ICT-56-2020 — Next Generation Internet of Things

D5.2
Extended design and first implementation of Security, Safety and Robustness mechanisms and tools

Version 20

Document Information	
Contract Number	957197
Project Website	https://vedliot.eu/
Dissemination Level	PU (Public)
Nature	R (Report)
Contractual Deadline	30 April 2022
Author	Anum Khurshid (RI.SE)
Contributors	Marcelo Pasin (UNINE), Olof Eriksson (VEONEER), Hans-Martin Heyn (UGOT), Jämes Ménétrey (UNINE), Piotr Zierhoffer (ANT), Luca Costantino (FC.ID)
Reviewers	Shahid Raza (RI.SE), Micha vor dem Berge (CHR), Stefan Krupop (CHR), Oliver Brunnegard (VEONEER)

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**
1	2022-02-02	Document template created.
2	2022-02-03	Initial Table of Contents and chapter titles added.
3	2022-02-22	Added to Section 5: Safety <Hans-Martin>
4	2022-02-28	Added to Section 5: Safety <Olof / Hans-Martin>
5	2022-02-28	Added content to Chapter 4 and 5: Embedded TEE and Remote Attestation <Anum>
6	2022-03-04	Updated Figure 5.5 in Safety Chapter <Hans-Martin>
7	2022-03-04	Added chapter on Renode CFU support <Piotr>
8	2022-03-07	Added chapter 8: Robustness <Luca>
9	2022-03-07	Added chapter 3: Attestation, trusted WebAssembly
10	2022-03-09	Updated chapter 8: Robustness <Luca>
11	2022-03-10	Updated section 8.1: Robustness <Luca>
12	2022-03-11	Updated section 8.2: Robustness <Luca>
13	2022-03-14	Updated section 8.2 and 8.3: Robustness <Luca>
14	2022-03-28	Added and updated section 3.2 and 4.2: TEE <Anum>
15	2022-03-31	Added Summary and Introduction <Anum>
16	2022-04-01	Added Conclusion <Anum>
17	2022-04-04	Updated figures in Section 7 <Anum>
18	2022-04-22	Modifications after internal review <Anum>
19	2022-04-25	Modifications after internal review <Jämes>
20	2022-04-30	Finalization <Jens>

Table of Contents

List of Figures

List of Tables

1 Executive Summary

The **VEDLIoT** project, funded by EU, builds a Very Efficient Deep Learning IoT platform for the next generation of IoT architecture. The project comprises of 8 work packages active from the first month. The Work Package 5 (WP5) develops part of the platform focusing on providing and integrating tools and mechanisms for security, safety and robustness. This document, "D5.2: Extended design and first implementation of Security, Safety and Robustness mechanisms and tools" brings together the work carried out in Work Package 5 during months 9-18 of the project. It is divided into nine chapters including six technical chapters.

The first chapter introduces and summarises the structure, and content of the rest of the document. The second chapter is an introduction to the project and work package goals with respect to the five tasks of the Work Package. In Chapter 3, we describe the work carried out on remote attestation of IoT devices using Trusted Execution Environments. We also focus on the implementation of WebAssembly runtime and remote attestation for TrustZone in this chapter. Our work on state-of-the-art and cutting-edge Trusted Execution Environments to support trusted execution of critical IoT software has continued to branch out since the first few months of the project. We discuss the design and implementation of these TEE-based mechanisms, including TrustZone-M and RISC-V in the Chapter 4. In Chapter 5, we bring an initial discussion on safety standards with respect to ML specification, verification, inference and learning. We also discuss the integration of safety requirements with the work carried out in Work Package 2. Chapter 6 presents our work in extending Renode, a full-fledged open-source simulator for complex networks of embedded systems. In months 9-18, we have worked on supporting CFU - Custom Function Units, tailored one-off AI accelerators designed to be integrated with RISC-V soft CPU. One of the main goals and tasks in the Work Package 5 is to support development of distributed remote attestation services for software components of IoT. The implementation and architecture of such a trusted verifier service is described in detail in Chapter 7. We also discuss the initial experiments, results and future steps to argue robustness in the project, in Chapter 8. Finally, we close the document with Chapter 9 by providing our main achievements, future directions along with the next steps in each activity of the Work Package (carried out in months 9-18 of the project).

2 Introduction

VEDLIoT (Very Efficient Deep Learning in IoT) is developing a framework for the Next Generation of IoT architecture, which will be configurable to be placed at any level from sensor nodes to the cloud. This developing VEDLIoT platform is targeted towards cars, robotization and predictive maintenance of industry and assisted living at home. The huge amount of collected data and the high computational power required in such applications call for complex solutions within a limited time frame. In VEDLIoT, Artificial Intelligence (AI) and Deep Learning (DL) are utilized to handle the complexity of the problem. Another critical challenge is the **security, safety and robustness** for these systems, which is targeted in Work Package 5 (WP5). The following document follows up from the first deliverable (D5.1) as it takes the design and implementation of security, safety and robustness mechanisms and tools to the next level. A representation of the different components of VEDLIoT and their relation to WP5 is shown in Figure 2.1.

The work package is divided into five tasks which deal with the following targets: (i) the development of end-to-end trust through a distributed attestation mechanism, (ii) implementing communication security based on mutual attestation, (iii) providing means for secure execution of critical code on edge devices inside trusted execution environments and (iv) developing software-based monitoring and adaptation mechanisms to control safety and robustness in distributed AI systems.

Task 5.1 End-to-end attestation of distributed trusted environments (M1-M36)

Our efforts on implementing end-to-end attestation were introduced in the deliverable D5.1. We continue our work on implementing attestation and trusted execution on embedded systems which is part of the VEDLIoT goals. We conducted a systematic survey describing the general principles of attestation, in particular highlighting the differences between local and remote variants. The survey also targets the existing support for attestation mechanisms in the TEE implementations currently available in commodity hardware. Section 3.1 of this document describes the results and dis-

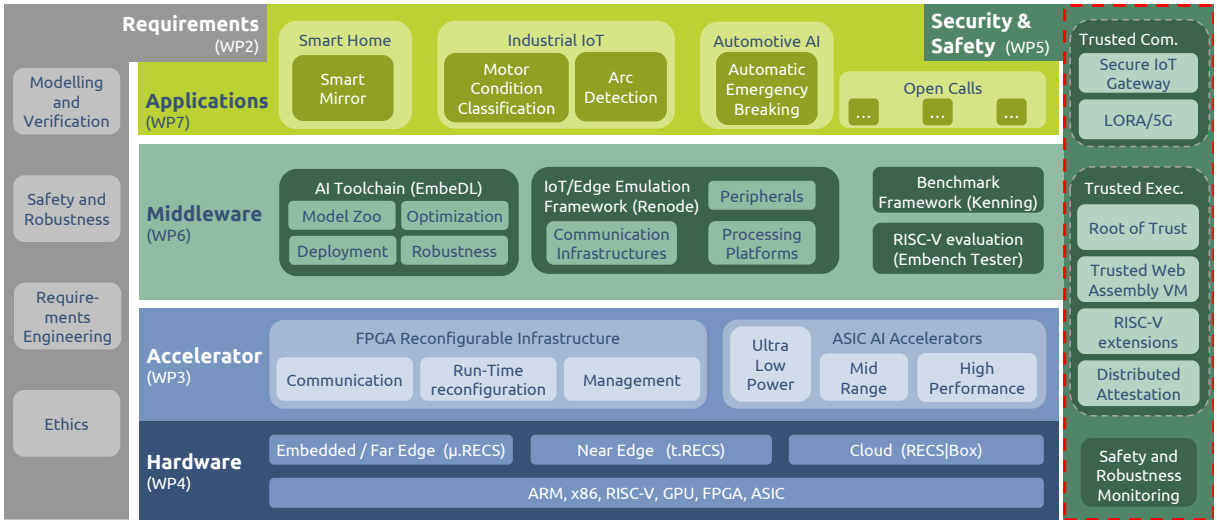


Figure 2.1. Global picture of VEDLIoT. This deliverable discusses WP5 related aspects.

cussions of the survey.

To support the attestation process, the work on the development of a distributed service, SIRE, capable of regulating the participation of devices on IoT applications has matured. The SIRE system is still not fully developed, but a few parts have now been implemented and are being tested for their security and scalability. We discuss the progress and future work on SIRE in Chapter 7.

It is known that almost all standard attestation mechanisms suffer from the drawbacks of the Time-Of-Check to Time-Of-Use (TOCTOU) invalidity. It remains a concern for the device owners and users that the attested state of the device may have changed during the interval between attestation and device usage. Since device certificates reflect the attested software stack of the device, if the attestation result of the device is not valid due to a race condition, the certificate is misleading. As a part of the work on attestation in this task, we have begun the design and plan for an automated remote attestation and certification scheme building on standards like Remote ATtestation procedureS (RATS) [44] to eliminate the TOCTOU problem. More details of this preliminary research are shown in Section 3.2 of this document.

Task 5.2 Security support for distributed execution and communication (M4-M36)

Our efforts on security support for execution of critical software and communication in VEDLIoT includes hardware-based Trusted Execution Environments (TEEs), *e.g.*, Arm TrustZone-A, TrustZone-M and emerging RISC-V solutions.

WebAssembly is a portable, compact, low-level, stack based binary code format that allows standalone execution running inside and outside browsers. Very few options exist at the moment to host Wasm applications inside TEEs. We exploit the sandbox capabilities from software fault isolation and control-flow integrity to isolate Wasm code from the trusted OS. We have proposed WasmOnTZ, which is a trusted runtime to execute Wasm applications with remote attestation capabilities (details in Section 3.3). To the best of our knowledge, WasmOnTZ is the first Wasm runtime running entirely inside Arm TrustZone with full support for remote attestation, optimised explicitly for Wasm to establish trust on hosted applications. Our design was implemented using a standard development board found in the market, verified and tested. With many different experiments, we measured the costs of using the specific mechanisms needed by WasmOnTZ. We were able to demonstrate that our implementation is lightweight, achieving results comparable with Wasm running without a trusted environment.

TrustZone-M is Arm's recent addition in Cortex-M based devices extending the support of TEEs in the IoT architecture. Our work in VEDLIoT on this state-of-the-art TEE includes identifying serious vulnerabilities in TrustZone-M and proposing efficient solutions. We have proposed a secure communication framework to build secure communication between the two worlds of TrustZone-M, using a lightweight message protection scheme leveraging the Memory Protection Unit (MPU). ShieLD is the first framework proposed to solve the issue of vulnerable inter-world communication for the TrustZone-M platform. We present an extensive micro-benchmark performance evaluation of it's overhead on system operations proving its suitability for constrained IoT devices. Another problem in TEEs arises when a secure application is compromised (due to an undetected vulnerability) as it becomes extremely challenging to detect this compromise and deploy remedial actions. Several exam-

ples of stealthy rootkits exploiting compromised TEEs exist. The fundamental problem is that in a multi-vendor environment, the device users have to trust that software in secure areas of the device is trustworthy and will not abuse the privileged access to manipulate or exfiltrate data. To counter this problem, we have proposed TEE-Watchdog, a framework to map user/vendor-defined policies to the system memory, efficiently detect access violations and register policy-violating application's behaviour. This mechanism ensures fine-grained access control over secure system peripherals, immutability of proposed system components and confidentiality and integrity of the associated data. Chapter 4 of this document is dedicated to the work related to Arm TrustZone-M.

We also continued the work on the novel RISC-V architecture, to support hardware-assisted trusted execution of critical code. Following up from our previous collaborative work on PMP support on VexRiscv, we observed an incompatibility between the PMP driver's system call routine and a built-in RISC-V security feature, which effectively renders one PMP region unusable (see Section 4.3). To this end, we are continuing our investigations for potential solutions.

Task 5.3 Simulation platform for development and testing (M7-M30)

We continue our work on developing the improvements to the Renode Framework to enhance the ease of development of embedded systems. In VEDLIoT, we focus on IoT and machine learning, and such an open-source simulator for complex embedded systems or of networks of such devices aligns well with the project goal. As part of the Work Package 4, VEDLIoT is developing an FPGA based accelerator for Machine Learning payloads. Handling custom hardware can be even more challenging from the software perspective, if the hardware is being designed in parallel with the software. The RISC-V ISA is FPGA-friendly and calls for proper verification of the software-hardware interoperability at all stages of development. Chapter 6 is dedicated to an accelerator that will be focused around the use of custom RISC-V instructions, hence we setup the proper testing functionality with the Renode framework.

Task 5.4 Continuous integration workflow (M18-M36)

This task is scheduled for month 18. The task will prepare a system as part of Task 4.6 and Task 4.7. The system will provide continuous integration capabilities allowing the project members to add their own tests ensuring fitness of the software on each stage of development. The tests will range from verification of basic functionality to analysis of full system behavior, including remote attestation and encryption mechanisms.

Task 5.5 Safety and Robustness (M7-M36)

The task started in month 7 and addresses robustness and safety in a distributed AI system, considering aspects from sensor data collection to data communication and processing at several levels, from specification to testing.

As a part of this task, we are considering the safety of using ML algorithms in high-stake real-world domains like the automotive domain, discussing it in Chapter 5. Our investigation of ML-based systems has identified gaps when it comes to safety. We concentrate on safety as a cluster of concerns that interacts with other aspects of the system, such as hardware configuration, communication, and AI modelling. Each cluster of concerns represents a specific aspect of the system at different levels of

abstractions, this concept is taken from the compositional architectural framework developed in WP2 [139].

Another major goal of this task focuses on development of monitoring and adaptation mechanisms to increase robustness in distributed AI systems. To achieve this goal, we have designed and implemented the local monitoring of a device within a more complex framework that ensures advanced traffic-safety during their runtime execution. We have conducted tests and experiments using multi-layer machine learning strategy to improve the performance of a model. In particular, we evaluate the idea of splitting a dataset into specific domains to train specific models for better performance. In this document, Chapter 8 talks about the tests and experiments to increase robustness in AI systems.

3 Attestation and Trusted WebAssembly Execution

Our first progress in Work Package 5 was made in implementing a trusted environment for running distributed applications on Intel SGX (Software Guard Extensions). We followed up with that work to support trusted applications in embedded devices, which turned our attention to Arm TrustZone. Along with trusted execution of applications, we also started analysing how remote attestation occurs on Arm hardware as compared to other systems. We focus on investigating remote attestation and trusted WebAssembly execution. This chapter is divided into three major sections: (i) a survey on remote attestation schemes for TEEs, (ii) a preliminary discussion on Time-Of-Check to Time-Of-Use (TOCTOU) invalidity and (iii) trusted WebAssembly execution.

