ICT-56-2020 — Next Generation Internet of Things

D5.1
Design and early release 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 July 2021
Author	Marcelo Pasin (UNINE)
Contributors	Jämes Ménétrey (UNINE), Anum Khurshid (RI.SE), Alysson Bessani (FC.ID), Piotr Zierhoffer (ANTMICRO), Micha vor dem Berge (CHRISTMANN)
Reviewers	Olof Eriksson (VEONEER), Roland Weiss (SIEMENS), Nils Kucza (UNIBI)

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**
795c783	2021-06-17	Document and structure created.
795c783	2021-06-16	Initial draft for the Table of Contents.
6912031	2021-06-23	Add chapter for Trusted Application Execution.
1078ddb	2021-07-01	Add Renode chapter.
0890854	2021-07-05	Updated work summary report in introduction.
dca58c7	2021-07-05	Add a label to Renode chapter.
7966252	2021-07-07	Added chapter on Trusted Execution Environment.
c60bc79	2021-07-09	Added Trusted Membership Service chapter.
7d56ff1	2021-07-09	Added Introduction, Summary and Conclusion.
9b4f47f	2021-07-11	Updated Renode final remarks.
f6ac4ab	2021-07-19	Added brief reference to D2.2.
ed7e2c2	2021-07-23	Reviewed trusted membership service.
49b1852	2021-07-26	Applied remaining internal reviews.
58447ed	2021-07-29	Final review and cosmetics.

This log reflects actual revision numbers from Git (version control software used).
This PDF was generated Thursday 29th July, 2021, at 23:41.

Table of Contents

List of Figures

1 Executive Summary

The EU-funded VEDLIoT project (Very Efficient Deep Learning IoT system) builds a platform for improving deep learning algorithms using IoT, edge and cloud resources. Such platform is intended to include novel software toolchains and hardware accelerators, with use cases in the industrial, automotive, and smart home areas. In VEDLIoT, part of the platform is developed in Work Package 5, offering tools for security, robustness and safety.

This document reflects the work done in Work Package 5 during the first 9 months of the project. It is split in six chapters: one introduction, four technical chapters, and closing remarks. The introduction describes the Work Package structure (with 5 tasks) and presents the overall progress made in the first 9 months. All tools already developed in Work Package 5 are contextually presented in Chapter 2, the introduction.

A first prototype for trusted execution of WebAssembly applications was developed using Intel SGX. In Chapter 3, we describe this implementation along with the main performance results of our implementation. WebAssembly is a lightweight, portable, binary instruction format that can be efficiently embedded. There are already compilers of most popular languages to generate WebAssembly code. We implemented a runtime environment for executing trusted WebAssembly applications inside Intel SGX enclaves. Our evaluation shows that several pieces of open software can be fully executed inside an SGX enclave via WebAssembly, with low performance overheads.

One of the main goals in Work Package 5 is to provide means for distributed remote attestation service for software components. The starting steps towards building such a service is described in Chapter 4. There, we propose the development of a service with intrusion and fault-tolerant remote attestation verification, membership and configuration.

We also proposed and implemented a novel Trusted Execution Environment (TEE) support for RISC-V processors. In order to test our ideas, we extended VexRiscv, an open-source soft processor built with a pipeline that can easily accomodate plug-in components. Our prototype implements RISC-V Physical Memory Protection, a memory protection unit that provides support for secure processing and enables limiting the physical addresses accessible by software. We provide our implementation as open-source, with extensible documentation for users that intend to use and extend our work. Chapter 5 gives many more details about our implementation and future plans.

Next in order, Chapter 6 presents our first steps in extending Renode, a full-fledged open-source simulator for complex networks of embedded systems. In our initial work we improved Renode with features related to Custom Function Units. Renode's support for CFUs is based on its integration with Verilator, a tool used to simulate peripherals written in Verilog.

Finally, Chapter 7 provides our main achievements in the period covered by the work reported in this deliverable (the first 9 months of the project). We also point out our future directions, with the next steps in each activity of the Work Package.

2 Introduction

The EU-funded VEDLIoT project (Very Efficient Deep Learning IoT system) develops a platform that uses 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, and smart homes. Figure 2.1 displays the overall platform under development by VEDLIoT, with different abstractions for each of the platform components. **Work Package 5 (WP5)** is dedicated to **security, robustness and safety**, and this document is the first output produced.

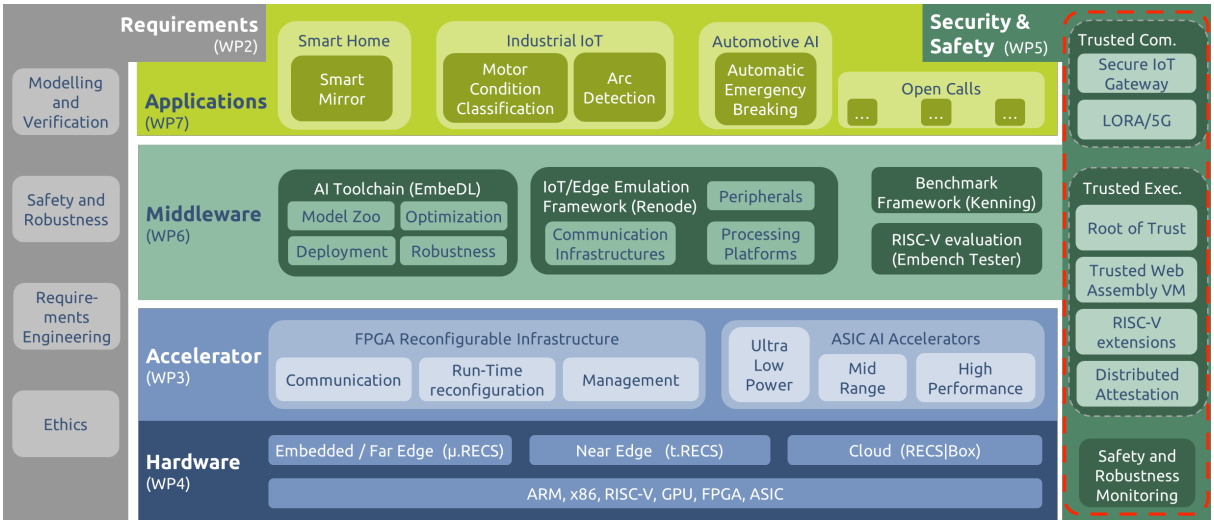

Figure 2.1. VEDLIoT abstracted overview.
WP5 components appear on the right-hand side.

WP5 deals with the development of a toolbox of system-level functionalities towards adding dependability and security support for the execution of edge applications. We leverage hardware features for trusted execution environments combined with well-established dependability techniques to support the tools developed in most other work packages (accelerator design, hardware, deep learning, and use cases). In particular, we focus on developing (1) end-to-end trust through a distributed attestation mechanism, (2) secure execution and communication of critical code on edge devices, (3) monitoring and adaptation mechanisms to increase the ability to control safety and robustness, and (4) tools for simulation and continuous integration in distributed AI systems. WP5 is split into five tasks, for which a small progress summary is given below.

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

As part of VEDLIoT goals, we started new research on means for implementing attestation and trusted execution on embedded systems. Our first progress was made in Task 5.2 (described below), implementing a trusted environment for running distributed applications on Intel SGX. Following that, to support trusted applications in embedded devices, we turned our attention to ARM SoCs, using TrustZone as a

trusted execution environment (TEE), combined with OP-TEE, an open-source, trusted operating system for ARM processors. TrustZone cannot directly support our initial work with Intel SGX because their design and implementation are very different. While SGX allows for normal user-level processes to create and attest protected enclaves, TrustZone splits the operating system into two parts, the normal world and the secure world. Trusted applications can only run in the secure world, and the operation necessary to change context between worlds is rather complex, and cannot be done at user-level.

Having moved to ARM architectures, we started analysing how remote attestation can occur on typical ARM hardware. We based our implementation on NXP's i.MX8M board, which provides all the components to support remote attestations: its ARM processor implements TrustZone and contains a key that is fused by the manufacturer and can be used as root-of-trust, as well as a secure boot mechanism, preventing an attacker from substituting the trusted software. We are now in the process of porting our WebAssembly runtime to work on ARM TrustZone. We intend to leverage WebAssembly's bytecode to provide code measurements, which is a fundamental part of the remote attestation process. The WebAssembly runtime can naturally abstract the functionally-constrained environment of TrustZone as well as the OP-TEE limitation that only supports programs written in C.

Dealing with dynamic system composition as proposed in VEDLIoT poses a number of challenges. One such challenge is to retrieve an up-to-date list of the devices on the system at any given time. Also, the system must ensure that joining devices satisfy high-level security requirements, as defined for critical IoT applications. For example, a given application may only accept specific devices from a specific manufacturer, or may only communicate with a specific version of a specific software component. Attestation is one way of ensuring such guarantees.

As part of the attestation process, we propose the development of a distributed service capable of regulating the participation of devices on IoT applications. This service will implement three main features. It will act as verifier of the device characteristics in the remote attestation procedure, comparing the device-attested measurements with pre-defined application requirements. It will provide coordination primitives to manage membership and failure detection, with a small storage space for configuration data. It will be fault- and intrusion-tolerant, to ensure that neither a failure nor successful attack to the service nodes will undermine the guarantees it brings to IoT applications. Chapter 4 of this deliverable gives details of the design of a **trusted membership service** implementing the features mentioned.

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

WebAssembly is an increasingly popular lightweight binary instruction format, which can be efficiently embedded and sandboxed. Languages like C, C++, Rust, Go, and many others can be compiled into WebAssembly. In the context of VEDLIoT, we implemented Twine, a **WebAssembly trusted runtime** designed to execute unmodified, language-independent applications. As soon as the project started, we leveraged our previous experience with Intel SGX to build the runtime environment without dealing with language-specific, complex APIs. While SGX hardware provides secure execution within the processor, Twine provides a secure, sandboxed software runtime nested

within an SGX enclave, featuring a WebAssembly system interface (WASI) for compatibility with unmodified WebAssembly applications.

The evaluation of Twine, with a large set of general-purpose benchmarks and real-world applications, resulted in a paper entitled *TWINE: An Embedded Trusted Runtime for WebAssembly* [71]. We used Twine to implement a secure, trusted version of SQLite, a well-known full-fledged embeddable database, as we believe that such a trusted database would be a reasonable component to be deployed in edge services. Our evaluation shows that SQLite can be fully executed inside an SGX enclave via WebAssembly and existing system interface, without adding significant performance overheads. We believe that the performance penalties measured are largely compensated by the additional security guarantees and its full compatibility with standard WebAssembly. More information about our implementation can be found in the paper, as well as in Chapter 3 of this deliverable.

As part of our work in providing **hardware-trusted execution** in VEDLIoT, we developed a novel Trusted Execution Environment (TEE) support for VexRiscv, an open-source RISC-V soft processor written in SpinalHDL. VexRiscv takes a uniquely software-inspired approach to hardware description. Virtually every component is a plug-in object connected to one or more stages in the pipeline. It can be scaled to fit a wide range of use cases, from highly constrained bare-metal embedded applications to multi-core Linux with the same code base.

A highly optimised RISC-V Physical Memory Protection (PMP) was developed under this activity in VEDLIoT. The PMP is an optional memory protection unit which provides support for secure processing and enables limiting the physical addresses accessible by software running on a given core. With the PMP, physical memory access privileges (read, write, execute) can be specified for each physical memory region using control registers. PMP is configurable in RISC-V's highest privilege level, machine mode (M), and controls memory access by software running in supervisor (S) and user (U) mode. Small devices may only have M- and U-mode implemented. Machines running Linux, for example, require S-mode, because that is the privilege level which controls the MMU.

Our PMP implementation is part of the latest VexRiscv master branch [76] and the source code is openly available. Furthermore, documentation, links and supplementary materials regarding VexRiscv development are collectively presented on the a webpage [67] to be used on the VEDLIoT project. The documentation is intended for (a) new LiteX users who need a quick start guide, (b) hardware developers who need a detailed explanation of the VexRiscv framework and (c) anyone interested in the current state of RISC-V security extensions. The documentation details instructions on building and running VexRiscv on the Arty A7 FPGA development board. There is also a description and instructions for running RISC-V PMP unit tests. The future of this work is an enhanced development of the RISC-V Memory Protection Unit (MPU) for establishing Trusted Execution States (TES). The idea is to duplicate PMP, but give control to S-mode software. This scheme allows developers to place an embedded OS in S- instead of M-mode without giving up access to the memory protection hardware required to isolate userspace threads. Then, a security monitor (i.e., hypervisor) can run in M-mode, which uses the existing PMP hardware to separate the OS from TEEs on the same device. A preliminary prototype of the MPU is being developed to test setting up a TES for embedded systems with Zephyr OS. More details about the

results obtained with our hardware-trusted execution environment can be found in Chapter 5 of this deliverable.

With the growing usage of IoT devices in homes and industry, new challenges regarding network security arise. Often the security of small IoT devices is not a key concern of vendors, thus presenting attack vectors and possible network infiltration entry points. In VEDLIoT, the **Secure IoT Gateway** aims to shield the IoT network from attacks by separating the IoT network from the local network with VPN technology. By using VPNs, we also encrypt the traffic sent and received by IoT devices. This is accomplished by using hardware (the IoT Bridges) with a VPN client, which are placed between the IoT devices and the local network. A centralised locally placed VPN gateway (local gateway) acts as the VPN server supplier for the IoT bridges in the local network. Besides these two locally placed components, we are also running a cloud based VPN service that allows for secure site-to-site VPN connections. Everything is controlled and monitored via a web application in the cloud. A brief description of the Secure IoT Gateway can be found in the Deliverable 2.2 (Integration of wireless communication into the Secure IoT Gateway).

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

The ability to simulate devices makes the development of embedded systems much easier. As part of the Work Package we are developing improvements to the **Renode Framework** [24], an open-source simulator for complex embedded systems or of networks of such devices. As VEDLIoT focus on IoT and Machine Learning aligns well with Renode capabilities, we are improving the framework with features related to CFUs - Custom Function Units.

