIP International Conference on Dependable Systems and Networks – DSN 2014*, June 2014.

[37] Luiz Bittencourt, Roger Immich, Rizos Sakellariou, Nelson Fonseca, Edmundo Madeira, Marilia Curado, Leandro Villas, Luiz DaSilva, Craig Lee, and Omer Rana. The Internet of things, fog and cloud continuum: Integration and challenges. *Internet of Things*, 3:134–155, 2018.

[38] Breno Costa, Joao Bachiega Jr, Leonardo Rebouças de Carvalho, and Aleteia PF Araujo. Orchestration in fog computing: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 55(2):1–34, 2022.

[39] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, page 86, 2016.

[40] Eric C Cyr, Mamikon A Gulian, Ravi G Patel, Mauro Perego, and Nathaniel A Trask. Robust training and initialization of deep neural networks: An adaptive basis viewpoint. In *Mathematical and Scientific Machine Learning*, pages 512–536. PMLR, 2020.

[41] Advanced Micro Devices. Secure Encrypted Virtualization API: Technical preview. Technical Report 55766, Advanced Micro Devices, July 2019.

[42] Tobias Distler, Christopher Bahn, Alysson Bessani, Frank Fischer, and Flavio Junqueira. Extensible distributed coordination. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–16, 2015.

[43] Patrick Doetsch, Michal Kozielski, and Hermann Ney. Fast and robust training of recurrent neural networks for offline handwriting recognition. In *2014 14th international conference on frontiers in handwriting recognition*, pages 279–284. IEEE, 2014.

[44] Ruediger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 269–286. Springer, 2017.

[45] Tolga Ergen and Mert Pilanci. Convex geometry and duality of over-parameterized neural networks. *Journal of machine learning research*, 2021.

[46] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE symposium on security and privacy (SP)*, pages 3–18. IEEE, 2018.

[47] David Goltzsche, Manuel Nieke, Thomas Knauth, and Rüdiger Kapitza. AccTEE: A WebAssembly-based two-way sandbox for trusted resource accounting. In *20th International Middleware Conference*, pages 123–135. ACM, 2019.

[48] Google. Cfu playground. `https://github.com/google/CFU-Playground/`. Accessed on: Jan. 14, 2023.

[49] Google. Confidential computing.

[50] Google. Protocol buffers. `https://developers.google.com/protocol-buffers/`.

[51] Kiryong Ha, Yoshihisa Abe, Thomas Eiszler, Zhuo Chen, Wenlu Hu, Brandon Amos, Rohit Upadhyaya, Padmanabhan Pillai, and Mahadev Satyanarayanan. You can teach elephants to dance: Agile VM handoff for edge computing. In *2nd Symposium on Edge Computing*, SEC '17, New York, NY, USA, 2017. ACM/IEEE, Association for Computing Machinery.

[52] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *38th Conference on Programming Language Design and Implementation*, PLDI '17, page 185–200, New York, NY, USA, 2017. ACM SIGPLAN, Association for Computing Machinery.

[53] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC'10)*, pages 145–158, 2010.

[54] Mahbub Hussain, Jordan J Bird, and Diego R Faria. A study on cnn transfer learning for image classification. In *UK Workshop on computational Intelligence*, pages 191–202. Springer, 2018.

[55] Code Sample Intel. Intel software guard extensions remote attestation end-to-end example, 2018.

[56] Florian Jaeckle, Jingyue Lu, and M Pawan Kumar. Neural network branch-and-bound for neural network verification. *arXiv preprint arXiv:2107.12855*, 2021.

[57] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *USENIX ATC'19*.

[58] Hyuk-Jin Jeong, Chang Hyun Shin, Kwang Yong Shin, Hyeon-Jae Lee, and Soo-Mook Moon. Seamless offloading of web app computations from mobile device to edge clouds via HTML5 web worker migration. In *Symposium on Cloud Computing*, SoCC '19, page 38–49, New York, NY, USA, 2019. ACM, Association for Computing Machinery.

[59] Lv Junyan, Xu Shiguo, and Li Yijie. Application research of embedded database SQLite. In *IFITA'09*.

[60] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International conference on computer aided verification*, pages 97–117. Springer, 2017.

[61] Guy Katz, Derek A Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, et al. The marabou framework for verification and analysis of deep neural networks. In *International Conference on Computer Aided Verification*, pages 443–452. Springer, 2019.

[62] Anum Khurshid. Extended design and first implementation of security, safety and robustness mechanisms and tools. `https://vedliot.eu/deliverable/d5-2-extended-design-and-first-implementation-of-security-safety-and-robustness-mechanisms-and-tools/`, 2021.

[63] Anum Khurshid and Shahid Raza. Autocert: Automated toctou-secure digital certification for iot with combined authentication and assurance. *Computers & Security*, 124:102952, 2023.