3.1 Attestation Mechanisms for Trusted Execution Environments

Confidentiality and integrity are essential features when building secure computer systems. This is particularly important when the underlying system cannot be fully trusted or controlled. For example, video broadcasting software can be tampered with by end-users to circumvent digital rights management, or virtual machines are candidly open to the indiscretion of their cloud-based untrusted hosts. The introduction of Intel SGX, AMD SEV, RISC-V, Arm TrustZone-A and TrustZone-M Trusted Execution Environments (TEEs) into commodity processors significantly mitigates the attack surface against powerful attackers. In a nutshell, TEEs let a piece of software be executed with stronger security guarantees, including privacy and integrity properties, without relying on a trustworthy operating system.

Remote attestation allows establishing a trust relationship with a specific piece of software by verifying its authenticity and integrity. Through remote attestation, one ensures to be communicating with a specific, trusted (attested) program remotely. TEEs can support and strengthen the attestation process, ensuring that programs are shielded against many powerful attacks by isolating critical security software, assets and private information from the rest of the system. In this work, we describe the current state-of-the-art best practices regarding remote attestation mechanisms for TEEs, complementary to [99], covering four major technologies available for commodity hardware, i.e., Intel SGX, Arm TrustZone-A/-M, AMD SEV and emerging TEEs using RISC-V ISA.

In this section of the deliverable we describe the general principles of attestation, in particular highlighting the differences between local and remote variants. We also present a small survey on the existing support for attestation mechanisms in the TEE implementations currently available in commodity hardware. We conclude the section discussing some future directions for research on the subject.

3.1.1 Attestation

Attestation enables a trusted environment to prove its identity to another piece of software. The target environment that receives an attestation request can assess whether a given proof is genuine by verifying its authentication, usually based on a symmetric-key scheme. Such a mechanism is required to establish secure communication channels between trusted environments, and is often used to delegate comput-

ing tasks securely. Attestation can be done locally, when target and trusted environments are hosted on the same system, or on the same CPU if the secret provisioned for the attestation is bound to the processor. As an example, Intel SGX's remote attestation (detailed in Section 3.1.2.3) leverages a local attestation mechanism to sign proofs in another trusted environment (called the *quoting enclave*) through a secure communication channel.

Remote attestation: Remote attestation allows to establish trust between devices and provide cryptographic proofs that the executing software is genuine and untampered [51]. In the remainder, we adopt the terminology proposed by the IETF to describe remote attestation and related architectures [46]. Under these terms, *a relying party* wishes to establish a trusted relationship with an *attester*, with the help of a *verifier*. The attester provides the state of its system, indicating the hardware and the software stack that runs on its device by collecting a set of *claims*. Once the attester is proven genuine, the relying party can safely interact with it and transfer confidential data or delegate computations. Remote attestation mechanisms are popular among the TEEs, due to their carefully controlled environments and their ability to generate code measurements/claims.

Mutual attestation: Trusted applications may need stronger trust assurances by ensuring both ends of a secure channel are attested. For example, when retrieving confidential data from a sensing IoT device (where data is particularly sensitive), the device must authenticate the remote party, while the latter must ensure the sensing device has not been spoofed or tampered with. Mutual attestation [137] protocols have been designed to appraise the trustworthiness of both end devices involved in a communication.

Threat model: Remote attestation protocols usually follow the Dolev-Yao intruder model [60], which assumes that an adversary has complete control over the communication channel (*i.e.*, an attacker can do everything except breaking cryptography). Recent state-of-the-art formal analysers, such as Scyther [56], assist the automatic verification of security protocols. Typically, these tools assert the secrecy of the private session keys, and the following claims of authentication: *aliveness*, *weak agreement*, *non-injective agreement*, *non-injective synchronisation* and *reachability* (*i.e.*, the protocol ended on both parties). While we omit to present these terms for conciseness[57, 98], they represent essential characteristics for security protocols.

3.1.2 Issuing Attestations using TEEs

Several solutions exist to implement hardware support for trusted computing, and TEEs are particularly promising. Depending on the implementation, they guarantee the confidentiality and the integrity of the code and data of trusted applications, thanks to the assistance of CPU security features. This work surveys modern and prevailing TEEs from processor designers and vendors with remote attestation capabilities for commodity or server-grade processors, namely Intel SGX [53], AMD SEV [58], and ARM TrustZone [116]. Besides, RISC-V, an open ISA with multiple open-source implementations, ratified the Physical Memory Protection (PMP) instructions, offering similar capabilities to memory protection offered by aforementioned technologies [10]. As such, we also included many emerging academic and proprietary frameworks that capitalise on standard RISC-V primitives, which are Keystone [89], Sanctum [55], TIMBER-V [145] and LIRA-V [132]. We have omitted the TEEs lacking re-

Features	SGX	TrustZone		SEV			RISC-V			
		TrustZone-A	TrustZone-M	Vanilla	SEV-ES	SEV-SNP	Keystone	Sanctum	TIMBER-V	LIRA-V
Integrity	●	○	○	○	○	●	●	○	○	○
Freshness	●	○	○	○	○	●	●	○	○	○
Encryption	●	○	○	●	●	●	●	○	○	○
Unlimited domains	●	○	●	◑	●	●	●	●	●	○
Open source	◑	◑	◑	○	○	○	●	●	●	○
Local attestation	●	○	○	○	○	○	○	●	●	○
Remote attestation	●	◑	◑	●	●	●	●	●	●	●
API for attestation	●	◑	◑	○	○	●	●	●	●	●
Mutual attestation	○	◑	○	○	○	○	○	○	○	●
User-mode support	●	●	●	●	●	●	●	●	●	○
Industrial TEE	●	●	●	●	●	●	○	○	○	○
Isolation and attestation granularity	Intra-address space	Secure world		VM			Secure world	Intra-address space		
System support for isolation	µcode + XuCode	SMC	MPU	Firmware			SMC + PMP		Tag + MPU	PMP

Table 3.1. Comparison of the state-of-the-art TEEs

mote attestation mechanisms (*e.g.*, IBM PEF [75]) and TEEs not supported on currently available CPUs (*e.g.*, Intel TDX [125], Realm [28] from ARM CCA [32]).

3.1.2.1 Analysed characteristics

We propose a series of cornerstone features of TEEs and remote attestation capabilities and compare many emerging and well-established state-of-the-art solutions in Table 3.1. Each feature is detailed below and can either be missing (○), partially (◑) or fully (●) available. Partial fulfilment means no built-in support, but extended by the literature. While we define these features below, we elaborate further about each TEE in the remainder of the section.

Integrity: an active mechanism preventing DRAM of TEE instances from being tampered with. *Freshness*: protecting DRAM of TEE instances against replay and rollback attacks. *Encryption*: TEE instances' DRAM is encrypted for providing some assurance that no unauthorised access or memory snooping of the enclave occurs. *Unlimited domains*: many TEE instances can run concurrently, while the TEE boundaries between these instances are guaranteed by hardware. Partial fulfilment means that the number of domains is capped. *Open source*: indicate whether the solution is partially or fully publicly available. *Local attestation*: a TEE instance can attest another instance running on the same system. *Remote attestation*: a TEE instance can be attested by remote parties. *API for attestation*: an API is available by the trusted applications to interact with the process of remote attestation. *Mutual attestation*: the identity of the attester and the verifier are authenticated upon remote attestations. *User mode support*: states whether the trusted applications are hosted in user mode, according to the processor architecture. *Industrial TEE*: contrast the TEEs used in production and made by the industry from the research prototypes designed by academia. *Isolation and attestation granularity*: the level of granularity where the TEE operates for providing isolation and attestation of the trusted software. *System support for isola-*

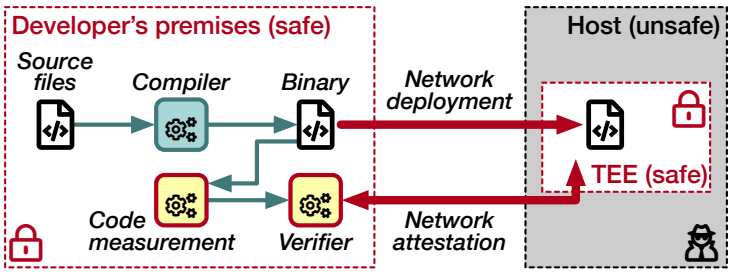


Figure 3.1. The workflow of deployment and attestation of TEEs

tion: the hardware mechanisms used to isolate trusted applications.

3.1.2.2 Trusted environments and remote attestation

The attestation of software and hardware components requires an environment to issue evidence securely. In practice, this role is usually assigned to some software or hardware mechanism that cannot be tampered with. These environments rely on measuring the executed software (*e.g.*, by hashing its code) and combining that output with cryptographical values derived from the hardware, such as a root of trust fused in the die or a physical unclonable function. We analysed today's practices for the leading processor vendors for issuing cryptographically signed evidences.

Figure 3.1 illustrates the generic workflow TEE developers usually follow for the deployment of trusted applications. Initially, the application is compiled and measured on the developers' premises. It is later transferred to an untrusted system, executed in the TEE facility. Once the trusted application is loaded and required to receive sensitive data, it communicates with a verifier to establish a trusted channel. The TEE environment must facilitate this transaction by exposing an evidence to the trusted application, which adds key material to bootstrap a secure channel from the TEE, thus preventing an attacker from eavesdropping on the communication. The verifier asserts the evidence, maintaining a list of reference values to identify genuine instances of trusted applications proceed to data exchanges.

3.1.2.3 Intel SGX

Intel Software Guard Extensions (Intel SGX) [53] introduced TEEs for mass-market processors in 2015. Specifically, Intel's Skylake architecture introduced a new set of processor instructions to create encrypted regions of memory, called *enclaves*, living within the processes of the user space. These instructions are their own ISA called XuCode [78] that together with Model Specific Registers (MSR) provide the requirements to form the implementation of SGX. A memory region is reserved at boot time for storing code and data of encrypted enclaves. This memory area, called the Enclave Page Cache (EPC), is inaccessible to other programs running on the same machine, including the operating system and the hypervisor. The traffic between the CPU and the system memory remains confidential thanks to the Memory Encryption Engine (MEE). The EPC also stores verification codes to ensure that the RAM corresponding to the EPC was not modified by any software external to the enclave.

A trusted application can receive a cryptographically signed evidence, *i.e.*, a *quote*, which may be enhanced by additional information, such as a public key (*e.g.*, used for establishing a communication channel). The quote binds a genuine Intel SGX processor with the measurement of the application when loaded into the enclave. This

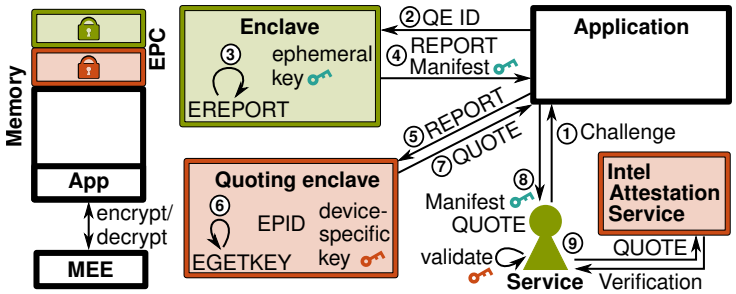


Figure 3.2. The remote attestation flow of Intel SGX

quote can then be forwarded to a relying party and be verified remotely using the Intel attestation service [21, 49] or a dedicated public key infrastructure [126].

Unlike local attestation, remote attestation requires an asymmetric-key scheme, which is made possible by the quoting enclave. The quoting enclave is a special enclave that has access to the device-specific private key through the EGETKEY instruction. In a remote attestation scenario, a service (*i.e.*, verifier) submits a challenge to the untrusted application with a nonce (Figure 3.2-①). Together with the identity of the quoting enclave, the challenge is forwarded to the application enclave (Figure 3.2-②). The application enclave (*i.e.*, attester) prepares a response to the challenge by creating a manifest (*i.e.*, a set of claims) and a public key (Figure 3.2-③), that is used to send back confidential information to the application enclave. The manifest hash is used as auxiliary data in the report for the local attestation with the quoting enclave. After verifying the report (Figure 3.2-⑥) the quoting enclave replaces the MAC with the signature from the EPID key and returns the quote (*i.e.*, evidence) to the application (Figure 3.2-⑦) which sends it back to the service (Figure 3.2-⑧). The service verifies the signature of the quote (Figure 3.2-⑨) using either the EPID public key and revocation information or an attestation verification service [21]. Finally, the service ensures the integrity of the manifest by verifying the response to the challenge.

Intel SGX has many advantages but suffers from many limitations as well. First, most of the SGX implementations limit the EPC size to 93.5 MB [141]. Newer Intel Xeon processors extend that limit to 512 GB. Besides, the enclave model prevents performing system calls and direct hardware access since the threat model distrusts the outer world, leading to the development of partitioned applications.

3.1.2.4 Arm TrustZone architectures

Depending on the architecture of Arm's processors, TrustZone comes in two flavours: TrustZone-A (for the Cortex-A) and TrustZone-M (for the Cortex-M).

Arm TrustZone-A provides the hardware elements to establish a single TEE per system [116]. Broadly adopted by commodity devices (including mobile devices, IoT edge nodes, *etc.*), TrustZone splits the processor into two states: the secure world (TEE) and the normal world (untrusted environment). A secure monitor instruction (*i.e.*, the SMC) is switching between worlds, and each world operates with their own user and kernel spaces.

Despite the commercial success of TrustZone-A, it lacks attestation mechanisms, preventing relying parties from validating and trusting the state of a TrustZone-A TEE remotely. Nevertheless, researchers proposed several variants of one-way remote

attestation protocols for Arm TrustZone [151, 92], as well as mutual remote attestation [17, 131]. Devices lacking built-in attestation mechanisms may rely on a secret fused in the die as a root of trust to derive private cryptographic materials (*e.g.*, a private key for evidence issuance). We describe the remote attestation mechanism of Shepherd et al. [131] as a study case. This solution establishes mutually trusted channels for bi-directional attestation, based on a Trusted Measurer (TM), which is a software component located in the trusted world and authenticated by the TEE's secure boot, to generate evidences based on the OS and TA states (*i.e.*, a set of claims). A private key is provisioned and sealed in the TEE's secure storage and used by the TM to sign evidences, similar to a firmware TPM [120]. Using a dedicated protocol for remote attestation, the bi-directional attestation is accomplished.