CFUs are a mechanism of expanding the capabilities of a CPU with custom instructions, dedicated to accelerating Machine Learning processing. Support for CFUs is based on Renode's integration with Verilator, a tool used to provide Renode with an ability to co-simulate peripherals with models written in Verilog. More information on the standard and the Renode Framework is available in Chapter 6.

During the course of the project so far, we have implemented several improvements to the Verilator integration in Renode. First of all, we improved the existing layers of Verilator integration. We reduced the complexity of integration by migrating from a third-party communication library to POSIX sockets. This also allowed us to provide this integration, originally available on Linux only, for other operating systems, like Windows and macOS.

During this time, the integration was enhanced with support for more communication protocols. At the moment Renode can communicate with verilated peripherals implementing Wishbone, AXI4-Lite and AXI4 interfaces. These additions allowed us to generalise our APIs, paving the way for proper CFU support.

Anticipating the high traffic between Renode and CFU implementations, we decided to implement an additional communication method relying on native calls instead of sockets. This allows not only for faster processing without the overhead induced by the network stack, but also simplifies the communication flow and the debugging process. The progress made in the implementation of a CFU for Renode is described in Chapter 6 of this deliverable.

Task 5.4 Continuous integration workflow
Duration: M18-M36

This task aims at preparing a system that can be used by project participants to test and verify the software for the SoC created as part of Work Package 4. The system will be based on the platform developed in Task 5.3. It will provide continuous integration capabilities, ensuring fitness of the software on each stage of development. This task has not yet started, as it is scheduled for month 18.

Task 5.5 Safety and Robustness
Duration: M7-M36

This task addresses the question on how to argue about robustness and safety in a distributed AI system. Thus, the definition and development of appropriate monitoring and adaptation mechanisms are the main part of this task. From sensor data collection to data communication and processing, many aspects will be considered at several levels, from specification to testing.

The implementation of the monitor components will guarantee timely and secure execution, exploiting the existence of trusted execution environments. Robustness will be verified with respect to certain defined execution envelopes or contexts. Although the task started in month 7, all its work is still incipient. So far, most efforts to kick off this task were made in task 5.1 to help to build the necessary secure execution environment.

3 Trusted Application Execution

Trusted code execution is currently one of the major open challenges for distributed applications such as the ones being developed in the framework of VEDLIoT. Data is a key asset for many companies and the ability to execute code and process data out of their premises is a prerequisite for outsourcing computing tasks, either to large data centres in the cloud or to the edge of the network on thin clients and IoT devices. Trusted execution environments (TEEs) such as Intel SGX [39], ARM TrustZone [77], AMD SME/SEV [22] and RISC-V Keystone [64] gathered much attention lately as they provide hardware support for secure code execution within special hardware constructs that are shielded from the outside world, including the operating system and privileged users. Still, despite the many frameworks and runtime environments that have been developed recently, programming applications for TEEs remains a complex task. Developers must generally use custom tools and APIs, and they are restricted to a few supported programming languages.

In this chapter, we describe a trusted runtime that supports execution of unmodified applications compiled to WebAssembly (Wasm) [48], a portable binary-code format for executable programs originally designed for efficient execution within Web browsers. Among its many benefits, Wasm is optimised for speed, can be efficiently embedded, sandboxed, and is considered secure [65].

LLVM is a compiler toolchain, one of the most popular compilation infrastructure nowadays. It natively supports Wasm as a standard compilation target. Thanks to that, programs developed in languages such as C, C++, Rust, Swift, Go, C#, D, Delphi, Fortran, Haskell, Julia, Objective-C, and many others, can already be used as input to produce Wasm executables. Therefore, by supporting Wasm, one can provide a generic runtime environment without resorting to language-specific, dedicated APIs.

Our approach completely abstracts the application from the underlying hardware and operating system (OS). We developed our runtime as a first step in Work Package 5, in order to be able to seamless execute trusted applications in the whole VEDLIoT environment, be in IoT devices, in the edge, or in third-party clouds.

In the sections that follow, we present our first implementation of a lightweight embeddable Wasm virtual machine running in a TEE, named Twine (trusted Wasm in enclave). Figure 3.1 depicts its typical workflow. It acts as an adaptation layer between the application and the underlying TEE, the OS and hardware. Twine includes a comprehensive WASI (WebAssembly system interface) layer to allow for native execution of legacy Wasm applications, without recompilation.

We currently support Intel SGX enclaves as TEEs: Twine dynamically translates WASI

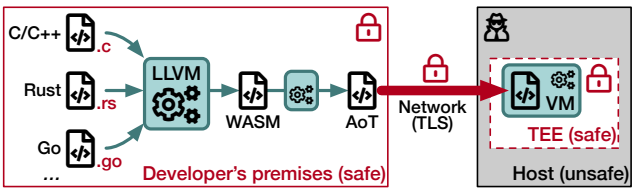

Figure 3.1. Overview of Twine*'s workflow.*

operations into equivalent native OS calls or to functions from secure libraries purposely built for SGX. In particular, Twine maps file operations to Intel protected file system [54], and persisted data is transparently encrypted and never accessible in plaintext from outside an enclave. Whereas a TEE provides a secure hardware execution runtime in the processor, Twine provides a secure software runtime (sandbox) nested within the TEE, with a WASI interface for compatibility with legacy Wasm, abstracting the underlying environment from the application.

We evaluated Twine with several micro- and macro-benchmarks, as well as a full SQLite implementation. We compared its performances against existing software packages, with and without secure operations inside a TEE. Our results reveal that Twine performs on par with systems providing similar security guarantees. We also observed non-negligible performance overheads due to execution within the TEE under some workloads. We believe this penalty is largely compensated by the additional security guarantees and full compatibility with Wasm code thanks to the WASI interface.

The contributions presented in this chapter were extracted from a paper published in the 37th IEEE International Conference on Data Engineering [69]. Our work represents the first real open-source implementation of a general-purpose Wasm runtime environment within SGX enclaves with full support for encrypted file system operations. In addition to this chapter, the paper presents related work, and an extensive evaluation of our implementation, offering a good understanding of its performance costs and associated bottlenecks. As a subordinate effect, we proposed an improved implementation for Intel's protected file system, showing the derived performance improvements in the paper as well. This chapter is split into three sections. We provide some background on Intel SGX and WebAssembly in Section 3.1. The design and implementation details of Twine are described in Section 3.2. We then summarise the evaluation of our prototype in Section 3.3.

3.1 Background for Twine

This section provides background information on Intel SGX in (Section 3.1.1) and the Wasm ecosystem (Section 3.1.2) to help understand the architecture and design of Twine.

3.1.1 Intel SGX

Software Guard Extensions (SGX) [39] are a set of processor instructions found in modern Intel processors [53] that allow programmers to create encrypted regions of memory, called *enclaves*. Enclave memory content is automatically encrypted and decrypted when read and written by instructions running inside the enclave itself. Enclave encryption keys are kept inside the processor and no instruction has access to the keys, not even when running with high hardware privilege levels, as OSs and virtual machine managers do. The memory inside an enclave is protected from any unauthorised access, even from machine administrators with physical access.

Enclave memory access is accelerated by using a large cache memory, called EPC (enclave page cache). EPC size is limited, with the latest CPUs offering up to 256 MiB. The processor keeps unencrypted copies of all enclave pages in EPC, and paging is used when the EPC is full. The hardware also maintains cryptographic hashes for all enclave pages in EPC, in such a way that a modification from outside an enclave can be automatically detected. The EPC helps reduce access time to encrypted memory

but also limits the number of pages concurrently available. Swapping degrades performance and enclaved applications should strive to avoid it [75].

Instructions inside enclaves can access data outside the enclave, but calling instructions outside requires a special *out call* instruction (`OCALL`). Upon an `OCALL`, the CPU exits the protected enclave to execute code on the outside. Conversely, there is an *enclave call* (`ECALL`) instruction to call code inside an enclave. `OCALL` and `ECALL` instructions are slow because switching the context from inside to outside an enclave is costly (up to 13'100 CPU cycles in latest server-grade processors). It has been shown that enclaved applications can avoid such calls to reduce performance loss [84].

In order to build composed software using enclaves, one must have a method to establish trust. For example, a client must know if it can trust a given server and vice versa. Intel SGX offers a remote attestation mechanism to prove that an enclave can be trusted. Each processor has a secret key fused in its die, used to derive many other keys. One of the derived keys is used to build enclave attestations, calculated as a signature of the whole content of an enclave at its creation. An external attestation service confirms that a given enclave runs a particular piece of code on a genuine Intel SGX processor.

3.1.2 WebAssembly

WebAssembly (Wasm) is a W3C recommended open standard for a portable and executable binary code format. It was originally designed to improve the performance of applications embedded in Web browsers, similar to the now-deprecated Microsoft ActiveX, and directly superseding `asm.js` [41]. Since then, its support was extended to standalone environments (*i.e.*, outside browsers). Full application execution, especially in standalone environments, requires access to OS services, *e.g.*, process and memory management or I/O, typically available via common system calls (for instance, exposed by a POSIX interface). Hence, the interaction of Wasm with the underlying OS is standardised through a specific API called WebAssembly system interface (WASI) [70]. This interface allows for several implementations suited to different OSs and incorporating several non-functional abstractions, including virtualisation, sandboxing, access control, *etc.* In the latest specifications, the WASI interface consists of 45 functions covering various capabilities: access to process arguments and environment variables, file system interaction, events polling, process management, random number generation, socket interaction and time retrieval.

There are currently several options to produce Wasm binaries. *Emscripten* [91] and *Binaryen* [1] can compile C/C++ into Wasm binaries with support for POSIX OS calls for standalone applications. These tools can convert and execute legacy applications into their Wasm representation. However, the conversion is only possible by requesting the Wasm runtime to expose functions that are generally bound to a specific OS, *i.e.*, not a standard nor a public interface. Wasm applications become tightly coupled to a given OS, defeating one of its main purposes, *i.e.*, portability. WASI solves the issue with a standard and lightweight interface that Wasm runtimes can comply with to support a large variety of interactions abstracted from the OS. The introduction of this abstract layer limits the coupling of Wasm applications to just WASI. As a result, Wasm applications using WASI are system-agnostic and can run on any compliant OS or browser.

LLVM [63] is a compilation toolchain for several different programming languages.

Table 3.1. Comparison of Wasm runtimes.

Wasm runtime	Language	Embeddable	Interpreter	JIT	AoT
Wasmtime [8]	Rust	✓*†	✗	✓	✓
Wasmer [7]	Rust	✓*†	✗	✓	✓
Lucet [4]	Rust	✓*†	✗	✗	✓
WAVM [9]	C++	✓*	✗	✓	✓
Wasm3 [6]	C	✓*	✓	✗	✗
WAMR [10]	C	✓	✓	✓	✓

*Requires adaptation for SGX. †Intel does not support Rust for enclave development.

The compilation is split into front- and back-end modules. The connection between them uses the LLVM intermediate representation code. LLVM supports several front-end modules for various languages and, similarly, many back-ends to generate different binary formats. Since v8.0, LLVM officially supports and can generate Wasm code with WASI. All compiler front-ends using recent LLVM versions can consequently generate Wasm code. Note that, while Wasm represents an abstract machine, WASI represents its abstract OS, *i.e.*, a standard interface to run Wasm applications outside of a browser. Due to this tight dependency, tools generating Wasm code must be adapted to couple the Wasm code generated with WASI calls.

The execution of Wasm code must be handled by a dedicated runtime, able to execute the instructions and implementing WASI calls. We discuss below the advantages and drawbacks of existing Wasm runtimes and explain why Twine settled for one of them. Table 3.1 summarises the main properties of the Wasm runtimes considered. We compare them in terms of execution modes, implementation language and whether they can be embedded into a TEE, such as SGX enclaves.

Wasmtime [8] is a Rust-based standalone runtime. It uses Cranelift [3], a low-level retargetable just-in-time (JIT) compiler with similarities to LLVM. Wasmtime can be used by various programming languages thanks to the wrappers available with the runtime. Embedding a JIT compiler inside an SGX enclave, despite its potential performance benefits, increases the trusted computing base by a large factor. Moreover, Wasmtime and Cranelift are implemented in Rust: while tools exist to support Rust binaries in SGX enclaves [88], we opted in Twine for the well-supported standard Intel toolchain.

Lucet [4] is a native Wasm compiler and runtime also implemented in Rust. It is designed to safely execute untrusted WebAssembly programs embedded in third-party applications. It supports ahead-of-time (AoT) compilation of Wasm applications using Cranelift. While the runtime is not coupled to Cranelift as Wasmtime, Lucet presents similar integration challenges (Rust, large TCB).

Wasmer [7] is a Rust-based Wasm runtime for lightweight and portable containers based on Wasm. It allows for JIT and AoT compilations with multiple back-ends, including LLVM and Cranelift. It supports the two prominent application binary interfaces (ABI): WASI and Emscripten. We turned away from Wasmer for the same reason

as the previous alternatives.

WAVM [9] is a Wasm virtual machine written in C++. It supports both WASI and Emscripten ABIs and offers various extensions, such as 128-bit SIMD, thread management and exception handling. While implemented in C++, hence with native support for enclave development, its tight coupling with LLVM makes it difficult (if possible at all) to embed it inside SGX.

Wasm3 [6] is a micro-interpreter for Wasm, optimised for size, able to execute in restricted memory environments and to provide fast startup latency. It was designed for constrained edge devices with very limited resources (*e.g.*, Arduino and Particle). Having a reduced set of dependencies and small code base, it can easily fit within SGX enclaves. However, it only supports interpreted code and, hence, provides limited performance for executing Wasm binaries.

The WebAssembly micro runtime (WAMR) [10] is a standalone Wasm runtime supported by the *bytecode alliance* open source community. This runtime supports two interpreted execution modes, one slower and one faster, the former using less memory than the other. It also supports two binary execution modes, AoT and JIT, both using LLVM. WAMR is implemented in C with a small footprint (runtime binary size of 50 KiB for AoT, 85 KiB for interpreter) and very few external dependencies, which is ideal for small embedded devices with limited resources. WAMR can be linked with SGX enclaves out of the box, which significantly simplifies the integration of Wasm and SGX. We, therefore, opted for WAMR as underlying runtime for Twine, as detailed in Section 3.2.