[64] Juyong Kim, Yookoon Park, Gunhee Kim, and Sung Ju Hwang. Splitnet: Learning to semantically split deep networks for parameter reduction and model parallelization. In *International Conference on Machine Learning*, pages 1866–1874. PMLR, 2017.

[65] Matthias König, Holger H Hoos, and Jan N van Rijn. Speeding up neural network robustness verification via algorithm configuration and an optimised mixed integer linear programming solver portfolio. *Machine Learning*, pages 1–20, 2022.

[66] Chris Lattner and Vikram S. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, CGO '04, pages 75–86. IEEE, 2004.

[67] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[68] Jianlin Li, Jiangchao Liu, Pengfei Yang, Liqian Chen, Xiaowei Huang, and Lijun Zhang. Analyzing deep neural networks with symbolic propagation: Towards higher precision and faster verification. In *International static analysis symposium*, pages 296–319. Springer, 2019.

[69] Zhiyuan Li, Srinadh Bhojanapalli, Manzil Zaheer, Sashank Reddi, and Sanjiv Kumar. Robust training of neural networks using scale invariant architectures. In *International Conference on Machine Learning*, pages 12656–12684. PMLR, 2022.

[70] Alessio Lomuscio and Lalit Maganti. An approach to reachability analysis for feed-forward relu neural networks. *arXiv preprint arXiv:1706.07351*, 2017.

[71] Emmanuel Maggiori, Yuliya Tarabalka, Guillaume Charpiat, and Pierre Alliez. Convolutional neural networks for large-scale remote-sensing image classification. *IEEE Transactions on geoscience and remote sensing*, 55(2):645–657, 2016.

[72] Zhihong Man, Kevin Lee, Dianhui Wang, Zhenwei Cao, and Chunyan Miao. A new robust training algorithm for a class of single-hidden layer feedforward neural networks. *Neurocomputing*, 74(16):2491–2501, 2011.

[73] Stefano Mariani, Giacomo Cabri, and Franco Zambonelli. Coordination of autonomous vehicles: taxonomy and survey. *ACM Computing Surveys (CSUR)*, 54(1):1–33, 2021.

[74] John Preuß Mattsson, Göran Selander, Shahid Raza, Joel Höglund, and Martin Furuhed. CBOR Encoded X.509 Certificates (C509 Certificates). Internet-Draft draft-ietf-cose-cbor-encoded-cert-05, Internet Engineering Task Force, January 2023. Work in Progress.

[75] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, New York, NY, USA, 2013. ACM, Association for Computing Machinery.

[76] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. Webassembly as a common layer for the cloud-edge continuum. In *Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge*, FRAME '22, page 3–8, New York, NY, USA, 2022. Association for Computing Machinery.

[77] John F. Meyer. Closed-form solutions of performability. *IEEE Transactions on Computers*, 31(07):648–657, 1982.

[78] Microsoft. Azure confidential computing.

[79] Microsoft. Azure Sphere.

[80] Matthew Mirman, Timon Gehr, and Martin Vechev. Differentiable abstract interpretation for provably robust neural networks. In *International Conference on Machine Learning*, pages 3578–3586. PMLR, 2018.

[81] Nir Morgulis, Alexander Kreines, Shachar Mendelowitz, and Yuval Weisglass. Fooling a real car with adversarial traffic signs. *arXiv preprint arXiv:1907.00374*, 2019.

[82] Mozilla. Standardizing WASI: A system interface to run WebAssembly outside the web., March 2019.

[83] Jämes Ménétrey, Christian Göttel, Anum Khurshid, Marcelo Pasin, Pascal Felber, Valerio Schiavoni, and Shahid Raza. Attestation mechanisms for trusted execution environments demystified. In *22nd IFIP International Conference on Distributed Applications and Interoperable Systems*, DAIS '22. Springer, 2022.

[84] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. Twine: An embedded trusted runtime for WebAssembly. In *37th International Conference on Data Engineering*, ICDE '21, pages 205–216. IEEE, 2021.

[85] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. WaTZ: A trusted WebAssembly runtime environment with remote attestation for TrustZone. In *42nd International Conference on Distributed Computing Systems*, ICDCS '22. IEEE, 2022.

[86] Marcelo Pasin. Design and early release of security, safety and robustness mechanisms and tools. https://vedliot.eu/deliverable/deliverable-d51/, 2021.

[87] Luca Pulina and Armando Tacchella. Challenging smt solvers to verify neural networks. *Ai Communications*, 25(2):117–135, 2012.

[88] Automatic Differentiation In Pytorch. Pytorch, 2018.

[89] Benjamin Recht, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. Do cifar-10 classifiers generalize to cifar-10? *arXiv preprint arXiv:1806.00451*, 2018.

[90] RISC-V International. RISC-V Specifications. https://riscv.org/technical/specifications/. Accessed on: Mar. 4, 2022.