Arm TrustZone-A presents some advantages and drawbacks. Similar to SGX, the memory available to TAs, for instance, in the open source trusted OS OP-TEE, is limited to a few MB [70]. Due to this constraint, software needs to be partitioned to leverage TrustZone. Furthermore, OP-TEE is small and does not implement a POSIX API, making developing TAs difficult, notably when porting legacy code. While most components of TrustZone have open source alternatives (*e.g.*, the firmware and the trusted OS), many vendors do not disclose the implementation of the secure monitor.

Arm TrustZone-M, much like its predecessor TrustZone-A, provides an efficient mechanism to isolate the system into two distinct states/processing environments[33, 29]. The TrustZone-M extension brings the possibility of trusted execution into resource-constrained IoT devices (*e.g.*, Cortex-M23/M33/M35P/M55). Despite the similarity regarding the high-level concept, TrustZone-M differs from TrustZone-A in low-level implementation of some features. The switch between the secure and the normal world is embedded in hardware (carried out using `SG` instruction) and is much faster than the secure monitor instruction (*i.e.*, the SMC)[84, 34].

Since TrustZone-M is a relatively new addition, recently available for the IoT infrastructure, existing work on attestation mechanisms for the hardware/software is scarce. Nonetheless, TrustZone-M fulfils some basic requirements for attestation like (i) Secure storage, (ii) Secure boot, (iii) Secure inter-world communication and (iv) isolation of software. Thus, schemes like [15] have leveraged TrustZone-M to develop attestation and use TZ-M's TEE capabilities to establish a chain of trust. TrustedFirmware-M, following the guidelines of PSA, also supports initial attestation of device-specific data in the form of a secure service[101, 31].

TrustZone-M provides several advantages as a TEE to support remote attestation but also has a few drawbacks. It provides efficient isolation of the software modules and a faster context switch between the secure and normal world. This is advantageous as it is critical to have minimum attestation latency in the real-time operations of autonomous embedded systems. The availability of hardware-unique keys in TrustZone-M enabled devices further ensures that the attestation results generated by the TCB cannot be forged. Besides, the software stack may be fully open source, thanks to the absence of a secure monitor. On the other hand, since the components involved in measuring, attesting, and verifying the data/system need to be protected as part of the TCB, it increases the TCB size on the attested devices, raising the attack surface.

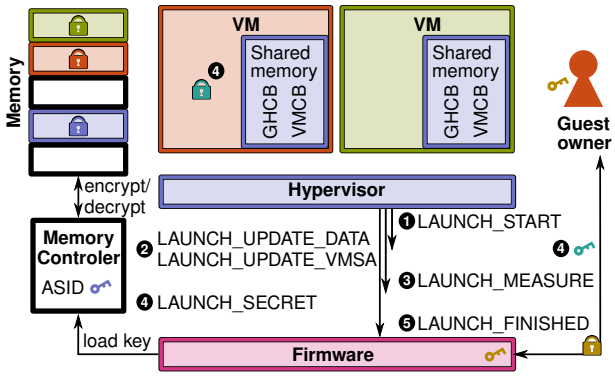


Figure 3.3. The remote attestation flow of AMD SEV

3.1.2.5 AMD SEV

AMD Secure Encrypted Virtualization (SEV) [58] allows isolating virtualised environments (*e.g.*, containers and virtual machines) from trusted hypervisors. SEV uses an embedded hardware AES engine, which relies on multiple keys to encrypt memory seamlessly. It exploits a closed Arm Cortex-v5 processor as a secure co-processor, used to generate cryptographic materials kept in the CPU. Each virtual machine and hypervisor is assigned a particular key and tagged with an Address Space IDentifier (ASID), preventing cross-TEE attacks. Code and data are protected by AES encryption with a 128-bit key based on the tag outside the processor package.

At its core, SEV leverages a Chip Endorsement Key (CEK), a secret fused in the die of the processor issued by AMD for its attestation mechanism. SEV may start the virtual machines from an unencrypted state, similar to SGX. In such cases, the secrets and confidential data must then be provisioned using remote attestations queries. The AMD secure processor cryptographically measures the content of the virtual machine into a launch digest (*i.e.*, claim). While SEV and SEV-ES (SEV Encrypted State) [83] only support remote attestation during the launch of the guest operating system, SEV-SNP supports a more flexible model. That latter bootstraps private communication keys, enabling the guest virtual machine to request attestation reports (*i.e.*, evidence) at any time and obtain cryptographic materials for data sealing, *i.e.*, storing data securely at rest (Figure 3.3).

The approach increases the attack surface of the secure environment since the TCB is enlarged. The guest operating system must also support SEV, cannot access host devices (PCI passthrough), and the first edition of SEV (called *vanilla* in Table 3.1) is limited to 16 virtual machines. Future iterations of SEV Secure Nested Paging (SEV-SNP) [18], plan to overcome limitations, typically by means of in-silicon redesigns.

3.1.2.6 RISC-V architectures

There exist several proposals for TEE designs for RISC-V based on PMP instructions including support for remote attestation. We survey the most important ones in the following. Keystone [89] is a modular framework that provides the building blocks to create trusted execution environments, rather than providing an all-in-one solution that is inflexible and is another fixed design point. Keystone utilises a secure boot mechanism that measures the secure monitor image, generates an attestation key and signs them using a hardware-visible secret (*i.e.*, root of trust). The secure monitor exposes a Supervisor System Interface (SBI) for the enclaves to communi-

cate. A subset of the SBI is dedicated to issue evidences signed by provisioned keys (*i.e.*, endorsed by the verifier), based on the measurement of the secure monitor, the runtime and the enclave's application. As such, when a remote attestation request takes place, the remote party (*i.e.*, verifier) sends a challenge to the trusted application. The response contains the evidence with the public session key of the attester. Finally, the verifier asserts or denies the evidence based on the public signature and the measurements of components (*i.e.*, claims).

Sanctum [55] has been the first proposition with support for attesting trusted applications. A remote attestation protocol is proposed, as well as a comprehensive design for deriving trust from a root of trust. Upon booting the system, `mroot` generates the cryptographic materials for signing if started for the first time and hands off to the secure monitor. Similar to SGX, Sanctum owns a dedicated signing enclave, that receives a derived private key from the secure monitor for evidence generation. A regular enclave can request an evidence to the signing enclave based on multiple claims. This evidence is then forwarded to the verifier by the secure channel.

TIMBER-V [145] achieved the isolation of execution on small embedded processors thanks to hardware-assisted memory tagging. The trust manager exposes an API for the enclaves to retrieve an evidence, based on a given enclave identity, a secret platform key (*i.e.*, root of trust), and an arbitrary identifier provided by the trusted application. The remote attestation protocol is twofold: the remote party (*i.e.*, verifier) sends a challenge to the trusted application (*i.e.*, attester). Next, the challenge is forwarded to the trust manager as an identifier to issue an evidence, which is authenticated using a MAC. The usage of symmetric cryptography is unusual in remote attestation because the verifier requires to own the secret key to verify the evidence.

LIRA-V [132] drafted a mutual remote attestation for constrained edge devices. The protocol relies exclusively on machine mode (M-mode) or machine and user mode (M-mode and U-mode). The measurement of code (*i.e.*, claim) is computed on parts of the device physical memory regions by a program stored in the ROM. LIRA-V's mutual attestation protocol works similar to the protocol illustrated in TrustZone-A and is formally verified.

3.1.3 Takeaways from the Survey

This section presented and compared state-of-the-art remote attestation schemes, which leverage hardware-assisted TEEs, helpful for deploying and running trusted applications from commodity devices to cloud providers, in applications as seen in VEDLIoT. TEE-based remote attestation has not yet been extensively studied and seems to remain an industrial challenge. Whether provided by manufacturers or developed by third parties, remote attestation remains an essential part of the design of trusted computing solutions. This study sheds some light on the limitations of state-of-the-art TEEs and identifies promising directions for future work. Finally, this survey has been published in the 22nd international conference on distributed applications and interoperable systems conference (DAIS'22) [106].

3.2 Remote ATtestation procedureS (RATS)

In this section of the chapter, we present an ongoing work on PKI-integrated automated remote attestation of IoT devices following state-of-the-art attestation mechanisms. As mentioned earlier, Remote Attestation (RA) is a process where one trusted

entity (verifier) remotely verifies the software integrity of a device (attester). Procedures like over-the-air updates, device certification, vulnerability patching rely on RA to establish the integrity of the device they are communicating with. Remote attestation serves as a foundation for many security services besides certification, such as software update, system reset, and runtime software/device verification. The Internet Engineering Task Force (IETF) are working on an interoperable standard RA procedure; i.e., Remote ATtestation procedureS (RATS) [45]. RATS attests devices that implement Trusted Platform Module (TPM1.2 or TPM2.0). The device being attested (IoT device) provides TPM-generated information regarding the health of it's software using the quote mechanism. This evidence is verified by the attestation body using the the TPM event logs and pre-established proofs.

3.2.1 Trusted Platform Module

The **Trusted Platform Module (TPM)** is an international standard [12] by the Trusted Computing Group (TCG) to enable hardware Root of Trust (RoT). TPM 2.0 was created by TCG to provide flexible usage for the evolving IoT platforms: automotive, IIoT, smart homes, etc. The TPM establishes RoT in the underlying platform and provides secure storage, system's measurements and secure reporting. The TPM memory is shielded from access by any entity other than the TPM. The TPM provides assurance of the integrity of the software on device by taking *measurements*. An integrity *measurement* is a value that shows a possible change in the trusted state of the platform. Loading a new software, a system update or modification of configurations are examples of events that can trigger the TPM to take new measurements and record them in Platform Configuration Registers (PCR). The TPM maintains a log of all measurements and the log can be evaluated to determine the integrity of the measurements. The PCR values are hash-of-hashes and are hence dependent on the order in which the events are hashed. A PCR generated from three distinct events X, Y and Z will have a different value if the same events occurred in a different order (e.g., X, Z and Y or Z, X, Y).

TPM Quote provides a mechanism that the PCRs can be used by other devices to verify the integrity of a device in consideration. Any device that needs to perform verification needs to send a request preferably accompanied by a unique random nonce which adds freshness and prevents replay attacks. This nonce is combined with a PCR value and signed by the device being verified. This structure is called a quote and is used to verify the device's integrity. TPM 2.0 supports the TPM2_Quote command to generate a quote. It takes TPM2B_DATA which is the nonce, TPML_PCR_SELECTION containing the PCRs selected and returns the quote i.e., TPM2B_ATTEST and the signature TPMT_SIGNATURE. A **TPM Event Log** contains entries against every PCR measurement made. The Log contains the measurement hash and information regarding the event that triggered the measurement. The Event Log can be read by any verification process to verify the integrity of the PCR values. Event logs are stored in different formats, optimized to a particular environment they are generated in.

3.2.2 Time-Of-Check to Time-Of-Use (TOCTOU) Invalidity

Almost all standard attestation mechanisms like RATS suffer from the drawbacks of the Time-Of-Check to Time-Of-Use (TOCTOU) invalidity [112]. It remains a concern for the device owners and users that the attested state of the device for which the certificate was issued may have changed during the interval between attestation and

device usage. Since the certificate reflects an attested software stack of the device, if the attested state of the device has changed due to exploitation of a vulnerability or addition of new software, the certificate is misleading. Over the years, a few solutions have been proposed to counter the TOCTOU problem. We are designing an automated remote attestation and certification scheme building on standards like RATS for remote attestation to eliminate the TOCTOU problem in certificates. We propose the integration of its processes into PKI (utilizing X509v3 Certificates).

3.3 WebAssembly Runtime with Remote Attestation for TrustZone

Security is critical when designing and deploying distributed applications for mutually untrustworthy stakeholders, *e.g.*, developers, data providers and hosting companies. The problem grows in complexity when one considers decentralised operations of heterogeneous IoT, edge and cloud devices, all at risk of being compromised.

Trusted execution environments (TEEs), *i.e.*, Intel SGX [53] or Arm TrustZone [116], offer hardware support for securely executing applications in shielded environments, *i.e.*, enclaves. While TEEs are promoted by the commercial offerings of major cloud providers, they do not guarantee that the code itself is trustworthy and has not been tampered with. Remote attestation [111, 48, 62, 77, 86] is typically used to assess the code before its execution. However, while this key security feature is provided by some TEEs, *e.g.*, Intel SGX [124], Arm TrustZone lacks built-in support for remote attestation. Given the swift growth of popularity of Arm-based architectures in the IoT edge computing [11] and its recent adoption in the general computer market [27], this is concerning. In addition, recent attacks have shown how to compromise IoT devices via malicious firmware updates [96] or software flaws [95], typically beyond the threat model of TrustZone. Nonetheless, the inability of manufacturers to verify the authenticity of their software upon execution makes these platforms inadequate for handling sensitive tasks and data, in particular when dealing with recent scenarios such as trustworthy machine learning systems at the edge [91].

WebAssembly (Wasm) [72] is a new bytecode standard for near-native speed applications. It enables developers to build their software components with the productivity benefits of modern programming languages while supporting legacy code, since modern compilers support Wasm [88]. TrustZone is a constrained environment that runs small executables using a specialised API. Wasm fits well in this model, thanks to its small runtime overhead, portability, supporting non-standard system interfaces, and fast execution as its bytecode can be compiled ahead-of-time. Embedding Wasm in TrustZone can be helpful in many use cases. The smartphone industry could rely on Wasm as an interoperable bytecode to execute secure applications inside TrustZone, as exclusively reserved for the manufacturer's needs or for strategic partners [123]. Automotive applications could rely on IoT devices to run machine learning algorithms inside enclaves, ensuring the validity of the results via attestation and using Wasm to leverage legacy machine learning frameworks.