3.2 Trusted runtime for WebAssembly

Twine is an execution environment suited for running Wasm applications inside TEEs. It is built with two main blocks: a Wasm runtime and a WASI interface (see Figure 3.2). The Wasm runtime runs entirely inside the TEE, and WASI works as a bridge between trusted and untrusted environments, abstracting the machinery dedicated to communicate with the underlying OS. Thus, WASI is the equivalent to the traditional SGX adaptation layer comprised of the `OCALL`s. The main advantage of relying on WASI is that it brings a triple abstraction. Firstly, the programming language can be freely chosen by the developers, provided it can be compiled with LLVM or another compiler that supports Wasm and WASI as a compilation target. This lifts the restrictions imposed by SGX, typically forcing enclaved applications to be written in C/C++. Secondly, the TEE is abstracted away from the applications. Applications can be safely executed as long as the TEE is able to interpret or execute Wasm (supported by WASI), opening the door to other TEE technologies. Finally, WASI is system-agnostic, as long as the OS can provide an equivalent of the API required by WASI. Since WASI mimics

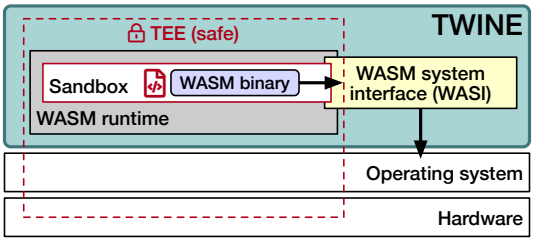

Figure 3.2. Overall Twine *architecture.*

the system calls of POSIX systems, many Unix variants can implement it.

On top of its portability benefits, WASI focuses on security by sandboxing. Regular applications usually call the OS through a standard interface (*e.g.*, POSIX). WASI adds a thin layer of control between Wasm OS calls and the actual OS interface. As a result, the runtime environment can limit what Wasm can do on a program-by-program basis, preventing Wasm code from using the full rights of the user running the process. For instance, a WASI implementation can restrict the application to a subtree of the file system, similar to the capabilities offered by *chroot*.

The combination of the enclave and sandbox capabilities of SGX and WASI, respectively, ends up in a two-way sandboxing system partially inspired by MiniBox [66]. The system, which is considered untrusted in the threat model of SGX, cannot compromise the integrity of the enclave code nor the confidentiality of the data stored in its memory. Likewise, Wasm applications, considered untrusted from the system's owner standpoint, cannot interact directly with the OS unless WASI explicitly grants permission in the Wasm runtime. Therefore, the Wasm application providers and the hosting platform can agree on the trust guarantees given by SGX and those of a reference Twine enclave with strong peer-reviewed sandboxing capabilities, making WASI a mutually trusted demilitarised zone.

3.2.1 Threat model

Twine leverages the protection of TEEs to offer a trusted environment for running Wasm applications. Many guarantees offered by Twine are inherited from the underlying TEE, which in our implementation is Intel SGX. Note that a different TEE may not withstand the same level of threats.

Assumptions. We assume that no physical attack is possible against the computer hardware. The TEE offers the level of protection as specified, and standard cryptography cannot be subverted. Application and OS codes present no vulnerabilities by implementation mistake nor careless design.

SGX enclaves. Code and data inside enclaves are considered as trusted, and nothing from outside can be considered trusted. The non-enclaved part of a process, the OS and any hypervisor are thus potentially hostile. The memory inside of an enclave can only be read in encrypted form from the outside. Writing the memory enclave from the outside causes the enclave to be terminated. Side-channel or denial-of-service attacks may exist, and applications running inside enclaves must be written to be resistant to them. While we consider side-channel attacks out of scope, mitigations exist [32, 74].

Operating system. The OS follows an honest-but-curious model. In principle, the OS follows its specification and poses no threat to user processes. A compromised OS may arbitrarily respond to enclave calls, causing its malfunction; enclaves should be carefully crafted to ignore abnormal responses or even abandon execution in such cases.

3.2.2 WASI

As presented in Section 3.1, we considered Wasmtime, Wasmer, Lucet, WAVM, Wasm3 and WAMR as runtime candidates for implementing Twine. Wasmtime, Wasmer, Lucet and WAVM may be executed inside SGX enclaves, but require substantial adaptations

to comply with the SGX enclaves' restrictions. Moreover, some of these runtime environments (except WAVM and Wasm3) are written in Rust and require additional effort to use as a trusted runtime, since Intel does not support this programming language for enclave development. Wasm3, on the other hand, is small but only offers an interpreter, this being an inadequate constraint for running standalone applications. Finally, WAMR is also small, has few dependencies, and can link to binary code (albeit generated ahead of time, that is, no JIT). We chose to use WAMR and replace its WASI interface, as explained below, in such a way that we can abstract the enclave constraints while implementing systems calls.

WASI is the interface through which Wasm applications communicate with the outside world, similar to POSIX's capabilities for regular native programs. The development of TEE enabled applications requires to deal with crossing the boundary between trusted and untrusted environments, materialised with `ECALL`s and `OCALL`s in Intel SGX. We believe that leveraging WASI as the communication layer meets the purpose of Wasm, where the implementation is abstracted away for the application itself. As a result, the applications compiled in Wasm with WASI support do not require any modification to be executed inside a TEE.

The toolkit of WAMR provides an ahead-of-time compiler, enabling to compile Wasm applications into their native representation using LLVM before they reach Twine's enclave. As such, Twine does not contain a Wasm interpreter and can only execute ahead-of-time compiled applications. The main advantage of this choice is that native code execution is faster than code interpretation, which is critical to be competitive with the other secure TEE solutions [25, 38]. Moreover, the Wasm runtime has a smaller memory footprint than the code interpreter, which are essential factors in the context of SGX and cloud/edge computing. The option of embedding a JIT compiler was not considered, as bringing LLVM machinery in an enclave requires modifying it to the restrictions of SGX.

Unlike Twine, Intel SGX only guarantees the integrity of the enclave binary and not the confidentiality. Integrity is verified with a signature in the code, but the code itself must be in plaintext to be loaded into an enclave memory. Twine is able to offer the confidentiality of Wasm applications because the Wasm code is supplied using a secure channel after the enclave has been started. When the Wasm code is received, it is mapped into a secure memory area called *reserved memory* [55]. That memory area enables one to load arbitrary executable code and manage the pages' permissions as if they were outside the enclave. Therefore, Wasm applications never leave the secure memory of the enclave.

3.2.3 WASI implementation details

By the time Twine was developed, WAMR already included a WASI implementation that relies heavily on POSIX calls. POSIX is not available inside SGX enclaves, so the implementation of WASI written by the authors of WAMR needs to frequently cross the trusted boundary of the enclave and plainly routes most of the WASI functions to their POSIX equivalent using `OCALL`s. While this approach enables to run any Wasm applications that comply with WASI inside an enclave, this does not bring additional security regarding the data that transits through POSIX.

We designed Twine to implement a different WASI interface for WAMR, that is more tailored to the specific TEE used (namely SGX). We estimated that plainly forward-

ing WASI calls to outside the enclave was not the best option. First, for performance reasons: most WASI calls would simply be translated to OCALLs. Second, we wanted to leverage trusted implementations when available, as for instance Intel protected file system (IPFS), described below (Section 3.2.4). Therefore, we refactored WAMR's WASI implementation to keep its sandboxing enforcement, and we split the remaining into two distinct layers, one for specific implementations, when available, and one for generic calls. Generic calls are handled by calling a POSIX-like library outside the enclave while providing additional security measures and sanity checks. Such calls are only implemented when no trusted compatible implementation exists. For instance, time retrieval is not supported by Intel SGX. Hence, Twine's POSIX layer leaves the enclave to fetch monotonic time while ensuring that the returned values are always greater than the previous ones. If a trusted implementation exists (as the many in Intel SDK), we use it to handle its corresponding WASI call. Sometimes a trusted implementation needs to call outside the enclave, but they often offer more guarantees than merely calling the OS. One notable example is the protected file system, described below. Finally, Twine includes a compilation flag to globally disable the untrusted POSIX implementation in the enclave, which is useful when developers require a strict and restricted environment or assess how their applications rely on external resources. In particular, the interface may expose states from the TEE to the outside by leaking sensitive data in host calls, *e.g.*, usage patterns and arguments, despite the returned values being checked once retrieved in the enclave.

Memory management has an important impact on the performance of the code executed in an enclave (see Section 3.3). WAMR provides three modes to manage the memory for Wasm applications: *(1)* the default memory allocator of the system, *(2)* a custom memory allocator, and *(3)* a buffer of memory. Twine uses the latter option since we measured that an application that heavily relies on the memory allocator of SGX to enlarge existing buffers performs poorly. For instance, SQLite microbenchmarks in Section 3.3.1, which requires to extend its internal buffer for every new record being added. Before using a preallocated buffer for SQLite, we noticed the complexity of the SGX memory allocator to be above linear.

In its current implementation, Twine requires to expose a single ECALL to supply the Wasm application as an argument. This function starts the Wasm runtime and executes the start routine of the Wasm application, as defined by WASI ABI specifications [11]. Future versions of Twine would only receive the Wasm applications from trusted endpoints supplied by the applications providers, as shown in Figure 3.1. The endpoint may either be hard-coded into the enclave code, and therefore part of the SGX measurement mechanism that prevents binary tampering, or provided in a manifest file with the enclave. The endpoint can verify that the code running in the enclave is trusted using SGX's remote attestation. As a result, Twine will provide both data and code confidentiality and integrity by relying on SGX capabilities, as well as a secure channel of communication between the enclave and the trusted application provider. While the enclave must rely on the OS for network communication, the trusted code can use cryptographic techniques (*e.g.*, elliptic-curve Diffie-Hellman) to create a channel that cannot be eavesdropped on.

3.2.4 Intel Protected File System (IPFS)

To validate the abstraction offered by WASI, we implemented a subset of the WASI calls (*i.e.*, those related to file system interaction) using the Intel protected file sys-

tem [54] (IPFS). Part of Intel SGX SDK, it mimics POSIX standard functions for file input/output. The architecture of IPFS is split in two: *(1)* the trusted library, running in the enclave that offers a POSIX-like API for file management, and *(2)* the untrusted library, an adapter layer to interact with the POSIX functions outside of the enclave, that actually read and write on the file system. Upon a `write`, content is encrypted seamlessly by the trusted library, before being written on the media storage from the untrusted library. Conversely, content is verified for integrity by the trusted enclave during reading operations.

IPFS uses AES-GCM for authenticated encryption, leveraging the CPU's native hardware acceleration. An encrypted file is structured as a Merkle tree with nodes of a fixed size of 4 KiB. Each node contains the encryption key and tag for its children nodes. Thus, IPFS iteratively decrypts parts of the tree as the program running in the enclave requests data [85]. This mechanism ensures the confidentiality and the integrity of the data stored at rest on the untrusted file system. While the enclave is running, the confidentiality and the integrity of the data are also guaranteed by SGX's memory shielding.

IPFS has several limitations, which are considered to be outside of its security objectives by Intel. Since the files are saved in the regular file system, there is no protection against malicious file deletion and swapping. Consequently, this technology lacks protection against: *(1)* rollback attacks, IPFS cannot detect whether the latest version of the file is opened or has been swapped by an older version, and *(2)* side-channel attacks, IPFS leak file usage patterns, and various metadata such as the file size (up to 4 KiB granularity), access time and file name. We note how Obliviate [23], a file system for SGX, partially mitigates such attacks.

IPFS provides convenient support to automatically create keys for encrypting files, derived from the enclave signature and the processor's (secret) keys. While automatic key generation seems straightforward, a key generated by a specific enclave in a given processor cannot be regenerated elsewhere. IPFS circumvents this limitation with a non-standard file open function, where the caller passes the key as a parameter. Our prototype relies on automatic generation as an alternative to a trustworthy secret sharing service [47]. We leave as future work to extend the SGX-enabled WASI layer to support encrypted communication through sockets.

In conclusion, files persisted by Twine are seen as ciphertext outside of the enclaves, while transparently decrypted and integrity-checked before being handled by a Wasm application.

3.3 Evaluation

The evaluation results presented in the paper [69] were intended as an answer to the following questions:

- What is the performance overheads of using the runtime WAMR in SGX, compared to native applications?
- Can a database engine realistically be compiled into Wasm and executed in a TEE, while preserving acceptable performances?
- How do the database input and output operations behave when the EPC size limit is reached?

- What are the primitives that generate most of the performance overheads while executing database queries? Can we improve them?

In this section we summarise Twine's evaluation results, and we refer to the paper for further details.

3.3.1 PolyBench/C micro-benchmarks

As a micro-benchmark experiment, we leveraged PolyBench/C [78], a group of CPU-bound programs commonly used to evaluate the performance of Wasm execution environments [56, 45]. Figure 3.3 shows the results for 30 PolyBench/C (v4.2.1-beta) tests, compiled as native (plain x86-64 binaries) and Wasm compiled ahead-of-time. Results are given for the native execution, those using WAMR for Wasm, and finally using Twine for Wasm in SGX. Results are normalised against the native run time.

We can split the PolyBench/C test results in 5 groups, based on the proportion between the execution modes (native, WAMR and Twine): *(1)* similar execution time (`doitgen` and `seidel-2d`); *(2)* WAMR results similar to each other, but overall slower than to native (`2mm`, `3cmm` and `durbin`); *(3)* Twine is slower than WAMR and native (`deriche`, `gemver` and `lu`); *(4)* execution times vary significantly between each variant (`atax`, `gemm` and `jacobi-2d`); *(5)* WAMR is faster than its native counterpart.

Wasm applications are usually slower than native ones due to several reasons: increased register pressure, more branch statements, increased code size, *etc.* Following previous work [56], we investigated `deriche` and `gramschmidt` using Linux's performance counters, as both produced better results with Wasm (averages over 3 distinct executions). Our analysis reports 58,002,746 L1 cache misses for native `deriche` and 57,384,578 for its Wasm counterpart. Similarly `gramschmidt` produces 3,679,222,800 and 3,673,458,022 for native and Wasm L1 cache misses. These results confirm that these two Wasm programs produce slightly fewer L1 caching misses (1.1% and 0.2%).