[91] Hadi Salman, Greg Yang, Huan Zhang, Cho-Jui Hsieh, and Pengchuan Zhang. A convex relaxation barrier to tight robustness verification of neural networks. *Advances in Neural Information Processing Systems*, 32, 2019.

[92] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *Conference on the Theory and Application of Cryptology*, pages 239–252. Springer, 1989.

[93] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4(3):161–174, 1991.

[94] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[95] Gagandeep Singh, Rupanshu Ganvir, Markus Püschel, and Martin Vechev. Beyond the single neuron convex barrier for neural network certification. *Advances in Neural Information Processing Systems*, 32, 2019.

[96] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. Boosting robustness certification of neural networks. In *International Conference on Learning Representations*, 2018.

[97] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.

[98] Johannes Stallkamp, Marc Schlipsing, Jan Salmen, and Christian Igel. The german traffic sign recognition benchmark: A multi-class classification competition. In *The 2011 International Joint Conference on Neural Networks*, pages 1453–1460, 2011.

[99] Kasia Świrydowicz, Eric Darve, Wesley Jones, Jonathan Maack, Shaked Regev, Michael A Saunders, Stephen J Thomas, and Slaven Peleš. Linear solvers for power grid optimization problems: a review of gpu-accelerated linear solvers. *Parallel Computing*, 111:102870, 2022.

[100] Vincent Tjeng, Kai Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. *arXiv preprint arXiv:1711.07356*, 2017.

[101] German traffic sign recognition benchmark dataset. Gtsrb-benchmark, 2022.

[102] Hoang-Dung Tran, Xiaodong Yang, Diego Manzanas Lopez, Patrick Musau, Luan Viet Nguyen, Weiming Xiang, Stanley Bak, and Taylor T Johnson. Nnv: the neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In *International Conference on Computer Aided Verification*, pages 3–17. Springer, 2020.

[103] Stefanos Tsimenidis. Limitations of deep neural networks: a discussion of g. marcus' critical appraisal of deep learning. *arXiv preprint arXiv:2012.15754*, 2020.

[104] Bill Venners. The Java virtual machine. *Java and the Java virtual machine: definition, verification, validation*, 1998.

[105] Verilator authors. Verilator. `https://www.veripool.org/verilator/`. Accessed on: Jan. 14, 2023.

[106] VMWare. Spring makes java simple.

[107] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient formal safety analysis of neural networks. *Advances in Neural Information Processing Systems*, 31, 2018.

[108] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal security analysis of neural networks using symbolic intervals. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1599–1614, 2018.

[109] Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J Zico Kolter. Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. *Advances in Neural Information Processing Systems*, 34:29909–29921, 2021.

[110] Eric Wong and Zico Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *International Conference on Machine Learning*, pages 5286–5295. PMLR, 2018.

[111] Logan G Wright, Tatsuhiro Onodera, Martin M Stein, Tianyu Wang, Darren T Schachter, Zoey Hu, and Peter L McMahon. Deep physical neural networks trained with backpropagation. *Nature*, 601(7894):549–555, 2022.

[112] Weiming Xiang, Hoang-Dung Tran, and Taylor T Johnson. Reachable set computation and safety verification for neural networks with relu activations. *arXiv preprint arXiv:1712.08163*, 2017.

[113] Weiming Xiang, Hoang-Dung Tran, and Taylor T Johnson. Output reachable set estimation and verification for multilayer neural networks. *IEEE transactions on neural networks and learning systems*, 29(11):5777–5783, 2018.

[114] Kaidi Xu, Huan Zhang, Shiqi Wang, Yihan Wang, Suman Jana, Xue Lin, and Cho-Jui Hsieh. Fast and complete: Enabling complete neural network verification with rapid and massively parallel incomplete verifiers. *arXiv preprint arXiv:2011.13824*, 2020.

[115] Helong Yu, Xianhe Cheng, Chengcheng Chen, Ali Asghar Heidari, Jiawen Liu, Zhennao Cai, and Huiling Chen. Apple leaf disease recognition method with improved residual network. *Multimedia Tools and Applications*, 81(6):7759–7782, 2022.

[116] Alon Zakai. Emscripten: An LLVM-to-JavaScript compiler. In *International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, page 301–312, New York, NY, USA, 2011. ACM, Association for Computing Machinery.

[117] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. Efficient neural network robustness certification with general activation functions. *Advances in neural information processing systems*, 31, 2018.

[118] Xiang Zhang, Xiaocong Chen, Lina Yao, Chang Ge, and Manqing Dong. Deep neural network hyperparameter optimization with orthogonal array tuning. In *International conference on neural information processing*, pages 287–295. Springer, 2019.

[119] Yuyi Zhong, Quang-Trung Ta, Tianzuo Luo, Fanlong Zhang, and Siau-Cheng Khoo. Scalable and modular robustness analysis of deep neural networks. In *Asian Symposium on Programming Languages and Systems*, pages 3–22. Springer, 2021.