In this section we present our work to develop WaTZ, an efficient and secure runtime for trusted execution of Wasm code inside TrustZone, adding support for remote attestation. We leverage the sandbox isolation of Wasm to mitigate, and possibly prevent, vertical privilege escalations and lateral attacks. We combine TrustZone with Wasm bytecode to issue trustworthy evidences, attesting the genuineness of running software. In Section 3.3.3, we adapted and fully implemented the remote attesta-

tion protocol from SGX [80], using a public-key infrastructure. As such, we facilitate the deployment of fully decentralised applications spanning various devices at the core or the edge of the network. Wasm extends OP-TEE [93], a popular open-source trusted OS, and we validated our prototype with hardware Arm boards.

The main research questions that WaTZ intends to answer, and the main contributions of this work are the following:

Are there system challenges when embedding Wasm into Arm TrustZone? TrustZone requires using a trusted operating system, which declares non-standardised API to interact with system resources. This, coupled with the constrained nature of TEEs (*e.g.*, no system call, limited memory), increases the complexity of hosting general-purpose applications compiled in Wasm. We show (Section 3.3.2) that WaTZ is the first system to run Wasm applications in TrustZone, while leveraging WASI standard, a POSIX-like layer for Wasm, to interact with the TEE facilities. Our WaTZ prototype supports a subset of WASI API to evaluate a database engine (SQLite [82]) and a machine learning library (Genann [3]).

How can relying parties trust the remote execution of Wasm applications? Given the lack of built-in remote attestation in TrustZone, we propose a protocol (Section 3.3.3) to attest Wasm code embedded in our trusted environment. We identify the hardware requirements of IoT devices to provide attestable guarantees (*e.g.*, a root of trust, secure boot), showing how they can be combined with the trusted environment to verify Wasm binaries. We contribute an extension of the WASI specifications, called WASI-RA, to enable the hosted Wasm applications to attest against trusted parties and communicate shared secrets and confidential data, based on Intel SGX's remote attestation protocol.

The remainder of the section is organised as follows. We first present in Section 3.3.1 some background information and related work. Section 3.3.2 introduces the overall design and architecture of the WaTZ runtime. The remote attestation mechanism of WaTZ is described in Section 3.3.3 and we wrap-up the discussion in Section 3.3.4.

3.3.1 Background and Related Work

Our WaTZ runtime supports trusted execution of Wasm code inside TrustZone with remote attestation mechanisms. This section briefly introduces the underlying technologies and highlights how our approach improves related work.

Arm TrustZone:

TrustZone provides the hardware elements enabling TEEs on Arm processors [109]. OP-TEE [93] is a popular open-source runtime environment with native support for TrustZone. It offers a developer-friendly setup [69] to build trusted applications (TA). OP-TEE follows the TEE architecture and API standardised by GlobalPlatform (GP API) [4], built around three components: a client application, a dedicated Linux driver and the OP-TEE OS. The OS of the normal world is referred to as Rich Execution Environment (REE). The *host* application runs in the normal world, as a client of a Trusted Application (TA) in the secure world. Host applications leverage *client* APIs.

WebAssembly: WebAssembly [72] is a W3C open standard for a portable, compact, low-level, stack-based binary code format. Initially focused on browser-embedded applications, the specifications allow for standalone execution running outside

	AOT	WASI	RA	RA in WASI	μRT	IoT TEE	TEE(s)
Twine	✓	✓	✗	✗	✓	✗	SGX
Veracruz	✗	✓	✓	✗	✗	✗	Nitro, CCA
Enarx	✗	✓	✓	✗	✗	✗	SGX, SEV
AccTEE	✗	✗	✗	✗	✗	✗	SGX
Se-Lambd	✗	✗	✓	✗	✗	✗	SGX
Teaclave	✗	✗	✓	✗	✓	✗	SGX
WaTZ	✓	✓	✓	✓	✓	✓	TrustZone

Table 3.2. Comparison of the related work features

browsers as well. We exploit the sandbox capabilities from software fault isolation [143] and control-flow integrity [14] to isolate Wasm code from the trusted OS. We also leverage WebAssembly System Interface (WASI) [9], a POSIX-like interface used by Wasm programs to interact with the underlying OS.

WebAssembly and TEEs: Few options exist to host Wasm applications inside TEEs. Twine [102], an embedded trusted runtime for Wasm, executes Wasm applications inside Intel SGX enclaves. Twine relies on the WebAssembly Micro Runtime (WAMR) [8], enabled with WASI to interact with a secure file system. Enarx [2] targets Intel SGX enclaves and AMD SEV virtual machines. Veracruz [7] only supports VM-based TEEs, such as Arm CCA [32] and AWS Nitro [1] enclaves, having recently dropped SGX and TrustZone enclaves considered too constraining [6]. AccTEE [66] and Se-Lambda [118] run Wasm binaries in Intel SGX enclaves using the V8 JavaScript/Wasm engine. Table 3.2 compares WaTZ against state-of-the-art TEE runtimes for Wasm along the following dimensions: *AOT* (process ahead-of-time compiled Wasm bytecode), *WASI* (enable system interaction), *RA* (support remote attestation), *RA in WASI* (provide a WASI API to control the remote attestation), μ*RT* (use a lightweight runtime, which is <1 MB in memory), *IoT TEE* (designed for IoT devices), and *TEE(s)* (summarises the TEE technologies).

3.3.2 WaTZ: System Overview

This section introduces WaTZ's threat model, design, architecture, and details of the trusted runtime.

3.3.2.1 Threat model

We consider the following aspects.

(a) Hardware. WaTZ leverages the following hardware capabilities: *(i)* TrustZone security extensions, *(ii)* a root of trust, and *(iii)* secure boot. While we consider a powerful attacker with physical access to the devices, we assume that the protections offered by hardware cannot be subverted. We consider storage rollback attacks out of scope, which can be mitigated using hardware monotonic counters [100].

(b) Secure world. We assume the bootloader and trusted OS do not contain vulnerabilities enabling an attacker to breach the TEE. The cryptographic primitives and algorithms are considered correct. Code and data inside the secure world are trusted and cannot be accessed from the normal world, except through dedicated channels controlled by WaTZ. Finally, side-channel attacks [119, 73, 150] are out of scope.

(c) Normal world. We make no assumption regarding the normal world, which includes the rich OS and the user space. Compromised OSes may arbitrarily respond to trusted

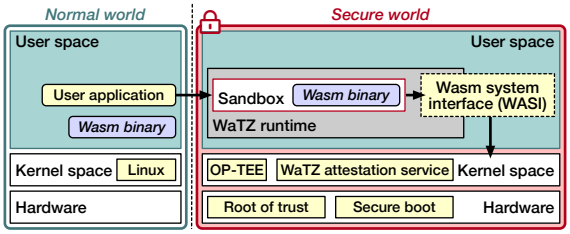


Figure 3.4. Overall architecture of WaTZ

OS calls, causing its malfunction. The trusted applications relying on the normal world should be carefully crafted to ignore abnormal responses or even abort execution in such cases.

3.3.2.2 Design overview

WaTZ is a trusted runtime to execute Wasm applications with remote attestation capabilities. Figure 3.4 illustrates its components. Its small footprint (265 kB on disk, with the runtime and WaTZ's components) brings several advantages. Smaller programs offer smaller attack surfaces. TEEs and small devices (edge or IoT) are usually tight in memory. Its small footprint allows deploying and running a complete Wasm virtual machine inside TrustZone using a small edge-scale Arm processor. We estimate that the increase of the trusted computing base (TCB) due to the embedding of the Wasm runtime in the TEE is outweighed by its benefits.

To provide trusted OS features to Wasm applications, we contribute an adaptation layer that binds WASI to the API existing in the trusted environment (*i.e.*, GP API in our prototype). This, in conjunction with the reliance of Wasm applications on WASI, allows WaTZ to run unmodified Wasm applications, while benefiting from a TEE.

To offer remote attestation, we designed and implemented WASI-RA, an extension of WASI enabling the hosted Wasm applications to interact with the process of attestation. We guarantee that WaTZ is booted correctly and was not tampered with using a secure boot system. When WaTZ loads a Wasm application, its bytecode is stored in the secure memory and measured to produce a hash. Using our interface, a Wasm attester can request WaTZ to generate trusted evidences based on the hardware secret and the hash of the Wasm bytecode. Furthermore, WaTZ includes a remote attestation protocol to enable a third-party verifier to check if the evidences are genuine, which relies on the measured fingerprint of the Wasm applications. Upon positive attestation, our protocol simplifies the establishment of a hybrid cryptosystem for the verifier and the attester, which can be later used to create secure channels.

3.3.2.3 Embedded runtime

We built WaTZ as an embedded Wasm runtime with a WASI interface as this design provides several advantages, *i.e.*, removing some barriers for building TAs. First, WaTZ opens the choice for programming languages. Provided that the compiler can emit Wasm bytecode and support WASI, it is a clear advantage over vanilla bare OP-TEE, which limits developers to C only. Second, this hides the complexity of writing code dedicated to OP-TEE since WASI abstracts the implementation details of GP API. Furthermore, WebAssembly separates the virtual address spaces used for Wasm applications and the native runtime process, and code, stacks and heap are handled separately, making memory-oriented attacks or developer mistakes more unlikely. Besides, WASI ensures that the applications do not harm the secure world and acts

as a gatekeeper to run operations outside of the virtual machine. Finally, applications are not tightly coupled to the underlying TEE, and WaTZ can load any regular Wasm/WASI application without changes. As a result, Wasm brings more flexibility, versatility and security compared to their native counterpart, allied with its strong sandboxing mechanism that isolates each hosted Wasm application. This newly introduced isolation layer extends the single trusted world of TrustZone to an environment with multiple secure and mutually distrusting enclaves.

Instances of trusted Wasm applications are started by a user space process in the normal world, which uses the standard TEE API to prepare a buffer containing the Wasm application and trigger WaTZ in the secure world. Once in the secure world, WaTZ copies the bytecode into a secure, sandboxed memory, calculates the hash for future attestation and starts the execution immediately. The Wasm applications natively use the standard WASI interface to interact with the OS, which, in turn, diverts the calls to WaTZ. We implemented a new kernel module for OP-TEE, *i.e.*, the WaTZ *attestation service*, enabling the runtime to generate evidences for relying parties, in order to prove the authenticity of the applications. While several open-source alternatives exist to execute Wasm code, we settled on WAMR [8], a lightweight and embeddable runtime implemented in C, ideal for TEEs in general (small TCB) and OP-TEE in particular (*i.e.*, TA written in C).

3.3.2.4 Execution modes

WAMR can execute code in three modes, each with its benefits and drawbacks: interpreted, compiled just-in-time (JIT) and compiled ahead-of-time (AOT). Interpreted is the simplest yet slowest, as it does not require pre-processing the bytecode. When using JIT compilation, the bytecode is translated into native code whenever executed, but embedding a compiler in the runtime increases its complexity, size and dependencies. Indeed, WAMR uses LLVM [88] as its JIT and AOT compiler, which is not trivial to port to a restricted environment like OP-TEE. With AOT compilation, the bytecode is translated before execution, so the runtime does not need to include a compiler, but requires the TA to allocate executable memory. We opted for AOT compilation for WaTZ's runtime. However, OP-TEE's memory management API cannot modify the pages' protection to mark them as executable [5]. Hence, we extended the trusted kernel to provide such capabilities to TAs. According to our measurements, AOT execution is on average 28 times faster than Wasm interpretation.

3.3.3 Remote Attestation of WebAssembly

This section presents the remote attestation mechanism implemented in WaTZ, first by explaining how the hardware is trusted, then extending this principle to the secure OS.

Root of trust: We designed WaTZ for devices that expose a hardware root of trust to the secure world. We extended OP-TEE to deterministically derive a key pair from the hardware root of trust. Normal and secure OS can hence be updated without losing the key materials, and on-chip key generation guarantees that the private key never leaves the trusted kernel OS. The public key is then exported and used as an endorsement value to be verified during remote attestation requests. The key pair, *i.e.*, *attestation keys*, is at the core of WaTZ's mechanisms to provide attestation signatures and guarantee platform authenticity.

Secure boot: Secure boot is a security mechanism to ensure that the device is booting in a trusted state. WaTZ requires the device to implement secure boot, so only a trusted entity is able to provide software to boot the secure world (*i.e.*, bootloaders, trusted OS). Therefore, this enforces a *chain of trust* that protects the attestation keys from being extracted from the secure kernel OS. The boot sequence is as follows: the first-stage bootloader (ROM) verifies if the provided second-stage bootloader is genuine, based on the public key stored in one-time programmable fuses (eFuses) [113]. The previous booting component recursively verifies the next boot stages until the secure world is fully booted.

Proof of trust: WaTZ generates cryptographically signed reports, *i.e.*, *evidences*, asserting that an executing Wasm application is trustworthy and the device genuine, by producing a hash of the Wasm AOT bytecode stored in the secure memory at launch time. We offer an API to Wasm applications to issue evidences and establish a secure communication channel with a verifier. Then, the evidence is checked by the verifier using the corresponding public key of the device and verifies the code measurements to match with its reference values.

The evidence is created by interacting with the attestation service, implemented as a kernel module in OP-TEE (shown in Figure 3.4). The evidence includes: *i)* an anchor, which is a value defined by the transport layer to bind security parameters to a particular session (*e.g.*, a public session key); *ii)* the version of WaTZ, enabling the relying party to exclude outdated systems; *iii)* the claim, *i.e.*, the hash of the bytecode; *iv)* the public key of the attestation service, for the verifier to determine if the device is endorsed, and *v)* the digital signature of the evidence.

Security requirements: Our remote attestation protocol satisfies the following security requirements:

1. *Mutual key establishment*: A shared secret key is established for communication between the attester and the verifier, using the elliptic-curve Diffie–Hellman ephemeral (ECDHE) key-agreement protocol.
2. *Mutual entity authentication*: The attester and the verifier are mutually authenticated to prevent masquerading attacks. From the attester standpoint, the verifier's public key must be hardcoded into the Wasm application. This, combined with the application measurement, ensures that an attacker cannot change the key so that the software can only communicate with the intended remote service.
3. *Half trust assurance*: The attester attests the Wasm application and the platform integrity to the verifier. The verifier does not provide a similar proof to the attester, and the attester assumes the entity authentication is sufficient.
4. *Freshness*: ECDHE (*ephemeral*) requires the key pairs to be fresh, hence preventing replay attacks.
5. *Forward secrecy*: Compromised long-term secrets do not affect the security of earlier or future exchanges. Similar to *freshness*, ECDHE achieves this goal, which means the keys are renewed for every instance of remote attestation.