3.3.2 SQLite macro-benchmarks

SQLite [58] is a widely-used full-fledged embeddable database. It is perfectly suited for SGX, thanks to its portability and compact size. For this reason, we thoroughly evaluated it as a showcase for performance-intensive operations and file system interactions. SQLite requires many specific OS functions that are missing from the WASI specifications, due to standardisation and portability concerns in Wasm. Therefore, we relied on SQLite's virtual file system (VFS), and accesses to the file system are translated into the WASI API. Our modified virtual file system implements the minimal requirements to make SQLite process and persist data, reducing the POSIX functions to be supported by Twine WASI layer.

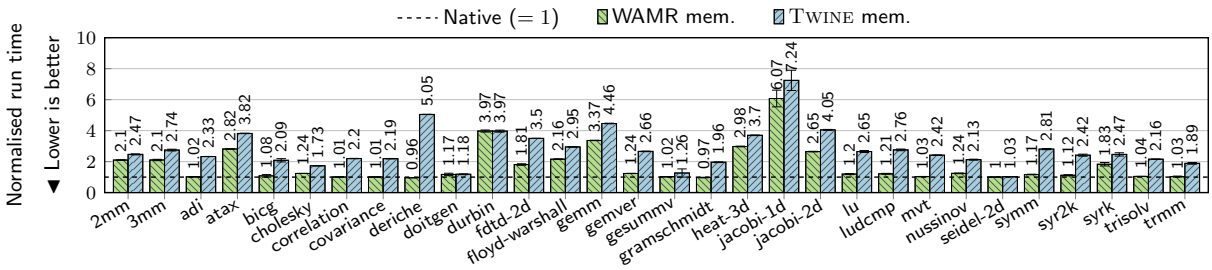

Figure 3.3. Performance of PolyBench/C benchmarks, normalised to the native speed.

Since memory allocation in SGX enclaves is expensive (in some tests, it took up to 45% of the CPU time to allocate it while inserting records in the database), memory preallocation greatly optimises performance when the database size is known in advance.

First, we used SQLite's own performance test program, `Speedtest1` [5], running 29 out of the available 32 tests, covering a large spectrum of scenarios (we excluded 3 experiments because of issues with SQLite VFS). Each `Speedtest1` experiment targets a single aspect of the database, *e.g.*, selection using multiple joints, the update of indexed records, *etc.* Tests are composed of an arbitrary number of SQL queries, potentially executed multiple times depending on the load to generate. Figure 3.4 shows our results, normalised against the native execution. We include results for in-memory configurations as well as for a persisted database, where WASI is used.

While we provide additional details in the paper, we observed across all tests that the WAMR's slowdown relative to native on average is 4.1× and 3.7× for in-memory and in-file database respectively. Twine's average slowdown relative to WAMR is 1.7× and 1.9× for in-memory and in-file database.

Experiments 100–120, 180–190, 230, 240, 270–300, 400 and 500 update the database (*e.g.*, creating tables, inserting, updating and deleting records). They share a similar pattern of performance penalty according to the variants. Experiments 130, 140 and 145–170 indicate the same performance for in-memory and persistent databases: since they only execute read operations, they act on the page cache, with no file system interaction. Using SGX with a persistent database adds a considerable overhead under certain circumstances. In particular, experiments 410 and 510, which overflow the page cache and randomly read records, cause additional latency due to the file system interaction, exacerbated by enclave `OCALL`s and encryption, up to 12.4× and 22.1× for Twine and SGX-LKL respectively compared to the equivalent queries using an in-memory database. Interestingly, experiments 142 (multiple `SELECT` with `ORDER BY`, non-indexed) and 520 (multiple `SELECT DISTINCT`) show faster results using a persistent database on-file for all the execution modes. Test 210 is I/O intensive: it alters the database schema and, consequently, all the records. Similarly, experiment 260 issues a wide-range of `SELECT` to compute a sum, explaining the high execution time across all execution modes, with a small overhead for SGX variants. In addition, test 250 is highly I/O intensive with a persisted database, because it updates every record of a table, requiring to reencrypt most of the database file.

Finally, 990 is a particular case of database housekeeping. It gathers statistics about tables and indices, storing the collected information in internal tables of the database where the query optimiser can access the information and use it to help make better query planning choices. The longer execution time of Twine and SGX-LKL with a per-

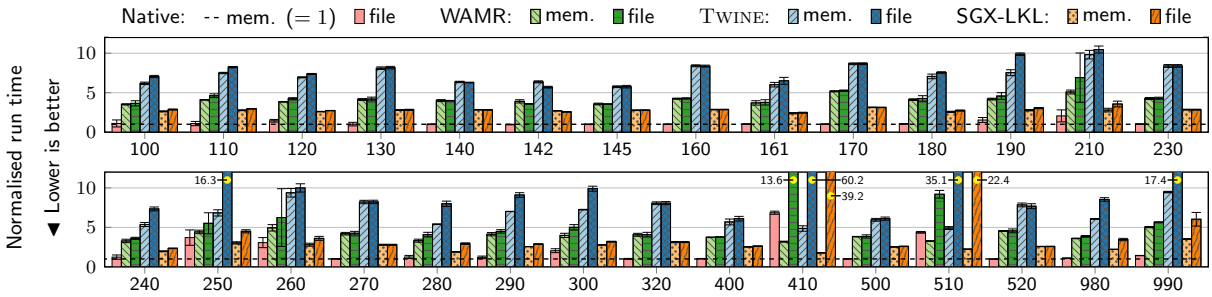

Figure 3.4. Relative performance of SQLite `Speedtest1` benchmarks.

sistent database is explained by the added complexity of I/O from the enclave.

One of the most important costs impacting the performace of Twine is the use of Intel's IPFS for file I/O. In the full paper, we detail out a number of modifications to IPFS in order to improve its performance. Compared to Intel's version, insertion achieves a 1.5× speedup and 2.5× for sequential reading. Finally, for random reading, we achieved a 4.1× speedup.

3.4 Closing remarks

The lack of trust when outsourcing computation to remote parties is a major impediment to the adoption of distributed architectures for VEDLIoT's sensitive applications. Whereas this problem has been extensively studied in the context of cloud computing across large data centres, it has been only scarcely addressed for decentralised and resource-constrained environments as found in IoT or edge computing.

We proposed an approach for executing unmodified programs in WebAssembly (Wasm) within lightweight trusted execution environments that can be straightforwardly deployed across client and edge computers. Wasm is a target binary format for applications written in languages supported by LLVM, such as C, C++, Rust, Fortran, Haskell, *etc*. Twine is our trusted runtime with support for execution of unmodified Wasm binaries within SGX enclaves. We provide an adaptation layer between the standard Wasm system interface (WASI) used by the applications and the underlying OS, dynamically translating the WASI operations into equivalent native system calls or functions from secure libraries purposely built for SGX enclaves. Our in-depth evaluation shows performance on par with other state-of-the-art approaches while offering strong security guarantees and full compatibility with standard Wasm applications.

Twine has been made freely available as open-source software.

4 Trusted Membership Service

An important challenge in IoT systems is how to deal with the dynamism on the system composition (i.e., devices joining and leaving the system, crashes and recoveries, etc.) Besides the fundamental problem of getting an up-to-date list of the devices on the system at any given time, there is also the problem of ensuring joining devices satisfy the high-level security requirements defined for critical IoT applications. For example, a certain application can only accept devices from a pre-defined manufacturer and if they are running the last version of a given operating system.

To solve these type of problems, we propose the development of a distributed service capable of regulating the participation of devices on IoT applications. This service will implement three main features:

1. It must act as verifier of the device characteristics in remote attestation procedures [31], comparing the device-attested measurements with pre-defined application requirements.
2. It must provide coordination primitives [40], similar to the ones provided by Zookeeper [52], for supporting membership management, failure detection, and limited storage (mostly for configuration data).
3. It must be fault- and intrusion-tolerant [42], just like a permissioned blockchain, to ensure that a successful attack to the service nodes will not undermine the security guarantees it brings to IoT applications.

In summary, we aim to design a *Trusted Membership Service* (TMS) capable of providing a reliable source of information about which devices are available to the application, and to which extent these devices are running up-to-date software, among other properties. The objective is to significantly increase the trust between interacting IoT devices by regulating their participations in applications based on pre-defined policies.

In this chapter we outline the basic characteristics of TMS and present our initial design for it. We start by describing an application scenario of TMS (Section 4.1), then we proceed to discuss some background and existing techniques/tools that are being employed in the system (Section 4.2), and finally we discuss the main architecture of the system (Section 4.3). We conclude the chapter with a description of the status of the service and an outline of our next steps.

4.1 Application Scenario

The VEDLIoT TMS consider a set of IoT devices running one or more applications sharing the same environment, interconnected with edge servers and a cloud infrastructure, as described in Figure 4.1.

In this scenario, the complexity of the manual configuration of devices and the assurance of correct software updates (to remove vulnerabilities) are the main issues that need to be addressed. More specifically, we consider the following two specific problems:

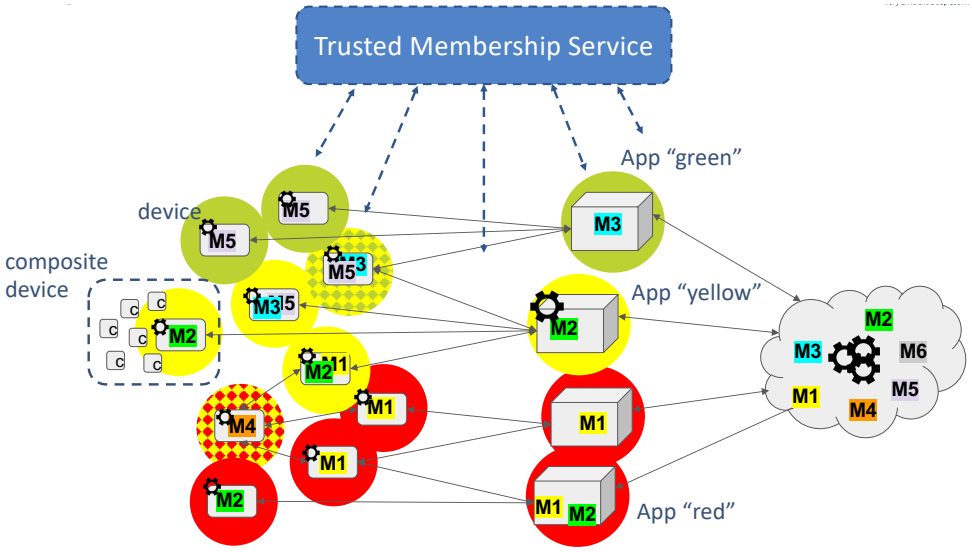

Figure 4.1. Diverse IoT applications managed by VEDLIoT Trusted Membership Service (TMS). Each color represents a different application and each device can run one or more deep learning model (e.g., M1, M2), obtained from edge services or the cloud.

- *Devices and servers need to be organized in application-specific topologies.* Some questions and challenges we need to solve are: How to make a device trust another? How to ensure all nodes are running correct and up-to-date software? With minimal pre-configuration, how to establish secrets among nodes (cryptographic keys, for example)?
- *Devices and servers need configuration and coordination.* The questions then are: How to store and disseminate common application configuration? How to obtain the list of nodes that are alive on an IoT application? How to make nodes agree about something (e.g., a leader, a data dissemination/aggregation tree topology)?

We propose to solve these problems by building a Trusted Membership Service (see Figure 4.1) that will regulate and help the interactions between IoT devices in untrusted environments. This service will leverage recent advances in trusted computing that enabled non-expensive IoT nodes to have secure hardware capable of supporting remote attestation procedures [31]. Using this technology, we want to provide TMS as a logically-centralized service, implemented using Byzantine Fault-Tolerant (BFT) State Machine Replication (SMR), capable of verifying if the devices satisfy the minimal requirements for participating in an application. Once a device is accepted on the application, the TMS can (optionally) provide key material for helping establish secure links between devices, effectively acting as a root of trust for IoT applications.

4.2 Background

In this section we describe the main technologies and concepts underlying the design of TMS. Parts of the text used here are abridged versions of material that appeared in previous publications of the author, in particular [40, 28].

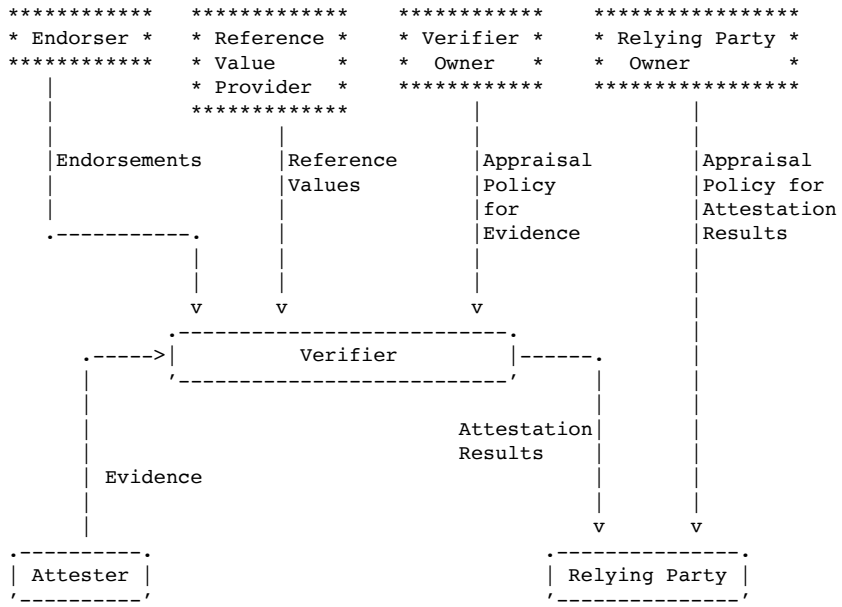

Figure 4.2. RATS architecture [31].

4.2.1 Remote Attestation

Remote attestation is a method by which a node, the *attester*, proves to another, the *verifier*, some properties of its hardware and software [31]. Typically, this is done by using some secure hardware components such as a Trusted Platform Module [86] or a processor-native trusted computing technology such as Intel SGX [53] and ARM TrustZone [77], deployed on the attester. It is also not uncommon that the verifier uses the evidence provided by the attester node to generate an attestation result (or proof, if successful) to inform another node, the *relying party*, about the properties of the attester.