WaTZ protocol for remote attestation: The GP API defines an interface to establish a secure communication channel using TLS. However, OP-TEE [65, 94] lacks the corresponding implementation. We extended and implemented the RA protocol of Intel SGX [80] (in turn inspired by SIGMA [87]) to rely on TLS. We changed the protocol compared to the original by various aspects: *(i)* removed the SGX specificities, such as

the interaction with the quoting enclave, as the kernel module of WaTZ provides the measurements, *(ii)* merged the two first messages to communicate from the client to the server as they tightly relate, *(iii)* provided a fixed structure for the last message to seamlessly handle the confidential data, eliminating the burden of the hosted Wasm application from decrypting that content, *(iv)* omitted the Intel SGX Enhanced Privacy ID (EPID) for conciseness, and *(v)* removed the dependency on Intel's PKI, since the device's key pair is emitted by WaTZ based on the embedded root of trust. Below, we detail each protocol's message and the required cryptographic operations.

(a) Message 0 (attester $\rightarrow$ verifier): The attester generates a session key pair $< a, G_a >$ and sends the public part G_a.

(b) Message 1 (attester $\leftarrow$ verifier): Upon reception of msg_0, the verifier generates a session key pair $< v, G_v >$. It computes the shared secret from the public session key of the attester G_a and its private session key v, which gives G_{av}. This shared secret is derived into a *key derivation key* (KDK), which is further derived into two shared secrets: K_m for calculating MACs and K_e to encrypt future messages in the session. These derivations are the same as in Intel SGX [80]. A reply message is sent to the attester, containing G_v, the verifier's ECDSA public key V (its identity), and a signature of both public session keys. The message is appended with its MAC.

(c) Message 2 (attester $\rightarrow$ verifier): The attester verifies the signature of the public session keys: different session keys may reveal a masquerading or replay attack, and verifies the MAC of msg_1. It also checks whether the service public key V matches the hardcoded key in the Wasm application. Doing so ensures the attester communicates with the intended service and prevents an attacker from altering that key as it is part of the code measurement. The attester computes the shared secret from the public session key of the verifier G_v and the private session key a, which gives G_{va} that is equal to G_{av} computed by the verifier. The key derivations follow the same process as in msg_1. The attester creates msg_2 by concatenating its public session key G_a with a newly generated evidence signed by the attester A, where the anchor of the transport layer is the hashed concatenation of the public session keys. Finally, it appends a MAC.

(d) Message 3 (attester $\leftarrow$ verifier): The verifier checks the MAC of msg_2 and verifies that G_a matches the one received in msg_0. It also verifies whether the anchor corresponds to the public session keys, revealing a masquerading or replay attack. It extracts the evidence's public attestation key and checks against its list of endorsed public keys to determine whether this is a known device. If the key is found, the digital signature of the evidence is checked, which indicates whether the hardware is genuine. Finally, to verify that the Wasm application is trustworthy, its code measurement claim is compared with a list of possible reference values. If all verifications pass, the protocol sends msg_3 with the confidential data, encrypted with AES-GCM, which requires iv (an initialisation vector).

We simplified the protocol by omitting the use of session identifiers. Such identifiers are needed for having multiple sessions with concurrent remote attestation requests. We also reduced the complexity by keeping the evidence in clear. If the secrecy of this structure is a concern, the protocol can be extended to protect the evidence using AES-GCM.

3.3.4 WaTZ Wrap-up

To the best of our knowledge, WaTZ is the first Wasm runtime running entirely inside Arm TrustZone with full support for remote attestation, optimised explicitly for Wasm to establish trust on hosted applications. Our design was implemented using a standard development board found in the market, verified and tested. With many different experiments, we measured the costs of using the specific mechanisms needed by WaTZ. We were able to demonstrate that our implementation is lightweight, achieving results comparable with Wasm running without a trusted environment. The complete results of our performance evaluation have been published in the 42nd international conference on distributed computing systems conference (ICDCS'22), including many design and implementation details from this deliverable, a security analysis and a formal verification of the remote attestation protocol [107]. The code and experiments are available through GitHub [105], opening the possibility to freely reuse it in decentralised applications on edge and IoT devices.

4 Embedded Trusted Execution Environments

This chapter is divided into two major sections describing our work on emerging Trusted Execution Environments (TEE) Technologies: **TrustZone-M**, ARM TrustZone's security extension for resource-constrained devices and **RISC-V**, the emerging open-source ISA. The work on state-of-the-art TEEs includes identifying serious vulnerabilities in TrustZone-M and proposing efficient solutions. Since TrustZone-M is now available for the Cortex-M series (which is leading the networked embedded devices market), this research improves the security for off-the-shelf IoT devices in the market. The chapter is divided into three sections: the first section describes the work done to establish a secure communication framework on TrustZone-M based IoT, the second section discusses the proposed mechanism for mitigating unauthorized activities within the secure world of TrustZone-M enabled devices and the last section of the chapter discusses an early work on the RISC-V platform which is a novel proposal intended to pave the ground for advancements in real-time operating systems and open-source CPUs (RISC-V).

4.1 Utilizing Arm MPU for secure communication in TrustZone-M

The TrustZone technology is a security extension incorporated into Arm processors which is explained in detail in Chapter 3.TEEs in general are essential for security-critical operations such as software/firmware update, remote attestation, handling sensitive data-generating peripherals, etc. Although TrustZone-M provides a hardware-based TEE, which effectively isolates security-critical operations from untrusted software components, it lacks mechanisms for secure communication between the TEE and the untrusted environment in an IoT device.

This section of the chapter describes a secure communication framework supported from the TrustedFirmware-M implementation of TrustZone-M. The proposed framework to build secure communication between the two worlds of TrustZone-M is a lightweight message protection scheme using the Memory Protection Unit (MPU). ShieLD is the first framework proposed to solve the issue of vulnerable inter-world communication for TrustZone-M platform. We present a real-world use case of the proposed mechanism along with an extensive micro-benchmark performance evaluation of it's overhead on system operations. Major content of this section is extracted from our publication "ShieLD: Shielding Cross-zone Communication within Limited-resourced IoT Devices running Vulnerable Software Stack" published in IEEE Transactions on Dependable and Secure Computing [85]. The related literature and extensive evaluation of the security attributes and functional overhead is presented in the paper as well.

The following sections provide: (i) background, (ii) ShieLD mechanism, (iii) implementation and evaluation and (iv) future directions of this work.

4.1.1 Background Technologies

TrustZone-M is designed for Cortex-M, which are a series of microcontroller processors that are programmed either bare metal (without libraries) or linked with some libraries that could provide OS-like features. Cortex-M based devices support the T32 instruction set, which is a subset of the A32 instruction set. The processor does not of-

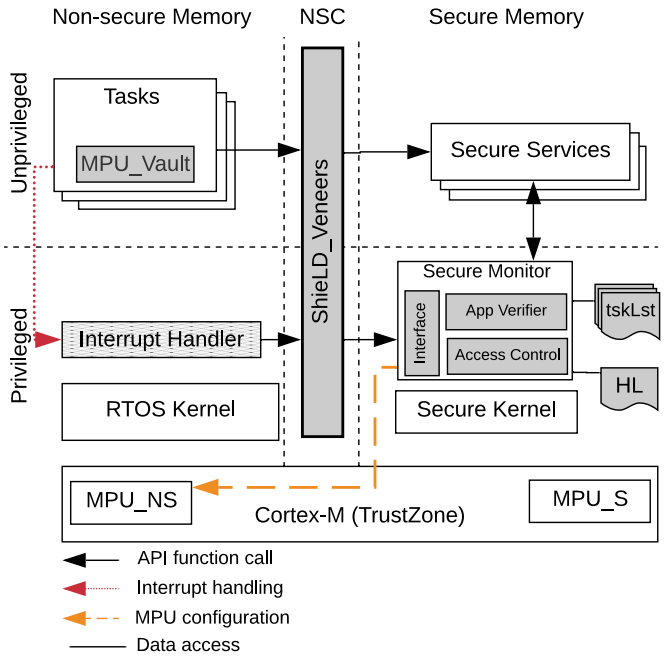


Figure 4.1. High-level architecture of ShieLD

fer complex memory management (no Memory Management Units), cache and often no FPUs (Floating Point Unit) either. TrustZone-M provides similar hardware-based isolation guarantees as conventional TrustZone for Cortex-A family (TrustZone-A) but, unlike TrustZone-A, the context switch between the two worlds depends on the setup of the memory map, without the need to enter the secure monitor mode. This design feature makes TrustZone-M energy-efficient, hence suitable for low-powered IoT devices. Saving and restoring the system context before and after the transition is handled by the Secure world.

All Cortex-M processors except Cortex-M0 have a **Memory Protection Unit**, which is a programmable block inside the processor that can be used to restrict access to a memory region by dividing the entire memory space (including Flash, SRAM) into a number of MPU regions and assigning access permissions to each region. The MPU can be configured to support 8 or 16 regions by privileged software using a series of 32-bit memory mapped registers. Privileged software such as an OS kernel, can change the MPU region setting at run-time based on the process being executed.

4.1.2 ShieLD Mechanism

Figure 4.1 depicts the high level architecture of ShieLD, highlighting its components and basic interactions. The Normal world runs the applications and a modified version of an interrupt handler on top of a Real-Time Operating System (RTOS) kernel. The Secure world runs the ShieLD components that are designed to implement security mechanisms. A small underlying secure kernel provides basic OS functions for software running in that world e.g., process management, file access, and memory management.

The secure monitor protects access to a protected memory region, i.e., MPU_Vault for secure cross-world message transmission. The MPU_Vault setup is initiated by the tasks in the Normal world. A task that needs access to a service in the Secure

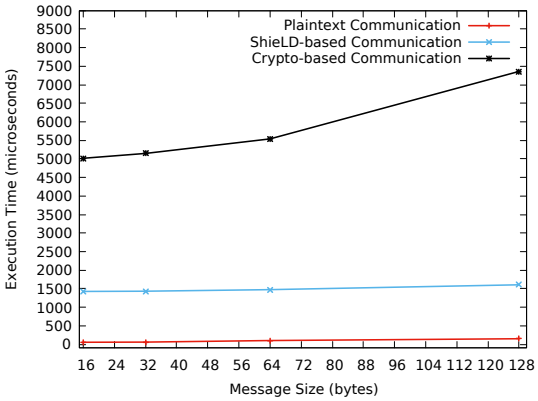


Figure 4.2. Execution time (in microseconds) of ShieLD-enabled communication compared to crypto-based and plaintext cross-world communication

world, first allocates an optimal block of memory area in the Normal world, and then sends a request to the secure monitor for its MPU_Vault setup. When the secure monitor receives a request for MPU_Vault allocation from a task in the Normal world, it verifies the legitimacy of the requesting task using an access control list (generated by ShieLD) containing tasks and the secure services they are allowed to access. After the authentication of the task, the secure monitor configures an MPU-protected MPU_Vault associated with the request and sets access permission of that region to Read/Write (RW). The address and region number of the MPU_Vault are then saved for future reference and a flag is set, indicating an MPU_ Vault is currently enabled.

The MPU_Vault is protected throughout the lifecycle of the interacting applications. In order to protect the contents of the MPU_Vault during system interrupts, we modify the Interrupt Service Routines entry and exit to redirect the control flow to secure monitor. The secure monitor changes the access permission of the MPU_Vault in use to be not readable and writable. It is also important to guarantee the integrity of ShieLD components in the system. The integrity of kernel code which is part of the static region of RTOS kernel, is protected during the secure boot. In TrustZone-enabled systems, when the device boots up, the Secure world is booted first, which later transfers control to the Normal world. Before passing the control to Normal world, ShieLD verifies the integrity of the kernel and enables MPU protections for its static region. Moreover, since the MPU is programmable by privileged software (including the RTOS), the MPU_NS is set up to be only writable/programmable from within the Secure world in ShieLD.

4.1.3 Implementation and Evaluation

We implemented a proof-of-concept of ShieLD building on TrustedFirmware-M (TF-M) [37] in the secure side with CMSIS RTOS2 as Normal world OS. TF-M provides a reference implementation of Secure world software for ARMv8-M [36]. It creates the foundations of TEE by providing a set of secure run-time services such as secure storage, cryptography, attestation etc. Additionally, secure boot in TF-M ensures integrity of run-time software.

The performance of ShieLD was evaluated on the Musca-A2 Test Chip Board by Arm [38]. The Musca-A2 board implements the Arm CoreLink SSE-200 subsystem featuring dual-core Cortex-M33 with CPU0 enabled at 50MHz [35]. We compare the communication overhead of ShieLD with plaintext and a state-of-the-art crypto-based

communication. This evaluation benchmarks the overhead of ShieLD on resource-constrained IoT devices. Figure 4.2 shows the comparison between these three communication modes for different message sizes. In the case of crypto-based communication, we see a visible escalation in total execution time with the increase in message size, this is due to the fact that time consumed by cryptographic operations is directly proportional to the size of the message being encrypted. In contrast, the ShieLD-protected communication has insignificant overhead with the increase of message size. It is an important feature of MPU-based protections that the execution time of setting up and protecting a region remains independent of the size of the message/region.

4.1.4 Summary of ShieLD

This work is a novel use of MPU to enable a secure vault that is exclusively accessible to the legitimate application in the Normal world that wants to access and execute security-critical operations in the Secure world. ShieLD provides similar security services (authentication, confidentiality, and integrity) as provided by the conventional crypto-based secure communication. We have implemented ShieLD in a TrustZone-M enabled IoT device and evaluated its memory and execution time (that translates to power/energy) overhead. Our empirical evaluation shows that ShieLD is extremely efficient when compared with the crypto-based communication protection. Though ShieLD targets IoT devices featuring TrustZone-M, the techniques proposed here could be extended to other TEEs.

4.2 Mitigation of Unauthorized Activity in TrustZone-M's Secure World

As mentioned previously in the chapter, Trusted Execution Environments (TEEs) like Arm TrustZone [30], Intel SGX [22], KeyStone [90] provide a mechanism to partition system resources and peripherals into secure and non-secure processing environments to enable isolation of critical components from the rest of the system. The way TrustZone-M is configured, system components in the secure areas have access to the entire system resources. The challenge here arises when a secure application is compromised (due to an undetected vulnerability), it becomes extremely challenging to detect this compromise and deploy remedial actions. Several examples of stealthy rootkits exploiting compromised TEEs [64, 110] exist. The fundamental problem is that in a multi-vendor environment, the device users have to trust that software in secure areas of the device is trustworthy and will not abuse the privileged access to manipulate or exfiltrate data. Industrial espionage in case of conflict of interest between vendors, illegal file sharing using compromised secure areas are some potential ways TEE-enabled systems have previously been exploited. In case of conventional IoT devices including smartphones, smart TVs, smart vehicles, the component vendors and suppliers are fewer and well-known, and the security of the underlying architecture and applications is matured over years of research and development. On the other hand, IoT device vendors are still emerging rapidly; resulting in spread of unregulated IoT devices. Running conventional anti-virus systems to monitor the secure application and the peripherals they access increases the code-base of secure areas and is not recommended.

We present TEE-Watchdog, a framework to map user/vendor-defined policies to the system memory, efficiently detect access violations and register policy-violating ap-

plication's behaviour. TEE-Watchdog ensures: (i) fine-grained access control over secure system peripherals, (ii) immutability of proposed system components and (iii) confidentiality and integrity of the associated policy files.

The related work, details about the design and the elaborate evaluation of the proof-of-concept implementation are presented in our manuscript "TEE-Watchdog: Mitigating unauthorized activities within Trusted Execution Environments in Arm-based low-power IoT devices" published in Security and Communication Networks. The following sections provide brief descriptions of the (i) Threat model and (ii) TEE-Watchdog Mechanism.

4.2.1 Threat Model

We consider that our system runs on resource-constrained IoT devices that support Arm TrustZone-M. Our security guarantees hold with the assumption that TrustZone itself is implemented correctly and no intentional flaws and bugs are introduced in it. We trust the privileged firmware and secure bootloader. We assume that Stack Limit registers are used appropriately to prevent stack manipulation and the non-secure Interrupt Service Routines (ISR) cannot interrupt secure ISRs. Moreover correct usage of Stack Limit registers ensures that secure ISRs are handled by privileged software. Moreover, we assume there are no validation bugs in the secure software that can lead to privilege escalation to the privileged firmware level.

We define and distinguish three classes of attackers in the ***threat model***. First, a trustworthy vendor could run an insecure third-party's piece of code/API in TEE unknowingly; this could be due to leaving known vulnerabilities in the code from the beginning, or code becoming vulnerable over a passage of time. This allows any attacker **(A1)** to exploit the known vulnerabilities. Second, the vendor itself is semi-trusted for being *honest-but-curious* and is able to access other applications data in the secure world **(A2)**. Third, a large number of IoT devices are now being manufactured by unknown and untrusted vendors who also don't have the mechanisms to maintain a secure supply-chain [1]; such untrusted IoT vendors and/or individual component suppliers could behave maliciously to collect critical end-user data **(A3)**.

Our ***attack model*** is based on the primary assumption that a service running in the secure world has autonomous access and control over the entire system resources (both normal and secure). So an attacker with access to the network stack can exfiltrate data of another service over the internet. The goal of such an attacker could be data leakage due to conflict of interest between manufacturing vendors (A2). Similarly, other attackers, like A1 and A3, can exploit the privilege level of their code in TEE which can access structures and functions.

4.2.2 TEE-Watchdog Mechanism

The system architecture of an IoT device with TEE-Watchdog is illustrated in Figure 4.3 that depicts 3 entities: an *IoT device*, an *IoT device peripheral* and an *external audit service*. We introduce TEE-Watchdog as a part of the secure kernel which is a privileged secure software. The application/service vendors provide a software's Manifest File enlisting functional details about the software, its sub-modules and required access to system peripherals. The software can be any user-level application or firmware of a peripheral. This **Manifest File** is converted to a memory-mapped **Access Table** at

[1] https://blog.checkpoint.com/2017/03/10/preinstalled-malware-targeting-mobile-users/

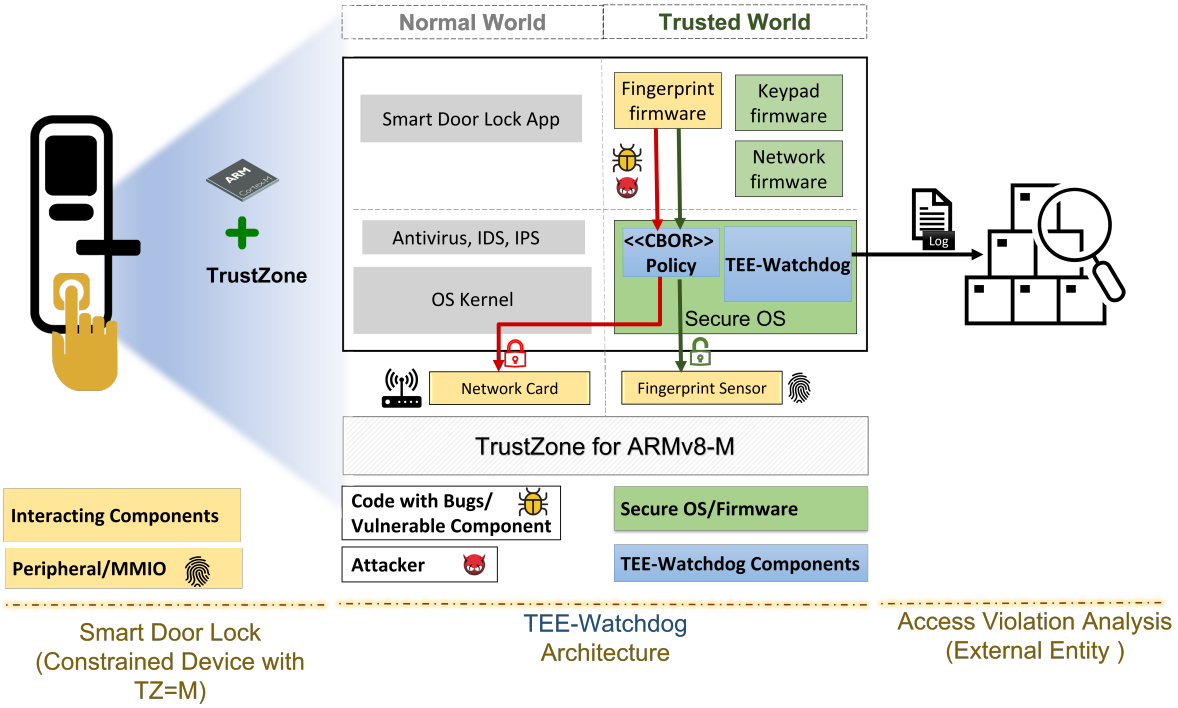


Figure 4.3. An overview of TEE-Watchdog security mechanism and its interaction with existing IoT system

system boot. The permitted peripherals are recorded in the Access Table where each application is listed along with its set of permitted peripherals. All system peripherals besides those enlisted are not permitted by default.

Cortex M23 and Cortex M33 have upto two MPUs if TrustZone security extension is enabled, we leverage the secure MPU to protect system resources in the secure world including code, data, MMIO regions and other system structures. When a secure service is invoked by a normal world application, it becomes the current active application/service in the secure world. Based on this active service in the secure side, the TEE-watchdog configures the entire secure memory space as per the *Access Table*.

The Audit Module is activated only when a service/application behaves outside its specified permissions and attempts to access memory or peripherals beyond a permitted list. Prior to this, the secure memory is divided into MPU protected regions by the Sandboxing Module with permissions configured according to the Access Table. During such a violation, the service making an illegal access to a secure peripheral would trigger a MemManage Fault. As soon as the processor triggers the fault, it populates the MemManage Fault Address Register (MMFAR) with the memory address of the resource being accessed, sets the MMARVALID bit in the MemManage Fault Status Register (MMFSR) indicating that MMFAR holds a valid fault address and sets the bit (bit 0 to 7) in MMFSR corresponding to the type of access that generated the fault. We modify the MemManage Fault Handler to introduce the Audit Module. The Audit Module checks the `LogFileExists` flag before continuing to investigate the fault. If the MMARVALID bit is set, the Audit Module reads the address from MMFAR (as shown in Figure 4.4) and enters it to the Log File along with the details of the violation and the application's UniqueID. The Log File suffers the risk of being overpopulated since the device memory is constrained. At any point, the Log File can only

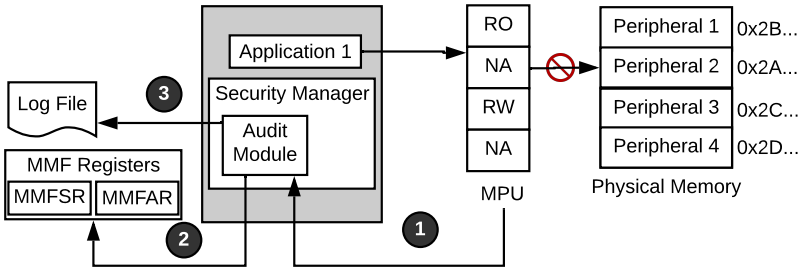


Figure 4.4. The Audit Module of the Security Manager performs behaviour logging of application deviating from their intended resource access

maintain a limited number of violation records. In order to ensure that all records of violation are stored, the Audit Module checks the number of existing log entries before making a new entry. If it is equal to `maxEntries`, the existing entries of Log File are moved for further processing to an external audit service and new entries can be stored on device. The Security Manager maintains a list of secure services and sets or clears the `isActive` flag based on the service that was invoked from the normal world or by another secure service. This ensures the link between the active application that caused the violation and the Log File entry. After the attempt is successfully logged, the control flow returns to the MemManage Fault handler and the fault is handled as per system settings.

4.2.3 Summary of TEE-Watchdog

TEE-Watchdog is a mechanism to restrict software access to secure system peripherals based on pre-defined security policies and permissions. TEE-Watchdog introduces a compact CBOR-encoded manifest file template for device vendors/manufacturers to use for specifying access policies. TEE-Watchdog also enables efficient behavioural logging of misbehaving software. TEE-Watchdog leverages the Arm MPU to create memory restrictions, uses the fault handling mechanism to log misbehaviour and utilizes standard CBOR encoding and decoding mechanism to parse the compact CBOR-encoded Manifest File. We have implemented TEE-Watchdog in a TrustZone-M enabled IoT device and evaluated its execution overhead and performance. Our micro-benchmark evaluation of the proof-of-concept implementation shows that additional operations introduced with TEE-Watchdog are at par with normal system operations. There is a 1.4% delay in latency of peripheral access due to TEE-Watchdog protections. Our proposed CBOR-encoded template for the policies has a 40% reduction in size of the manifest file as compared to the standard JSON file format which is a marginal gain considering the constrained nature of the IoT devices.

4.3 Memory Protection Unit for Open-source CPUs

The second part of this chapter discusses preliminary work and efforts on the novel RISC-V architecture. We have previously presented an implementation of RISC-V's Physical Memory Protection (PMP) and optimization techniques which reduce the LUT utilization of the RISC-V's PMP hardware on Artix-7 FPGAs by 50% relative to existing open-source alternatives (work done in M1-M8). We also demonstrated that certain features in the PMP specification are not required by modern embedded operating systems, and can be safely omitted for further efficiency gains. This was the basis of our implementation of supervisor PMP, a proposed RISC-V ISA extension for

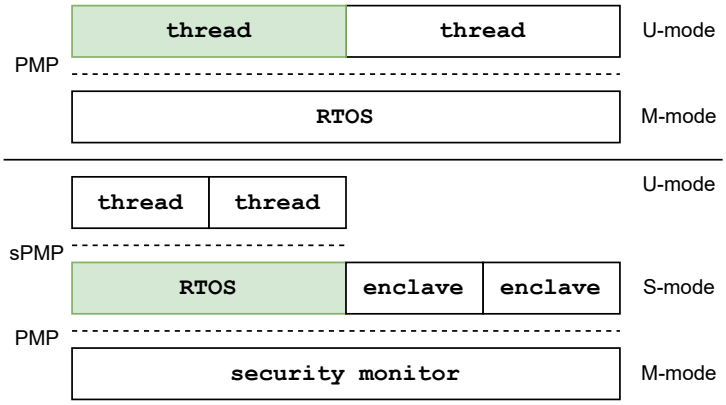


Figure 4.5. Above: Thread isolation with PMP. Below: RTOS virtualization with secure enclaves made possible by the sPMP extension

facilitating trusted execution on embedded devices. Our implementations and associated documentation are free and open-source plugins for the VexRiscv CPU project as mentioned in the previous deliverable. Our contributions have enabled secure enclaves on embedded soft processors.

We continued the work on RISC-V in months 9-18. Our target use case for the work was enabling userspace features in Zephyr RTOS on a soft CPU core. In our earlier work, we collaborated on the implementation of Zephyr's PMP driver in order to enable those features on RISC-V platforms. In the process, we observed a curious incompatibility between Zephyr's system call routine and a built-in RISC-V security feature, which effectively renders one PMP region unusable.

In contrast to other similar architectures (e.g., AArch32, ARC, etc.), RISC-V does not allow software to detect the current privilege level. Software can only determine which privilege level triggered the most recent exception by reading the mstatus CSR (which can only be done in M-mode). In theory, if software must know its current privilege level, it can make a system call to request that information from the kernel. This prevents untrusted code from discovering its own permissions.

This security feature can make porting software originally designed for other architectures somewhat challenging. Zephyr has a unified system call API for both privileged and unprivileged code. Every system call begins with a call to is_user_context(), which determines whether the method can be executed directly or if a context switch into kernel mode (M-mode) is required. Without hardware support for determining the current privilege level, we considered three workarounds to avoid significantly restructuring Zephyr's system call API:

1. Implement a dedicated system call. From the Interrupt Service Routine (ISR), the kernel reads mstatus.MPP to see whether U- or M-mode made the call.
2. Attempt a restricted operation, such as reading an M-level CSR. If an exception occurs, the CPU is in U-mode. If not, it is already in M-mode.
3. Reserve a single PMP region to protect a global read-only flag indicating the current privilege level (see Figure 4.5). The kernel updates this flag on every context switch.

Option (3) is currently the best option for resource-constrained RISC-V devices, and is now used in upstream Zephyr RTOS. The other options are unworkable because they incur a full context switch just to check the privilege level, which requires about 5000 CPU cycles.