Figure 4.2, taken from [31], illustrates how this architecture work and which kind of information is required from all intervening parties. As can be seen in the figure, the attester produce evidence about its properties, e.g., a set of local software measurements (hashes), signed with a hardware key certified by the manufacturer, for the verifier. The verifier checks the validity of this evidence using endorsements (e.g., a manufacturer public key) and reference values (e.g., a cryptographic hash of an operating system) to validate the claims made by the attester, producing the attestation result. Such result can be compared with an application-specific appraisal policy either on the verifier or on the relying party (or both). This mechanism can, for instance, be used to ensure a relying party only interact with an attester if it proves it is running a certified hardware with the most up-to-date version of a certain operating system.

The objective of TMS, as explained before, is to act as a verifier, enabling secure interaction between nodes in an IoT application.

4.2.2 Coordination Services

Coordination services provide a consistent and highly-available data store with enough synchronization power [50] for client processes to execute tasks such as mutual exclusion and leader election, and to store important system configuration. Such clients are often service processes deployed on clusters of hundreds or thousands of servers, but they can also be IoT devices deployed on untrusted environments. Three key fea-

tures of coordination services explain their wide adoption: (1) they provide a trust anchor for a much larger distributed system; (2) this trust is justified by their robust implementation using state machine replication protocols [57, 61] (see next section) to avoid any single point of failure; and (3) their accessible and limited interface, also called *coordination kernel* [52], which can be accessed through simple remote procedure calls (RPCs) that are intuitive even for programmers who are not experts in distributed computing [34].

Independently from the specifics of the coordination kernel and underlying implementation details, coordination services implement three main features:

- *Highly-available small storage*: The coordination service is often an anchor of trust in a much bigger distributed system and, amongst other things, is responsible for reliably managing small chunks of important data. Thus, it must be fault tolerant, which is usually achieved using the state state machine replication approach [79].
- *Interface with synchronization power*: Coordination tasks usually can be reduced to the problem of consensus [49]. Some older systems rely on the lock/lease abstraction [34], which is not wait-free since a client that fails with the lock can block other clients, at least for some time, until its lease expires. More recent systems provide wait-free operations [27, 52, 2] based on primitives with synchronization power [51], such as conditionally-atomic swaps and sequencers.
- *Client failure detection*: For implementing important coordination tasks such as leader election and fault-tolerant task assignment, it is essential that clients learn about the failure of other clients. Some systems explicitly provide such failure detection (e.g., by maintaining client sessions and notifying registered clients when such sessions terminate [52]), while others enable the expiration of objects, which can be interpreted as failures of the clients responsible for renewing the objects' time to live [27, 2].

We want the TMS to provide some of the capabilities of a coordination service, in particular to store important configuration for the IoT-applications it manages and for keeping an up-to-date list of active devices on each application.

4.2.3 State Machine Replication

In the *state machine replication* approach [79], an arbitrary number of client processes issue requests to a set of replicas. These replicas implement a stateful service that receives these requests and updates its state accordingly to the operation contained in the clients' requests. Once enough replicas transmit matching replies to the client, its invocation returns the result computed by the service.

The goal of this technique is to make the service state maintained by each replica evolve in a consistent way. In order to achieve this behavior, it is necessary to satisfy the following requirements [79]:

1. Any two correct replicas r and r' start with state s_0;
2. If any two correct replicas r and r' apply operation o to state S, both r and r' will obtain state S';

3. Any two correct replicas r and r' execute the same sequence of operations $o_0, ..., o_i$.

The first two requirements can be easily fulfilled if the service is deterministic, but the last one requires a *total order broadcast* primitive, which is equivalent to solving the *consensus problem* [49].

4.2.4 Blockchains

The concept of blockchain was introduced by Bitcoin to solve the double spending problem associated with cryptocurrencies in publicly open peer-to-peer networks [72]. A blockchain is an open database that maintains a distributed ledger comprised by a growing list of records called *blocks*, each of them containing transactions executed by the system. This authenticated data structure [82] consists of a sequence of blocks in which each one contains the cryptographic hash of the previous block in the chain.

A distributed system implements a *robust transaction ledger* (i.e., a blockchain) if it satisfies the following two properties (adapted from [43]):

- *Persistence*: If a correct node reports a ledger that contains a transaction tx in a block more than k blocks away from the end of the ledger, then tx will eventually be reported in the same position in the ledger by any honest node of the system.
- *Liveness*: If a transaction is provided as input to all correct nodes, then there exists a correct node who will eventually report this transaction at a block more than k blocks away from the end of the ledger.

Blockchain systems satisfy these properties abiding to either the *permissionless* or *permissioned* models [87]. Permissionless blockchains are maintained across peer-to-peer networks in a completely decentralized and anonymous manner [72, 89]. In order to determine the next block to append to the ledger, peers need to execute a Proof-of-Work (PoW) to create a valid block [43] (or an equivalent mechanism, e.g., Proof-of-Stake [44]) that is then disseminated to the network. The key idea behind the permissionless consensus, employed in Bitcoin and Ethereum, is to prevent an adversary from creating new blocks faster than honest participants. The first participant that finds a solution to a PoW puzzle gets to append its block to the ledger on all correct peers. Therefore, intuitively, as long as the adversary controls less than half of the total computing power present in the network, it is unable to tamper with the ledger.[1] This phenomenon also enables participants to establish a total order on the transactions by adopting the longest ledger with a valid PoW as the *de facto* transaction history.

The PoW mechanism makes permissionless blockchains slow and extremely energy demanding [87]. By contrast, permissioned blockchains do not expend as many resources and are able to reach better transaction latency and throughput. This is because nodes participating in this type of ledgers execute a traditional BFT consensus to decide on the next block to be appended to the ledger [36]. However, this approach requires a consortium of nodes that know each other for executing the consensus protocol. In this scenario, the bound on the adversary's power is structural,

[1] In fact the speed of the network also affects the maximum adversarial power tolerated, which is typically assumed to be much smaller than 50% [43].

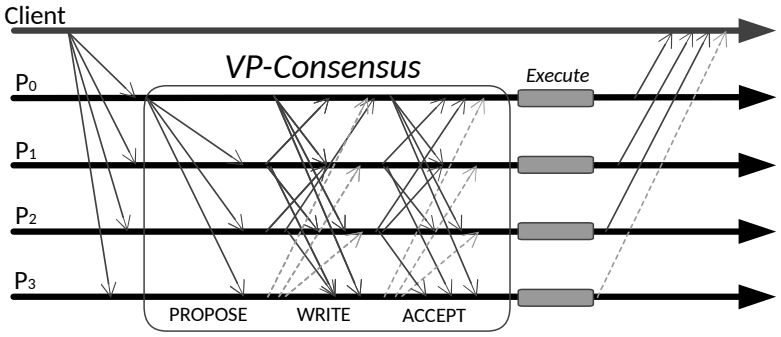

Figure 4.3. BFT-SMaRt ordering message pattern.

not computational, i.e., safety is ensured as long as the adversary controls less than a fraction of the nodes (usually a third).

4.2.5 BFT-SMaRt

BFT-SMaRt [30] is an open-source library that implements a modular SMR protocol [81], as well as features such as state transfer and group reconfiguration. This library have been developed by the VEDLIoT team at University of Lisboa during the last decade. Such development has been funded by multiple previous FP7 and H2020 projects, e.g., TClouds and SuperCloud. We will use this library to provide fault and intrusion tolerance for VEDLIoT TMS, therefore, in this section we describe its main features.

SMR protocol.

BFT-SMaRt uses the Mod-SMaRt protocol to implement the SMR properties described in the previous section. Mod-SMaRt is a modular SMR protocol that works by executing a sequence of consensus instances based on the BFT consensus algorithm described in [35]. The library implements Byzantine fault-tolerant state machine replication as described in previous section as long as less than a third of the replicas are faulty/compromised and the network is good enough to eventually deliver messages within certain (unknown a priori) time bounds [30]

During normal operation, the resulting communication pattern is similar to the well-known PBFT protocol [37] (Figure 4.3). Each consensus instance i begins with a leader replica proposing a batch of client operations to be decided within that instance. All replicas that receive the proposal verify if its sender is correct by exchanging `WRITE` messages containing a cryptographic hash of the proposed batch with all other replicas. If a replica receives `WRITE` messages with the same hash from more than two thirds of the replicas, it sends a signed `ACCEPT` message to all others containing this hash. If a replica receives `ACCEPT` messages for the same hash from more than two thirds of the replicas, it delivers the corresponding batch as the decision for this consensus instance, alongside *a proof comprised by the set of signed messages* received in this last phase.

If the leader replica is faulty and/or the network experiences a period of asynchrony, Mod-SMaRt may trigger a synchronization phase to elect a new leader for the consensus instances and synchronize all correct replicas [81].

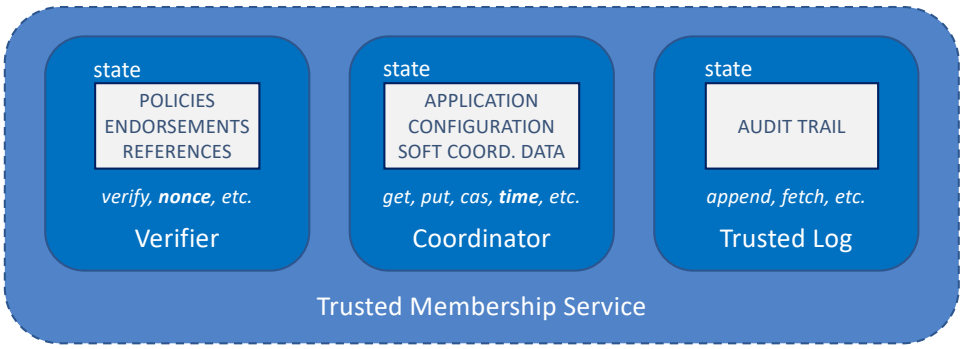

Figure 4.4. VEDLIoT Trusted Membership Service (TMS) main modules.

State transfer.

BFT-SMaRt also allows crashed replicas to recover and resume execution. This is done by using an intermediate layer between the Mod-SMaRt protocol and the replicated service, which is responsible for triggering service checkpoints and managing the request log.

The library provides two state transfer implementations in this layer. One uses an approach similar to PBFT that consists of storing the request log in memory which is periodically truncated after a snapshot of the service state is created. A recovering replica obtains the state by probing other replicas about their last completed consensus instance and asking $f + 1$ replicas to send the version of the state up to that instance.[2]

The other implementation is the durability layer described in [29]. When this layer is enabled, BFT-SMaRt stores the request log into stable storage to preserve the service state even if all replicas fail by crashing. In order to write requests to disk as efficiently as possible, delivered requests are written to the durable log in parallel with their execution by the service. To better exploit the large bandwidth of stable storage devices, the system tries to write multiple batches at once, diluting the cost of a synchronous write among many requests. This durability layer also enables replicas to execute checkpoints at different moments of their execution and a collaborative state transfer. These features alleviate the performance degradation caused by checkpoint generation and state transfer when the system is under heavy load.

A more recent work [28] extended BFT-SMaRt to keep a blockchain of transactions instead of a classical log. This is the version we plan to use for implementing TMS.

4.3 Trusted Membership Service

The VEDLIoT Trusted Membership Service is replicated infrastructure that will support remote attestation, application membership management, auditable integrity-protected storage and coordination primitives. These features will be implemented on each of the replicas in three main modules, as illustrated in Figure 4.4.

The first module, *Verifier*, is used to support remote attestation, following the ideas described in Section 4.2.1. This provide operations and functionalities for verifying the evidence provided by a device for attestation. To do this, the module must be

[2] In order to render this mechanism as efficient as possible, only one replica sends the entire state, while other f replicas send only a hash of it [37].

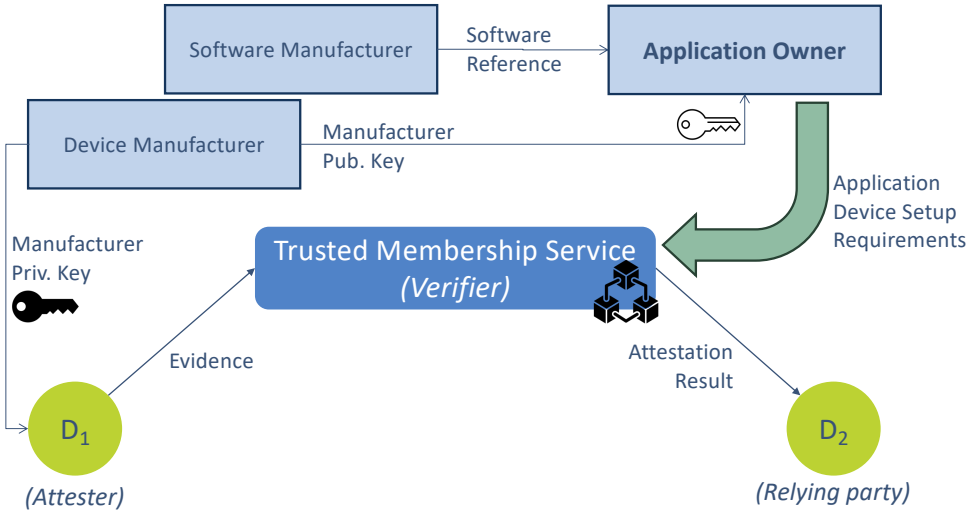

Figure 4.5. TMS operation for a device joining the IoT application.

pre-configured with endorsements (e.g., device manufacturer keys), references (e.g., software measurements) and application-specific policies, defining the requirements for each attested device to participate on a given application.

The second module is the *Coordinator*, which provides a key-value store interface enriched with a coordination kernel with synchronization power. This data will be kept in memory for enabling fast reads. This module can be used to store application configuration and some soft coordination data, e.g., the topology of a data aggregation tree composed by the application devices.

The last module is a blockchain-like *Trusted Log* used for storing information in an auditable data-structure. The information stored in this log will be persisted in stable storage for ensuring durability, even if all replicas crash (and later recover) [28].

TMS in action.

As explained before, these three modules will be deployed on each of the TMS replicas where they will interact locally to process each service request. Since the TMS will be implemented on top of BFT-SMaRt, and its extension implementing a blockchain [28], all these requests will be delivered to each replica in the same order, and processed deterministically.

Figure 4.5 illustrates how a replicated TMS can enable a device to join an application with zero manual configuration on the device. The process start with the service being configured by the application owner, which collects information from devices and software manufacturers for specifying the requirements for a device to participate on the IoT application. With this information, the TMS can verify if the evidence provided by D_1 is sufficient for including it on a certain application. The result of this attestation serves as a proof to other devices (e.g., D_2 in the figure) that D_1 is trustworthy enough to be included on the application.

4.3.1 Challenges

Implementing the replicated TMS we are proposing in VEDLIoT requires more than just the engineering effort of building the three modules described before on top of BFT-SMaRt. In this section we describe a set of challenges that need to be addressed

for making a fully-secure implementation of the service. We divided the challenges in two groups, the more immediate ones, which will be addressed in the first half of the project for having an usable version of TMS, and the long-term ones, which we will address not only to improve TMS, but also aiming to advance the state-of-the-art on BFT replication.

Immediate challenges.

There are two practical challenges that need to be solved for a replicated/BFT TMS to fully implement a remote attestation verifier [31]. Both of them are related with the determinism requirement of state machine replication. The first one is how to generate a secure timestamp relatively close to real time for registering the events stored in the system (e.g., join, leave, some data update) and ensure freshness for the remote attestation protocol messages. The challenge here is that clock reads on the different replicas will be different, so a robust method need to be used for consolidating a single timestamp that is not subject to manipulation by compromised replicas. Existing solutions for this typically employ a leader-generated timestampp [37] or require an additional protocol for approximate agreement on the clock values. We have to investigate and implement a method that is, at the same time, efficient and robust enough to be used in TMS.

The second challenge is similar, but related with adversary-resilient nonce generation. Nonces, in this case, should be random values generated by the replica group in an unpredictable way, even if some of the replicas are compromised by an adversary. To solve this we plan to investigate ways to get different random numbers from different replicas and generate a single nonce based on these. We believe a solution to the timestamp problem could also be used for nonce generation.

Long-term challenges.

Besides the immediate problems that need to be addressed for implementing TMS, we also want to advance the state-of-the-art on BFT replication by supporting two non-functional requirements that are not yet properly addressed in BFT state machine replication.

The first one is how to ensure the confidentiality on the data stored on a replicated service. Typically, BFT SMR maintains the *integrity and availability* of a secure service even if a fraction of the replicas fails in an arbitrary/Byzantine way (due to accidental or malicious events) [62, 37]. However, when considering the implementation of intrusion-tolerant services [42], BFT SMR systems do not provide any means for securing the *confidentiality* of the data kept in the service (i.e., its state) in face of compromises. For example, if a single replica of our proposed TMS is compromised, the adversary can access all the data stored in its three modules. In VEDLIoT, we are investigating ways to integrate secret sharing protocols [80] on BFT SMR for ensuring confidentiality of stored data.

A second challenge we want to address is how to improve the *scalability* of BFT-SMaRt (and consequently, TMS) in terms of the number of replicas supported by the system. As can be seem in Figure 4.3, BFT state machine replication typically require replicas to exchange a large number of messages to reach consensus. In particular, BFT-SMaRt requires a quadratic number of messages (on the number of replicas) to be exchanged for establishing agreement on the next batch of messages to be pro-

cessed. Although the performance of such protocols for small replica groups (< 10 replicas) is considered sufficient for most applications [30, 28], when a large group of replicas is considered, or when these replicas are deployed in different geographical locations, this negatively impact the performance of the protocol. In VEDLIoT we want to investigate new probabilistic agreement protocols that scale better when we using 10s to 100s of machines, without sacrificing latency (as done in many works addressing this problem, e.g., [90]), since TMS operates in an IoT environment scenarios where fast response is important.

4.4 Closing Remarks

This chapter presented the Trusted Membership Service (TMS), a component we want to introduce in the VEDLIoT architecture for securely manage devices on IoT application. TMS will implement features such as remote attestation, coordination, and auditable storage (of small data and configurations). We outlined the design of the system to be implemented in a decentralized way, using BFT-SMaRt, a Byzantine fault-tolerant replicated library [30]. The implementation of the service just started and we expect to report it on future deliverables of WP5. We also presented a set of challenges that we plan to address during the implementation of TMS. We also expect to report on these challenges on papers and future VEDLIoT deliverables.

5 Embedded Trusted Execution Environment

The demand for lightweight trusted execution environments (TEEs) for networked embedded microcontrollers (i.e., the Internet of Things) has drastically increased over the last few years. The ARM Cortex-M processor core series are the market leader for low-power networked embedded devices and are the first to introduce TrustZone for Cortex-M, a set of hardware extensions which enable the construction of TEEs for low-power embedded devices. The RISC-V ISA is an emerging open-source architecture and differentiates itself from existing instruction sets, such as ARM and x86, through its extensibility. Bare-metal embedded devices with a single privilege level can be built with only the core ISA. In many use cases, these devices are synthesized on FPGAs to reduce time-to-market, and to reserve the possibility of patching the hardware post-production.

Several FPGA-ready open-source RISC-V implementations have been developed, such as the Rocket Chip [26] and LiteX SoC with VexRiscv core [59]. These projects have greatly improved RISC-V's accessibility as a platform for research in computer architecture. The free and open RISC-V ISA is rapidly gaining market share in the soft processor space, but the community has yet to ratify a TEE extension. In VEDLIoT's Work Package 5, we are working on the first design of RISC-V's physical memory protection (PMP) hardware, which is optimized for FPGAs. We discuss the background, design, and initial implementation of the work in the following sections of this chapter.

5.1 Threat Landscape

The global trend of automating industrial practices with large-scale direct communication between machines over the Internet has dramatically altered the modern cyber threatscape. Society is now in need of embedded systems capable of both over-the-air (OTA) updates and remote attestation to maintain the growing fleet of IoT devices. These features allow operators to patch vulnerabilities, recover from a compromise and verify device integrity without resorting to costly and labour-intensive physical inspections. Despite the need, few embedded devices can perform these operations, since they require hardware-enforced isolation guarantees. Trusted Execution Environments (TEE) or enclaves, provide a virtual runtime for critical software and can be used to isolate key storage, the secure bootloader, OTA update mechanisms and remote attestation software. With TEEs, if an attacker gains control of the OS, the compromised device will be unable to attest its own integrity nor to prevent the owner from remotely restoring the device.

FPGAs are widely used in both consumer and industrial embedded systems. They are components in virtually all of the European Space Agency's hardware, and are viewed by the ESA as a key enabler of Space 4.0 [83, 33]. Intel and Xilinx, two leading FPGA technology firms, both offer ISO 26262-certified chips for automotive applications [14, 16]. They have even been used as baseboard management controllers [19], which provide remote administrator access to servers via the motherboard. With the proliferation of networked FPGAs in critical applications, strong security for soft processors must become a priority. Often, a soft processor can fill design gaps where a finite state machine would be too complex for the task, yet an off-the-shelf MCU lacks the needed versatility. In some use cases, the ability to remotely modify the hardware

itself is desirable, which is only possible on FPGAs. A safety- or security-critical bug in an ASIC could lead to the recall of an entire vehicle fleet, or permanently disable a satellite. After the discovery of the Meltdown and Spectre vulnerabilities in virtually all multi-core speculative processors [60, 68], this possibility should be taken seriously. Though FPGA-based soft processors are generally more expensive, slower, and less efficient than ASICs, their popularity is steadily growing and will continue to do so for the foreseeable future.

There are a handful of well-established, free and open ISAs available for use in soft processors, such as Power ISA, OpenRISC 1000 and Lattice Mico32 [15, 20, 18]. However, RISC-V, a relative newcomer, has rapidly gained market share due to its uniquely democratic process of revision and ratification, which promotes community engagement [13]. Revisions to the RISC-V ISA are made through public working group (WG) discussions and pull requests on the specification's GitHub repository. Implementers must adhere to the specification to take advantage of the compiler toolchains and operating system ports in the RISC-V ecosystem, but there are no contractual obligations to do so. There are ongoing discussions in the RISC-V TEE WG on how best to facilitate enclave support. We have designed optimizations for a proposal from that WG – Physical Memory Protection (PMP) and investigate its implications for soft processors. Our novel microarchitectural optimizations aim at reducing the overhead of PMP on FPGAs. We also plan on continuing this development to the Memory Protection Unit (previously known as supervisor physical memory protection - sPMP) which is an extension of the existing PMP hardware in RISC-V. It will enable secure and efficient code compartmentalization on embedded soft processors.

5.2 Physical Memory Protection in RISC-V

The RISC-V specification defines a physical memory protection extension which allows machine (M) mode to place R/W/X restrictions on lower privilege levels for arbitrary memory regions. M-mode can also lock regions, which applies the restrictions to all privilege levels until reset. On a multi-core system, each core has its own dedicated PMP unit. In embedded systems with memory-mapped IO, PMP is also used to restrict access to peripherals from unprivileged code. Regions are defined with two sets of control status registers (CSRs). The permissions and addressing modes for each region are defined in an 8-byte segment of the pmpcfg CSRs, while the base address is written to a pmpaddr CSR in an encoded format, depending on the mode. These are:

- TOR (Top of range): The region ends at the specified address, and starts at the end of the previous region.
- NA4 (Naturally-aligned 4-byte): The region starts at the specified address and is four bytes wide.
- NAPOT (Naturally-aligned power-of-two): The pmpaddr CSR contains both the start address and the size of the region, which is encoded as the number of trailing ones.

5.3 SpinalHDL

SpinalHDL is an open-source Scala-based hardware description language that allows developers to borrow concepts from functional and object-oriented programming. It

is specifically designed for FPGA development. The SpinalHDL compiler can generate both VHDL and Verilog code from Scala source files.

5.4 VexRiscv

VexRiscv is an open-source RISC-V processor written in SpinalHDL. In 2018, it was awarded first place in the RISC-V Foundation's SoftCPU contest for achieving the highest performance on both Lattice and Microsemi FPGAs [12]. It can be scaled to fit a wide range of use cases, from highly constrained bare-metal embedded applications to multi-core Linux with the same code base. VexRiscv takes a uniquely software-inspired approach to hardware description. Virtually every component is a plug-in object connected to one or more stages in the pipeline. Plug-ins can insert data into the pipeline at one stage, and VexRiscv will automatically propagate those signals through down the pipeline to be accessed by later stages. Plug-ins can also provide services, which are essentially APIs for other plug-ins to interact with. The result is a highly configurable CPU where everything down to the register file can be changed.

5.5 LiteX

LiteX is an open-source Python-based system-onchip (SoC) builder and is compatible with FPGAs from several vendors (e.g., Intel, Lattice, Microchip, Xilinx, etc.). The LiteX toolbox includes support for many peripherals (DRAM, Ethernet, SATA, USB, PCIe, etc.), enabling fast prototyping of complex SoC designs. Developers can choose from an array of RISC-V CPUs (VexRiscv, Rocket, PicoRV32, etc.) or import their own Wishbone-compatible Verilog design. LiteX is used in both commercial products and research projects, such as the Kestrel SoftBMC, TimVideos recording/streaming hardware and the LiteX Row Hammer Tester from Antmicro. In this work, we build our designs for a Xilinx Artix-7 FPGA with Vivado Design Suite 2017.1.

5.6 Zephyr RTOS

Embedded Real-time Operating Systems (RTOS) utilize memory protection hardware to isolate threads from kernel memory. This is an energy and cost-saving alternative to MMU-backed process isolation (e.g., Linux) because without virtual addressing, one can forgo page tables and a TLB. Zephyr RTOS, which runs on VexRiscv has kernel support for PMP. Each userspace thread requires, at minimum, three dedicated protection regions: one for global read-only data, an execute-only code region and one for thread stack memory (non-executable). The OS can assign any remaining regions to multiple threads for use as a shared buffer. On every context switch, the PMP control registers are overwritten with the next scheduled userspace thread's bounds. We use Zephyr's PMP concepts and usage to contextualize our hardware optimizations.

5.7 PMP Implementation in SpinalHDL for VexRiscv

In this section, we present our design and optimizations of PMP in SpinalHDL for VexRiscv. In order to establish the current state-of-the-art, we first analyzed three existing open-source implementations of PMP on other CPU cores:

- Rocket is an ASIC-optimized core written in Chisel [17].
- Ariane is a 6-stage core written in SystemVerilog [92].

- NEORV32 is a 2-stage core written in VHDL [73].

Each of these CPU cores implement PMP support in roughly the same way. A dedicated decoder unit (DU) is duplicated for each protection region, which translates the PMP control registers into region bounds. The set of region bounds and permissions are inputs to a permission checker unit (PCU), which is duplicated for each memory port. The PCUs use purely combinatorial logic to determine the read, write and execute permissions for each memory access request in a single CPU cycle. To get a performance baseline for our optimization efforts, we initially followed the purely combinatorial approach to PMP for our VexRiscv implementation, as was done in other open-source projects. We synthesized the full CPU with a LiteX SoC for an Artix-7 A35T FPGA and found that the hardware footprint, measured in terms of lookup table (LUT) utilization, is nearly doubled with the addition of a fully-featured PMP plug-in. We then designed a series of optimizations based on region alignment, addressing modes, granularity, logic, etc. We discuss a few of them below.

The PMP implementations used in other open-source projects use purely combinatorial logic. Each protection region has its own dedicated DU, which is problematic for FPGA synthesis. This component contains word-sized arithmetic operations, each of which occupy several CLBs when synthesized for our target platform. Each region is configured by an 8-bit field in a pmpcfg register, so up to four regions can be simultaneously modified on a 32-bit machine (e.g., VexRiscv). This means that on any given CPU cycle, there are at most four DUs computing new region bounds; the rest are idle. We observe that, in fact, a single DU is sufficient to reprogram all PMP regions if the CSR writes are done sequentially. Our solution introduces a finite state machine which inserts pipeline stalls as it moves the single DU from one region to the next. This distributes the operation over several cycles. As a result of these optimizations we have roughly halved PMP's hardware footprint on our target FPGA in exchange for what is essentially a marginal increase in context switch latency. On Artix-7 architectures, it is much more efficient to store data arrays (e.g., the pmpaddrN registers) in distributed RAM rather than registers. For example, a 32 × 6-bit dual-port RAM can be implemented with just 4 LUTs [21]. In the initial implementation, every pmpaddr register is accessed in parallel. In order to make use of distributed RAM, these must be accessed one or two at a time. Our sequential DU optimization does precisely that. Our PMP implementation is with 4-byte granularity and 16 regions. Future versions of the PMP specifications are expected to support 64 regions.