This discrepancy is by no means a critique of either the RISC-V ISA or Zephyr RTOS. With some effort, Zephyr's build system could be restructured to accommodate RISC-V more appropriately by leveraging, for example, user-level interrupts to intercept system calls from userspace. We raise this discussion here merely because it has a real-world impact on the usability of PMP at the time of this writing, and because other RTOSes will likely face similar compatibility challenges.

There are currently two approaches under consideration in the RISC-V TEE WG to tackle the embedded enclave problem. The sPMP extension enables a memory hierarchy illustrated in Figure 4.5, which provides a relatively straightforward programming model for developers. One disadvantage to this approach is the need to reprogram the protection regions on every context switch between trusted and untrusted software. This may incur unacceptably high latencies for some use cases.

The RISC-V Trusted Execution State (TES) extension we previously proposed (in D5.1 [115]) would, on the other hand, allow each PMP region to be tagged with a trusted bit. These tags determine which execution state the region applies to, thereby negating any need to reprogram PMP on state transitions. TES has several other features catered specifically to creating enclaves which may make it more suitable than S-mode Physical Memory Protection (sPMP) for IoT applications. The future of this work could be the implementation of TES on VexRiscv for an evidence-based comparison.

5 Safety

Safety is the handling of all operations and events, within or outside of an industry, to protect the employees and users of the industry equipment and products. The main focus is on minimizing hazards, risks and accidents. In this chapter, we discuss safety for AI/ML systems and focus on the safety requirements methods, safety verification methods and safety runtime methods. This chapter takes an automotive perspective on the safety aspects of VEDLIoT because the automotive use case in VEDLIoT is most safety relevant of all the current use cases in the project.

5.1 Safety Standards

We begin our discussions from safety standards relevant to the use cases of the VEDLIoT project. For Electric/Electronic (E/E) systems and components, the safety standard IEC 61508 [52] may be used which covers the basic functional safety aiming at two fundamental areas:

- Safety life cycle: an engineering process based on best practices to discover and eliminate design errors.
- Failure analysis: a probabilistic approach to account for the safety impact of system failures.

IEC 61508 has led to the conception of standards specific to different industries. For example, safety for the automotive industry is managed by the ISO 26262 [81] standard. This standard is aiming at minimizing safety risks due to E/E faults. Another standard applicable to the automotive industry is ISO 21488 which is focused on the risk and accident mitigation with respect to the product's intended function. Both standards require a thorough analysis of the risks and hazards coupled to a development process based on the requirements methods, as proposed in the VEDLIoT project deliverable, D2.5 [138]. Work Package 2 is working on an Architectural Framework which guarantees a documented architecture and design description coupled with a verification process at all levels of the product design.

5.1.1 Functional Safety for Automotive, ISO 26262

This standard defines vehicle safety as the absence of unreasonable risk due to malfunctioning E/E components. An overview is illustrated in Figure 5.1. The standard identifies Hazard Analysis and Risk Assessment (HARA) as a necessary tool to identify the vehicle hazard levels. Based on this analysis the safety engineer will create safety goals and functional safety requirements. This can be further decomposed into the hardware and software development processes. Part 6 of ISO 26262 defines the V-model for the software development process. The V-model enforces that the architectural design and the function verification tests fulfill the safety argumentation. And that all the components, including input and output signals, at lower architectural levels interact so that the individual results generate the correct level of safety at the top level. ISO 26262 helps manage methods for fault detection and avoidance to minimize the risks.

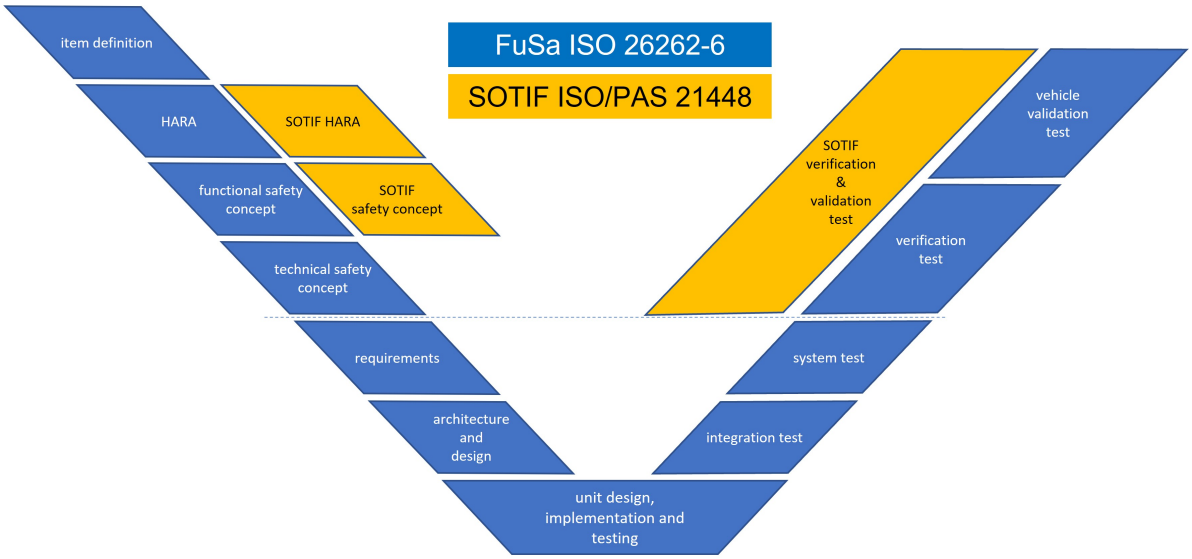


Figure 5.1. Overview of ISO 262626 and ISO/PAS 21488

5.1.2 Safety of the Intended Function for Automotive, ISO/PAS 21488

ISO 26262 is not well adapted to higher levels of vehicle automation. It does not manage faults occurring due to the inability of the sensors and similar component to correctly understand the environment. The ISO/PAS 21448 or SOTIF (Safety Of The Intended Function) [13] standard has divided the development process into four major phases: (i) design specification, (ii) development, (iii) verification, (iv) validation. These phases are processed multiple times in an iterative manner. The SOTIF standard assumes that there may be performance limitations of the processing components. The standard also discusses the problem with unsafe-unknown (no test and verification data available) and unsafe-known scenarios (the system is just outside operational boundaries), again trying to minimize risks in the operational system. The SOTIF standard also includes a dedicated HARA (Hazard Analysis and Risk Assessment) to find unacceptable performance functionality, reduced situational awareness, misuse and more.

5.1.3 Safety for AI/ML Systems

When we are using ML algorithms in the real-world domain, we need to consider trustworthiness, safety, and fairness prerequisites. There is an urgent need for corresponding techniques and metrics in some high-stake domains (e.g. the automotive domain). Although ISO 21488 does include argumentation for ML based systems, there are some obvious gaps for these system when it comes to safety. This has been investigated by other teams [103].

Varshney et al. investigated safety for ML models and compared it with four main engineering safety strategies in current industry [140]: (i) inherently safe design, (ii) safety reserve, (iii) safe fail, (iv) procedural safeguards. Salay et al. studied ISO 26262 part-6 with respect to safety of ML models. They identified that the fulfillment of safety methods for ML sofware compared to conventional software was reduced by 40% using the standard [122]. Amodei et al. identified five research problems that could result in unintended and unsafe behavior of real-world AI systems [19]. In a medium.com article, Ortega and Maini introduced three areas of technical AI safety as specification, robustness, and assurance [114]. In [104] the researchers put together

open challenges and following conclusions.

Design Specification: ML models learn the patterns in data to discriminate or generate their distribution for new unseen input and learn the target classes through their training data (and regularization constraints) and not using a formal specification. This may cause a mismatch between "designer objectives" and "what the model actually learned" which could result in unintended functionality of the system. The data-driven optimization makes it hard to define specific safety constraints. Seshia et al. looked at formal specifications for DNNs to lay an initial foundation for formalizing and reasoning about properties of DNNs [130]. Another way to manage the design specification problem is to break machine learning components into smaller algorithms (with smaller tasks) to work in hierarchical structures. Dreossi et al. presented the VerifAI toolkit for formal design and analysis of AI-based systems [61].

Implementation: ISO 26262 requires traceability from requirements to design and advanced ML models trained on high dimensional data are not transparent. Significant research has been performed on interpretability methods for DNN to provide instance explanations of model prediction and DNN intermediate feature layers [149]. However, the completeness of interpretability methods to grant traceability is not proven yet [16].

Testing and Verification: Significant verification of products are required for unit testing to meet the ISO 26262 standard. Coming to DNNs, formally verifying their correctness is challenging (provably NP-hard, see [130]) due to the high dimensionality of the data. Therefore, reaching a complete validation and testing bound to the operational design domain is difficult. As a result, researchers proposed new techniques such as searching for unknowns [146], predictor-verifier training [71], neuron coverage and fuzz testing [144].

Performance and Robustness: SOTIF standard treats the ML models as a black box and suggests using methods to improve model performance and robustness. Generalization error refers to the gap between model's empirical error on its training and test sets. On top of these, an operational error is referred to model's error rate on open-world deployment that could be higher than the test set error rate.

Run-time Monitoring Function: SOTIF and ISO 26262 standards suggest run-time monitoring functions as software error detection solutions. Designing monitoring functions to predict ML failure (e.g., false positive and false negative error) is difficult. ML models generate prediction probability for input instances but research shows prediction probability does not guarantee failure prediction [74]. In fact, DNN and many other ML models could generate incorrect outputs with high confidence in cases of distribution shift and adversarial attacks.

5.2 Safety Requirement Methods

To conduct a thorough analysis of the risks and hazards coupled to a development process, we discuss the requirements methods entailing the architectural framework introduced in Work Package 2.

5.2.1 Architectural Framework Safety Requirements

Deliverables 2.1 [139] and 2.5 [138] outline the concept of an architectural framework based on compositional thinking. The idea of a compositional architectural frame-

work is that the system of interest is divided into different clusters of concerns. Each cluster of concerns represents a specific aspect of the system at different levels of abstractions. A major group of concerns regard the different quality aspects of the system. Safety is such a quality aspect, and the architectural framework for VEDLIoT can encompass architectural views governing explicitly the safety aspects of the VEDLIoT system. Figure 5.2 illustrates that, besides safety, the architectural framework for VEDLIoT can also maintain clusters of concerns for security, ethics, and privacy. In this section, we will concentrate on safety as a cluster of concern that interacts with other aspects of the system, such as hardware configuration, communication and AI modelling.

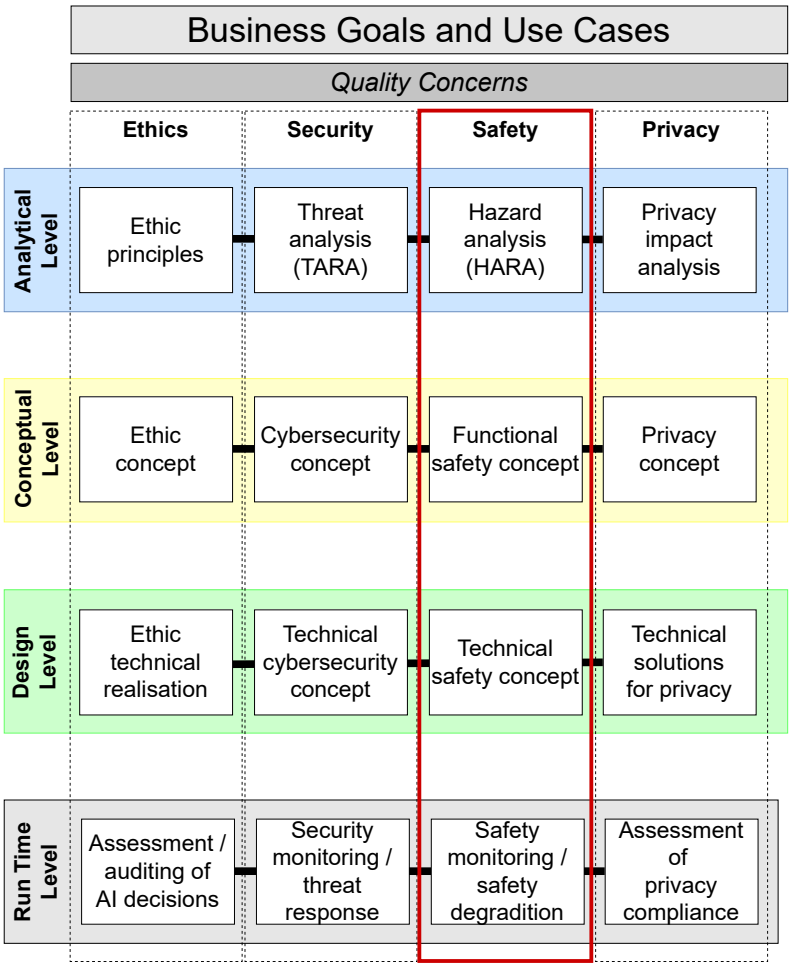


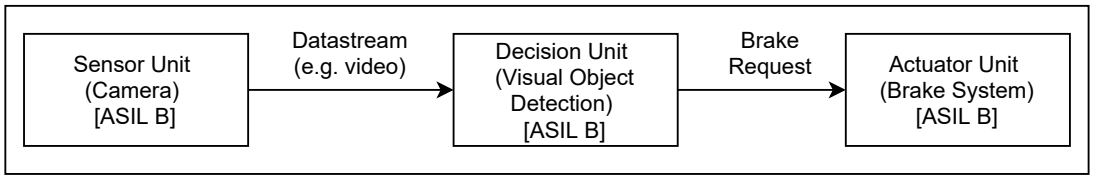
Figure 5.2. Architectural cluster of concerns for quality aspects of a VEDLIoT system

5.2.2 Interaction of Safety Aspects with Other Cluster of Concerns

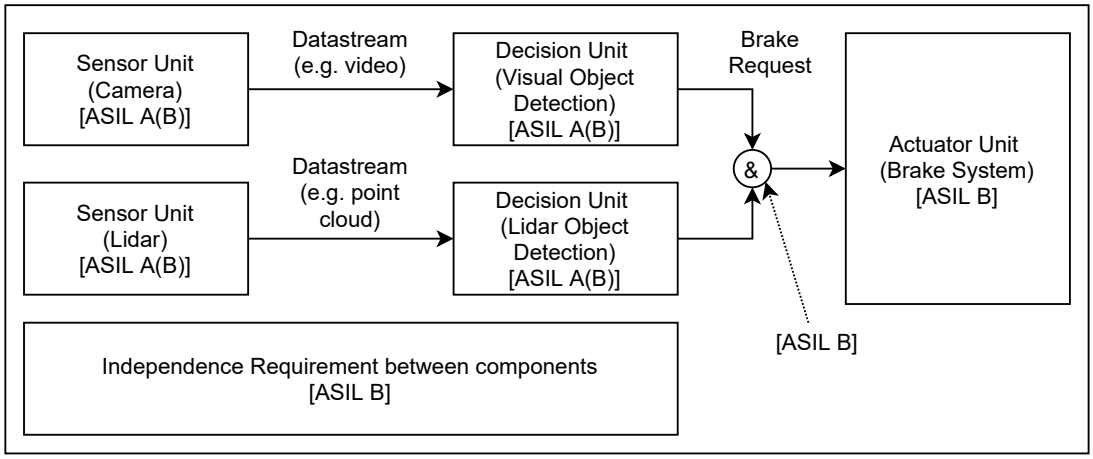
The full list of clusters represented in VEDLIoT's architectural framework can be found in Deliverable 2.5. Based on an example taken from the automotive use case, we will demonstrate how safety aspects influence other cluster of concerns of the VEDLIoT system.