5.8 Closing remarks

The future of this research includes extending PMP to implement the features of the proposed Memory Protection Unit (MPU) extension. This entails adding a new set of supervisor-accessible CSRs and another layer to the permission checking logic. Since MPU registers are encoded in the same way as the existing PMP registers, a single sequential decoder unit can be shared between the two plug-ins. As this proposal is still very much in flux, we've omitted a few nuances and are working on implementing only the core functionality where the MPU behaves almost identically to PMP, but there is no locking option. The MPU would enable setting up Trusted Execution States (TES) in RISC-V based embedded systems. Another direction for TES research in this work-package is developing a fine-grained separation of resources and peripherals in trusted regions of embedded devices. The TES can also be used to enable remote

attestation for low-end IoT devices.

6 Custom Function Unit Support in Renode

Renode [24] is an open source, software-agnostic simulation framework created by Antmicro that provides plug-and-play building blocks for creating custom, single or multi-node, virtual hardware setups. It provides access to extensive debugging, tracing and analysis functionalities, and can be integrated with local infrastructure for an automated, Continuous Integration-driven development and testing flow. It supports a broad range of the most popular architectures, e.g. Arm, RISC-V, OpenPOWER, provides full determinism of execution and ensures that the developed software behaves just as if it was running on real hardware.

Besides a range of SoCs, Renode supports various sensors, enabling developers to feed their systems with real video, audio, accelerometer and other types of data. This allows for automated testing of real-life scenarios, without the need of preparing Renode-specific software versions. This capability of employing sensor data and working with the same software that is running on target hardware is used by a range of organizations focused on Machine Learning. For example, Renode is part of the Continuous Integration system of TensorFlow Lite Micro, a popular Machine Learning framework for MCUs.

Renode allows users to freely combine CPUs, SoC inputs/outputs and peripherals to create complex, heterogeneous systems that can be easily modified without having to deal with tedious manual reconfiguration, flashing and resetting of physical boards. It is also very often used for design space exploration, allowing evaluators to come up with better architectures for their new products. Moreover, Renode can help developers facing low availability of hardware, or even enable software development before the hardware is on the market, which was most prominently exemplified by Renode's pre-silicon support for Microchip's PolarFire SoC, a RISC-V FPGA SoC.

A single emulated machine in Renode consists of several key elements - one or more CPU cores in one of the supported architectures, memory blocks and peripherals. Different blocks can be combined together in a Renode Platform file, reflecting the memory map of a specific SoC. These files can be easily adjusted by users, allowing them to introduce changes or improvements to emulated platforms.

Renode is an Instruction Set Simulator, which means it does not try to reflect the internal processing of the CPU or peripheral blocks. From the CPU perspective, it works on an instruction level, keeping track of the user-visible state. The peripherals, on the other hand, are modeled on the register level - their behavior is consistent with the actual hardware, but the internal operation, timing of events, etc. may not always reflect the real IP. This level of simulation is a good compromise between fidelity and performance.

CPU emulation is performed using the binary translation technique - Renode analyzes the machine code of the loaded software and translates it into the machine code of the host machine (typically AMD64). This allows for very efficient execution of the guest code. The translator is implemented in C, ensuring high performance of the process.

Peripheral devices are usually modeled in C#. Apart from the registers, peripherals

can expose interrupt and GPIO lines, or any other external interface that is needed. The high level programming language used for modeling makes this process relatively easy compared to modeling in lower level languages.

Within the VEDLIoT project, Renode aims to support developers with testing and debugging capabilities, especially in the area of edge AI development.

6.1 Custom Function Unit

With the emergence of tiny ML solutions, designers are more and more inclined towards using custom accelerators to increase their small systems' capabilities. There are many hardware accelerators dedicated to offload edge AI computation, but these generic solutions do not always address the needs of very specific tiny ML applications.

The emergence of RISC-V ISA, defined with extendibility in mind, allows designers to create very custom accelerators, tailored to their requirements. They can add specific instructions, occupying the unused opcode space in the ISA to offload complex operations that are common in their use case. While technically possible, this process can be difficult without proper organization and support, both from the hardware and the software perspective. Without a proper standard to describe such accelerators, designers need to spend time defining and implementing interfaces that could otherwise be reused and shared among the community. To address this problem, a RISC-V FPGA Soft Processor Working Group (with an active participation from Antmicro) prepared a draft RISC-V Custom Function Unit (CFU) Specification.

6.1.1 RISC-V CFU specification

A Custom Function Unit is a hardware implementation of some algorithm that is identified as "hot" in a specific use case. The designer creating a CFU aims to reduce the load of the CPU by moving several steps of the original solution to a faster, hardware-based implementation.

The goal of the CFU specification is to define interfaces and composition methods of Custom Function Units. The specification defines signals, software primitives, metadata and composition rules for CFUs.

The specification defines CFUs as hardware cores that:

- Accept requests and produce responses;
- May be either stateless or stateful;
- Do not have direct access to the system memory;
- Do not have access to internal CPU state, apart from the parameters passed to them.

Included in the draft are the definitions of the logic interface, the software interface, instruction encoding and CPU/CFU metadata for automated block composition. While this is still work in progress and the specification is not ratified as an official RISC-V extension, practical implementations of the draft are available, with the most prominent being the CFU Playground project [46] by Google (also supported by Antmicro).

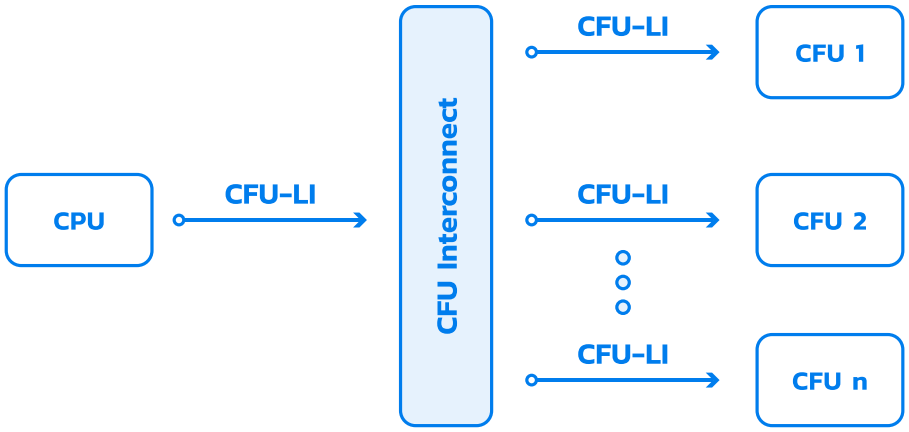

Figure 6.1. CPU/CFU complex

6.1.1.1 CFU Logic Interface (CFU-LI)

The CFU-LI defines ways to connect CPUs and CFUs together. The specification defines that the CPU always serves as the initiator of communication and the CFU responds to CPU's requests. The interface is layered into 5 levels:

- Level 0, for single-cycle combinational operations
- Level 1, for fixed-latency, pipelined operations with in-order completion
- Level 2, for variable-latency, possibly pipelined, in order operations with optional CFU backpressure
- Level 3, for variable-latency, possibly pipelined, in order operations with optional CFU and CPU backpressure
- Level 4, for variable-latency, possibly pipelined, in-order issue/out-of-order completion, non-cancellable operations, with CFU and CPU backpressure

The increased level of complexity of these levels is reflected in the number of signals required to connect the CFU and the CPU, starting from 6 signals for Level 0 and ending with 16 signals for Level 4.

6.1.1.2 Software interface

The CFU specification has to address several aspects of the software interface:

- A system may include many CFU units. They have to be selectable by the software
- A CFU may support many distinct operations (e.g. initialization, specific calculations, etc)
- A CFU may support multiple, selectable contexts with internal state
- A CFU request may fail, either due to improper internal state or an invalid call (e.g. wrong CFU unit selected)

RISC-V allows for custom instructions to be placed in any empty slot of the opcode space. The CFU specification, however, assumes that specific `custom-0` and `custom-1` opcodes (as defined by the RISC-V ISA) will be used for CFUs.

The `custom-0` instructions follow the RISC-V R instruction format, with the following operands:

- `rs1`, `rs2` - CPU registers which values should be passed to the CFU
- `cfid` - the CFU function ID
- `rd` - the CPU registers to hold the result of the operation

The `custom-1` instructions follow the RISC-V I instruction format, with the following operands:

- `rs1` - the CPU register which value should be passed to the CFU
- `imm` - an immediate value
- `cfid` - the CFU function ID
- `rd` - the CPU registers to hold the result of the operation

To support multiple CFU units with multiple contexts, a new machine-level CSR is proposed: `mcfu_ctrl`, with the following structure:

- `[7:0] cfu_index`
- `[15:8] reserved`
- `[23:16] state_index`
- `[31:24] reserved`

Since the CFU specification is still a draft, there are many aspects to the interface that are not yet defined: CFU discovery process, multi-hart support, different privilege level support etc. These details, while very important for physical implementation of CFUs, should not influence the emulation in a significant way.

6.2 Renode CFU integration

6.2.1 Current state

Since RISC-V ISA is designed to be extensible, Renode provides several features to help users take advantage of this possibility. The most important feature in this context is the ability to define custom instructions.

Renode users can define custom instructions in two ways:

- they can be implemented in C#, with full access to the rest of the framework,

- they can be implemented in Python, trading some flexibility and performance for the ease of prototyping.

To define a custom instruction, the user has to provide a masked instruction pattern (specifying required zeros and ones) and the actual implementation. It is worth noting that neither Python nor C# implementations require recompilation of Renode - everything can be loaded in runtime. Both of these approaches are well suited for CFU modeling. The current custom instruction support has two drawbacks that are going to be addressed in the course of the project.

6.2.1.1 Lack of dedicated CFU abstraction

Even though the CFU interface is currently being specified, users wanting to implement CFU functions in Renode cannot benefit from it. As the Renode custom instruction API is generic, each custom instruction has to perform the same routine of opcode parsing, arguments extraction and validation, etc. One of the preliminary goals to be achieved as part of VEDLIoT is to design and implement a CFU interface that will reduce the complexity of implementation of specific units.

6.2.1.2 Lack of Verilator support for CPU instructions

Custom Function Units are designed to perform arithmetic or logic operations, and they are usually developed in a hardware description language by appropriately skilled engineers. Requiring them to reimplement their HDL logic in C# or Python might prove to be a barrier reducing the adoption of Renode in their workflows. Moreover, if the implementation is developed in HDL first, then keeping both implementations synchronized requires additional maintenance work. If we consider the HDL implementation of CFU to be "software," then requiring users to reimplement them in higher level language violates one of Renode principles - run the same software that runs on hardware. For this purpose, this task aims to introduce an interface to allow HDL CFU implementation to work with Renode.

6.2.2 Verilog support in Renode

Renode is able to use memory-mapped peripheral models written in Verilog, via Verilator. Verilator is an open-source tool that translates Verilog into C++ or SystemC behavioral models. It is commonly used for simulation and testing of Verilog code, as it offers higher performance than event-driven logic simulators. HDL code translated with Verilator is commonly referred to as "verilated code."

To connect a verilated IP with Renode, the user needs to compile it against Renode's integration library. This shim layer enables simulation of:

- AXI4-Lite
- WishBone
- AXI4

Communication between Renode and the verilated block is executed via network sockets.

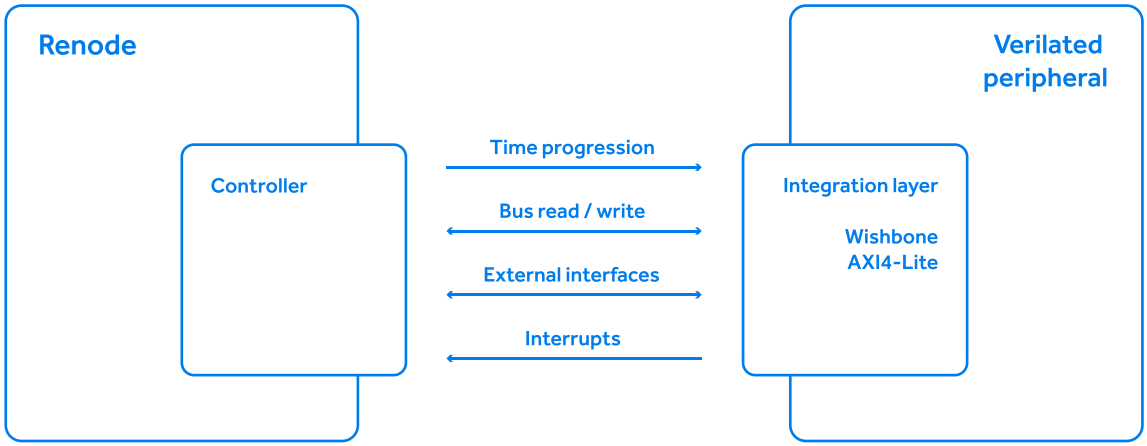

Figure 6.2. Renode - Verilator integration

6.2.3 Road towards CFU in Renode

To fully enable CFU support in Renode, two main tasks must be completed: adding support for an API-based integration (in contrast to the current socket-based one) and adding support for custom instructions handling.

6.2.3.1 API-based integration

As presented in the picture above, communication between Renode and a verilated IP is executed over two or more sockets. There are two main important features of this approach:

- Socket based communication enables distributed processing. This feature is important if we consider computation-heavy peripherals that are easier to simulate on a remote, more performant machine then the one used by the developer. On the other hand, we expect CFUs to be much simpler than peripherals. They should not require excessive processing power to complete an operation in a reasonable time.
- Socket based communication introduces additional processing overhead. Verilated peripherals in Renode are accessed in two situations: after a certain period of time and when the CPU tries to interact with them. While the CPU interaction depends solely on the software running on the emulated platform, the periodic communication can be configured according to the user's needs. Usually the user would prefer less frequent communication, trading fidelity for performance. It is assumed that the communication overhead is negligible in comparison to the actual processing time of verilated blocks. With the CFU units, however, there will be no periodic communication - they are accessed whenever a given CPU instruction is encountered. Since AI processing typically focuses on computation instead of peripheral access, with CFUs designed to offload most common instructions, it is assumed that accesses to verilated blocks will be frequent. This leads to a conclusion that socket-induced delay might greatly impact the whole emulation performance.

With this in mind, it was decided that a new interface will be designed, allowing verilated blocks to be connected over API instead of sockets.

The API was designed and the implementation is ongoing. One of the main challenges of this task is to hide the complexity of implementation from users attempting to connect their verilated models to the Renode's shim layer.

6.2.3.2 Verilated custom instructions

The current Verilator interface layer in Renode enables communication with perifer-als connected through a hardware bus. CFUs are designed as part of the processor pipeline, thus they do not fit in the currently available scenario. This task aims to define a new interface layer and integrate it with the current shim integration code. The API is expected to be simple. It will implement the register- and immediate-based functions, exception handling, register access and logging capability. The API will fo-cus on Feature Level 0, allowing the user to create combinatorial custom function units. Further levels will be considered at a later stage of the project.

6.3 Closing remarks

During the course of the project we improved the interface Renode uses to connect to verilated peripherals. We moved from a socket-based connection to a native API-based one, allowing us for more performant co-simulation. This will be a basis for future CFU connection. We introduced many improvements in the bus protocols han-dling and tested new scenarios of usage. Furthermore, we ported the support for co-simulation with verilated peripherals to Windows and macOS. All changes were released publicly as part of Renode repositories.

Future work will begin with designing and implementing the CFU interface. We will prepare an automated infrastructure for testing of the correctness of calculations and performance of the integrated setup. As a reference for CFU accelerator imple-mentations we will use the CFU Playground project [46]. As a follow-up, more CFU interface levels will be implemented to allow for more complex scenarios to be co-simulated with Renode. With these capabilities in place we will be able to focus on simulation of the SoC developed as part of Task 4.7.

7 Main achievements and future work

In the first 9 months of VEDLIoT, we started working in most tasks of Work Package 5 (some tasks initiated in month 7). Task 5.4 only starts in month 18. This document explained how the work is organised into tasks in the Introduction (Chapter 2). Following chapters reported the research and development work undertaken.

In Chapter 3, we presented our work on providing Twine, a **trusted execution environment** that is adapted to executing edge applications on Intel processors. We designed, implemented, and evaluated our implementation, and published a scientific paper describing our results. We are now working on a second component, fitted for ARM processors, that will be presented in the upcoming deliverables. We reuse part of Twine in this new component, and we are adding many functions for implementing **remote attestation**, which are absent in ARM processors.

We developed a highly optimised RISC-V **physical memory protection** as an extension to the RISC-V architecture, presented in Chapter 5. Our memory protection will be able to offer similar guarantees as found in ARM processors, and support similar attestation procedures as being currently developed in this Work Package. Our memory protection implementation is part of the latest VexRiscv (an open-source RISC-V soft processor) and the source code is openly available, with documentation and many links to supplementary materials. In the next steps, we intend to enhance our memory protection for establishing trusted execution states, allowing for simpler implementation when using user-space software.

In Chapter 4 we outlined the design of a **trusted membership service** to improve the security of distributed IoT applications. As discussed there, in the coming months we will be implementing a prototype of such service, targeting first a small number of replicas and connected devices. In parallel, we are investigating how to support this type of intrusion-tolerant, blockchain-like service in a more scalable way and preserving data confidentiality in case of successful attacks. Furthermore, we are also working on a robustness monitoring support for deep learning applications. Our first ideas are to monitor, detect and compensate bit flips and other non-malicious failures on devices. We expect to report progress on these efforts in the next Work Package 5 deliverables.

The improvements we developed for the **Renode framework**, an open-source simulator for complex embedded systems, are presented in Chapter 6. We worked on improving Renode with new features for custom function units, a mechanism very useful to expand a CPU with custom instructions to accelerate machine learning. Support for custom function units is based on Renode's integration with Verilator, a tool used to simulate peripherals with models written in Verilog. We improved the interface Renode uses to connect to Verilator peripherals and introduced many improvements in the protocols involved, and all changes were released publicly. For the near future, we will design and implement the actual custom function unit interface, along with an automated infrastructure for testing and benchmarking. This implementation will allow for more complex scenarios to be simulated with Renode, as the ones under development in Work Package 4.

8 References

[1] Binaryen, a compiler and toolchain for Wasm. Accessed on: Oct. 13, 2020.

[2] CoreOS etcd. `https://github.com/coreos/etcd/`.

[3] Cranelift, a code generator. Accessed on: Oct. 8, 2020.

[4] Lucet, a native WebAssembly compiler and runtime. Accessed on: Oct. 9, 2020.

[5] SQLite, measuring and Reducing CPU Usage. Accessed on: Oct. 14, 2020.

[6] Wasm3, the fastest WebAssembly interpreter. Accessed on: Oct. 10, 2020.

[7] Wasmer, a runtime for WebAssembly. Accessed on: Oct. 8, 2020.

[8] Wasmtime, runtime for WebAssembly & WASI. Accessed on: Oct. 9.2021.

[9] WAVM, a WebAssembly Virtual Machine. Accessed on: Oct. 14, 2020.

[10] WebAssembly Micro Runtime. Accessed on: Oct. 8, 2020.

[11] WebAssembly System Interface — WASI Application ABI, September 2020. Accessed on: Oct. 14, 2020.

[12] Alicia Daleiden. RISC-V SoftCPU Contest Highlights, December 2018.

[13] Andrew Waterman and Krste Asanovic. The RISC-V Instruction Set Manual Volume I: Unprivileged ISA, December 2019.

[14] Giulio Corradi and Steven McNeil. Xilinx Reduces Risk and Increases Efficiency for IEC61508 and ISO26262 Certified Safety Applications, October 2020.

[15] Intel. Nios® II Processor Reference Guide, October 2020.

[16] Intel Newsroom. Altera Functional Safety Package Combines FPGA Flexibility with 'Lockstep' Processor Solution to Reduce Risk and Time-to-Market, November 2015.

[17] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo and Andrew Waterman. The Rocket Chip Generator, April 2016.

[18] Lattice Semiconductor Corporation . LatticeMico32 Processor Reference Manual, December 2012.

[19] Michael Larabel. Raptor Announces Kestrel Open-Source, Open HDL/Firmware Soft BMC, January 2021.

[20] Opencores . OpenRISC 1000 Architecture Manual, January 2003.

[21] Xilinx . 7 Series FPGAs Configurable Logic Block, September 2016.

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

[23] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX. In *NDSS*, 2018.

[24] Antmicro. Renode. Accessed on: Jul. 1, 2021.

[25] 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 *USENIX OSDI'16*.

[26] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[27] Alysson Bessani, Eduardo Alchieri, Miguel Correia, and Joni S. Fraga. DepSpace: a Byzantine fault-tolerant coordination service. In *Proc. of the 3rd ACM European Systems Conference – EuroSys'08*, pages 163–176, April 2008.

[28] Alysson Bessani, Eduardo Alchieri, Joao Sousa, Andre Oliveira, and Fernando Pedone. From Byzantine replication to blockchain: Consensus is only the beginning. In *Proc. of the 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2020.

[29] Alysson Bessani, Marcel Santos, Joao Felix, Nuno Neves, and Miguel Correia. On the efficiency of durable state machine replication. In *Proceedings of the 2013 USENIX Annual Technical Conference*, San Jose, CA, USA, 2013.

[30] 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.

[31] H. Birkholz, D. Thaler, M. Richardson, N. Smith, and W. Pan. Remote attestation procedures architecture, 2021.

[32] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. DR.SGX: Automated and Adjustable Side-Channel Protection for SGX Using Data Location Randomization. In *ACSAC '19*.

[33] Klemen Bravhar, Victor Martins, Lucana Santos, and David Merodio Codinachs. Brave ng-medium fpga reconfiguration through spacewire: example use case and performance analysis. In *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 135–141. IEEE, 2018.

[34] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 335–350, 2006.

[35] Christian Cachin. Yet another visit to Paxos. Technical Report RZ 3754, IBM Research Zurich, 2009.

[36] Christian Cachin and Marko Vukolic. Blockchain consensus protocol in the wild (invited paper). In *Proceedings of the 31th International Symposium on Distributed Computing*, Vienna, Austria, 2017.

[37] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, New Orleans, Louisiana, USA, 1999.

[38] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX ATC'17*.

[39] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016. `https://eprint.iacr.org/2016/086`.

[40] Tobias Distler, Christopher Bahn, Alysson Bessani, Frank Fischer, and Flavio Junqueira. Extensible distributed coordination. In *Proc. of the 10th ACM European Systems Conference – EuroSys'15*, April 2015.

[41] Brendan Eich. From ASM.js to WebAssembly, June 2015.

[42] Joni Silva Fraga and David Powell. A fault- and intrusion-tolerant file system. In *Proceedings of the 3rd International Conference on Computer Security*, pages 203–218, 1985.

[43] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Proceedings of the 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Sofia, Bulgaria, 2015.

[44] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, Shanghai, China, 2017.

[45] David Goltzsche, Manuel Nieke, Thomas Knauth, and Rüdiger Kapitza. AccTEE: A WebAssembly-based two-way sandbox for trusted resource accounting. In *ACM Middleware'19*.

[46] Google. Cfu playground. Accessed on: Jul. 1, 2021.

[47] F. Gregor, W. Ozga, S. Vaucher, R. Pires, D. Le Quoc, S. Arnautov, A. Martin, V. Schiavoni, P. Felber, and C. Fetzer. Trust management as a service: Enabling trusted execution in the face of Byzantine stakeholders. In *IEEE/IFIP DSN'20*.

[48] 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 *PLDI'17*.

[49] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *Distributed Systems (2nd Ed.)*, pages 97–145. ACM Press/Addison-Wesley Publishing Co., 1993.

[50] Maurice P. Herlihy. Wait-free synchronization. *ACM Transactions on Programing Languages and Systems*, 13(1):124–149, 1991.

[51] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[52] 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.

[53] Intel Corporation. Intel processors with SGX extensions. Accessed on: Jan. 18, 2021.

[54] Intel Corporation. Overview of Intel Protected File System Library Using SGX, December 2016.

[55] Intel Corporation. *Intel Software Guard Extensions (Intel SGX) SDK for Linux OS — Developer Reference*, August 2020. version 2.11.

[56] 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*.

[57] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 41st International Conference on Dependable Systems and Networks (DSN'11)*, pages 245–256, 2011.

[58] L. Junyan, X. Shiguo, and L. Yijie. Application research of embedded database SQLite. In *2009 International Forum on Information Technology and Applications*, volume 2, pages 539–543, 2009.

[59] Florent Kermarrec, Sébastien Bourdeauducq, Hannah Badier, and Jean-Christophe Le Lann. Litex: an open-source soc builder and library based on migen python dsl. In *OSDA 2019, colocated with DATE 2019 Design Automation and Test in Europe*, 2019.

[60] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[61] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[62] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Trans. on Programing Languages and Systems*, 4(3):382–401, 1982.

[63] Chris Lattner and Vikram S. Adve. LLVM: a compilation framework for lifelong program analysis and transformation. In *IEEE CGO'04*.

[64] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *ACM EuroSys*, 2020.

[65] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything Old is New Again: Binary Security of WebAssembly. In *29th USENIX Security*, 2020.

[66] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. MiniBox: A two-way sandbox for x86 native code. In *USENIX ATC'14*.

[67] Samuel Lindemer. VexRiscv in VEDLIoT, July 2021.

[68] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 973–990, 2018.

[69] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. Twine: An embedded trusted runtime for webassembly. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pages 205–216. IEEE, 2021.

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

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

[72] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.

[73] S. Nolting. The neorv32 risc-v processor. `https://github.com/stnolting/neorv32`, 2020.

[74] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Andre Martin, Christof Fetzer, and Mark Silberstein. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *USENIX ATC '18*.

[75] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exit-Less OS Services for SGX Enclaves. In *ACM EuroSys'17*.

[76] Charles Papon. VexRiscv, a RISC-V implementation written in SpinalHDL, July 2021.

[77] Sandro Pinto and Nuno Santos. Demystifying ARM TrustZone: A comprehensive survey. *ACM CSUR*, 51(6):1–36, 2019.

[78] Louis-Noel Pouchet and Tomofumi Yuki. PolyBench/C, the Polyhedral Benchmarking suite 4.2.

[79] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computer Survey*, 22(4):299–319, 1990.

[80] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612?613, November 1979.

[81] Joao Sousa and Alysson Bessani. From Byzantine consensus to BFT state machine replication: A latency-optimal transformation. In *Proceedings of the 9th European Dependable Computing Conference*, Sibiu, Romania, 2012.

[82] Roberto Tamassia. Authenticated data structures. In *Proceedings of the 11th Annual European Symposium on Algorithms*, Budapest, Hungary, 2003.

[83] The European Space Agency. High Density European Rad-Hard SRAM-Based FPGA- First Validated Prototypes – BRAVE, 2017.

[84] Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yariv, and Noam Milshten. Switchless calls made practical in Intel SGX. In *3rd SysTEX*, page 22–27, New York, NY, USA, 2018. ACM.

[85] Tate Tian. Understanding SGX Protected File System, January 2017.

[86] Trusted Computing Group. Trusted platform module library specification, family ?2.0?, 2019.

[87] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In *Open Problems in Network Security - IFIP WG 11.4 International Workshop*, Zurich, Switzerland, 2015.

[88] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. Towards memory safe enclave programming with Rust-SGX. In *ACM CCS'19*.

[89] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2015.

[90] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *Proc. of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019.

[91] Alon Zakai. Emscripten: An LLVM-to-JavaScript compiler. In *OOPSLA/SPLASH*, page 301–312, New York, NY, USA, 2011. ACM.

[92] Florian Zaruba and Luca Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fd-soi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, 2019.