Assume a system that shall trigger the brakes when a person is detected in the lane in front of a vehicle. Assume further, that, through the Hazard Analysis (i.e., HARA), the following safety goal has been identified: *"The system shall not trigger the emergency brake unintentionally (ASIL[1] B)"*. An extract of the high-level system architecture, as

[1] Automotive Safety Integrity Level



(a) System architecture before safety decomposition



(b) System architecture after safety decomposition

Figure 5.3. Safety decomposition in system architecture of automatic emergency brake system.

illustrated in Figure 5.3a consists of a sensing element (camera), a decision unit (object detection algorithm on an embedded platform) and an actuator (brakes), which constitutes *the item* in accordance to ISO 26262. While the brakes are designed to a high safety integrity level, the camera and object detection algorithm might not be able to achieve even ASIL B.

Therefore, a safety decomposition in the functional safety concept results in redundancy in the sensing system, and a lower ASIL on each component: An additional lidar sensor, together with a second object detection algorithm specifically designed for detecting objects in lidar point clouds, allows for the reduction of the required safety integrity level of all redundant components to ASIL A(B). The additional sensing system however must be independent from the first object detection system. The final high level system architecture after safety decomposition is illustrated in Figure 5.3b.

Influence of Safety on AI Model Design and Data Selection: The independence criteria between the two redundant systems creates requirements on the AI model design. Each sensing system must have an independent algorithm capable of detecting obstacles from the sensor data, without relying on the second sensing system. Because of the independence criteria that was established in the functional safety concept on the conceptual level of the system architecture, independence between sensing systems is inherited to the cluster of concern governing the AI model design. As a consequence, two independent AI models will be required, one for each of the sensing systems. Furthermore, the data used for training, testing and validation must be independent. An example on how the independence of the data could be broken is using joint data collection campaigns. If the joint data collection campaigns misses to collect relevant datapoints (e.g., driving at high speeds), the fault will propagate to both AI models.

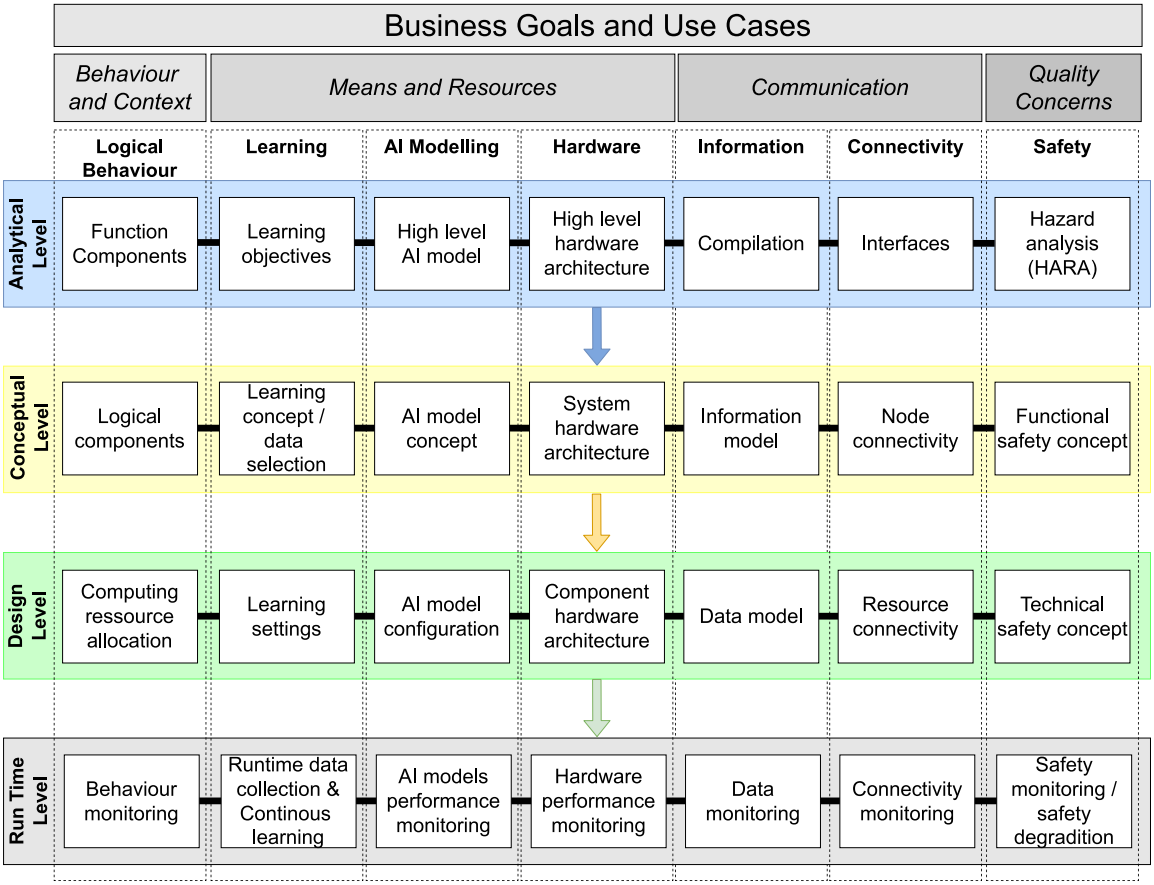


Figure 5.4. Interaction between different clusters of concern of the architecture, which represent different aspects of the VEDLIoT system

Influence of Safety on Hardware Architecture and Logical Components: By introducing safety decomposition in the functional safety concept, correspondences (morphisms) to the system hardware architecture view and the logical components view were established in order to fulfil the required safety concerns. On the next level of abstraction, the technical safety concept establishes a view on the overall system's architecture that allows the fulfilment of the functional safety concept. For example, the technical safety concept provides a view on the system architecture that requires the logical component "Visual object detection" to be deployed on safety certified hardware components. This creates a correspondence to the computing resource allocation view; indicating that the object detection algorithm only works at daylight, which creates a correspondence to the constraints/design domain view.

A major advantage of the compositional architectural framework is the ability to trace links or correspondences between the different architectural views, as illustrated in Figure 5.4. Let us assume that after years of service, the system's hardware shall be upgraded and one decides to replace the two independent processing units with a more powerful single unit. Then, the link between the system hardware architecture and the functional safety concept would remind the system designer of the safety concern that triggered the original design decisions years ago to have separated and independent processing units for the two independent object detection algorithms.

5.3 Safety Verification Methods

Simulated scenarios will enable testing difficult or even hazardous events. The variation of scenarios may also be large and is only limited by the simulator processing power. Simulations can be done at multiple levels involving more or less of the final product and also the user:

- Model In the Loop (MIL). This is where the ML model for the intended function or feature, at different levels of optimization, is tested.
- Software In the Loop (SIL). This includes testing basic software running on a processor including ML algorithms, interface blocks and possibly operating system components.
- Processor In the Loop (PIL). This is where the ML model is implemented on the target processor hardware.
- Hardware In the Loop (HIL). This is where the algorithms are running on the expected product hardware. This also includes sensors and communication interfaces at different levels of real or virtual hardware.
- Vehicle In the Loop (VIL). The dynamic behaviour of a vehicle is included and may have an impact on system data input and the required output data. The vehicle may be virtual or real.
- User In the Loop (UIL). Basically simulating the full system with algorithms, hardware, vehicle and the driver activities. The assumption indirectly affects the system input data.
- Human In the Loop (HITL). This in a way the same as UIL but in this case the we are focusing on the actual ML models and the interaction between a human and these models to solve the problem/function at hand.

We believe only a limited number of real-world field tests are necessary to verify simulated data which we will continue evaluating during the remainder of the project.

5.4 Safety Runtime Methods

As shown in both Figures 5.2 and 5.4, the VEDLIoT architectural framework explicitly supports a runtime level of abstraction at which runtime monitoring aspects can be defined. Dependencies, between the quality concerns and other aspects of the VEDLIoT system, concerning runtime, can also be created (similar to analytical, conceptual, and design levels of abstraction). We discuss runtime monitoring, deterministic and rule based run-time checks for safety monitoring below.

5.4.1 An Architectural View on Runtime Monitoring

Common safety standards, such as ISO 26262 [81] emphasise the system design phase and less the runtime aspects of the system. However, systems containing some form of machine learning are not comparable with rule-based systems in the sense that a predetermined (safe) behaviour cannot be guaranteed through enforcing a correct design of the system. The reason behind that is that machine learning bases on statistical inference, for which an absolute correct behaviour cannot be guaranteed at

design phase. Therefore, the architectural framework contains the explicit architectural view *Safety monitoring and safety degradation*. Safety monitoring can entail a number of measures to ensure the correct behaviour of the system. For example, the data coming from the sensor modules used for inference through the machine learning model can be checked for having the correct quality aspects required to achieve the desired behaviour and performance of the system.

A list of possible quality aspects on data can be found in Deliverable 2.5. Note that the cluster of concern *Information* contains an explicit architectural view called *data monitoring*. If the quality of data is deemed safety critical, a correspondence, or link, between the *safety monitoring* and the *data monitoring* can be created. Such a link between the safety aspect of the system and the information / data aspect of the system would emphasise the safety of the data at runtime. Besides runtime data monitoring, other links such as between safety monitoring and behaviour monitoring can exist. An example is the recording of false positive braking, i.e., false detection of objects, and disabling the system after a certain number of false positive detection events. As VEDLIoT emphasises the usage of distributed computing systems (i.e., IoT), some components of the safety critical system might be moved to a non-local processing unit (e.g., an edge device located at a mobile network base-station). In these cases, *connectivity monitoring* will be part of the safety case of the system.

5.4.2 Deterministic and Rule-based Runtime checks

Software: The operation of software modules may be checked by looking at the cycle time and amount of memory access. In some cases, software may end up in a loop due to unexpected input signals. This will be identified by the cycle time monitor. Slightly longer cycle times than normal may indicate processor management problems like increased number of system interrupts which could be due to a pending error.

Hardware: Most hardware modules, especially power supply units have the ability to check e.g. that voltage levels are within the limits for the processing and control hardware used in the different parts of the system. In some cases the actual current may be measured but most units have over-current protection and warnings. These supervision modules normally do not have any back-up or redundant hardware but since the outputs signals are active, during normal operation, it may be possible to identify an error in the monitoring function. Other hardware monitoring functions are time-out detectors, sometimes referred to as watch-dogs. By regularly requiring a trigger signal, it checks that other hardware units are running according to some time schedule. More accurate timing supervision may be necessary for system clocks. This can be achieved by comparing the clock cycles to a reference clock. The reference clock can be implemented as a dedicated hardware unit or as a combination of system clocks. In the latter case, error correlation between the system clocks and the reference, has to be recognized. If either a system clock or the reference gets off specification, a warning will be issued. Temperature is also an indicator of correct operation as hotspots in components could lead to failure. Even a slight increase in relative temperature of the system could be a warning of possible failure or just of higher environmental temperature.

Power consumption: As discussed above, voltage and current for individual hardware modules are monitored to ensure correct functionality. It may also be interesting to monitor the system power consumption either through software or by a

dedicated hardware function. Both peak and average power are useful indicators of a correctly functioning system.

5.5 Summarizing Safety

We have considered the existing Safety standards with respect to ML specification, verification, inference and learning. We have identified gaps and we will propose processing and standardization methods to enhance the safety level of ML based systems. We have identified the need of a monitoring solution capable of evaluating the performance of AI/ML models, in- and output data quality, communication robustness and capacity in real-time. We have also identified the need of a learning data selection tool that guarantees the correct number of datasets and the distribution of scenario variability over the datasets for each function/feature. We will continue this work in Task 5.5 during the remaining part of the VEDLIoT project.

6 Support for CFU Accelerators in Renode

This chapter presents our work in supporting CFU - Custom Function Units with Renode[24] - an open source simulation framework created by Antmicro. CFUs are AI accelerators tightly integrated with RISC-V soft CPUs, prepared to improve performance of operations frequent in a specific Machine Learning workflow.

The extendible nature of the RISC-V ISA[121] - its modularity, support for extensions and a future-proof design, encourages system designers to introduce custom improvements, tailored specifically to improve their planned workflows. This customizability, while unprecedented in the world of popular ISAs, is very well embedded in the RISC-V concept. The RISC-V ISA is, above all, modular. It's up to the designer to select which extensions, ratified as part of the standard or fully custom, will be supported by their CPU. Apart from the standard, "big" extensions, RISC-V ISA specification reserves a space for custom instructions. These can be used to provide micro-optimizations in the CPU architecture, expose specific features like accelerators or hardware loops, allow for deviations from the ISA specification, etc. While this modularity brings a lot of freedom to the design process, it is not trivial to handle from the software perspective. Software can either declare very specific requirements on the extensions available in supported CPUs, or take measures to address the variety of solutions in the RISC-V ecosystem. This latter approach is prevailing in generic, widespread software like Linux - whenever a CPU is unable to handle certain instructions and raises an exception, firmware addresses it with software emulation of offending instructions. While fairly robust, this flow requires a significant amount of consideration and implementation to be made.

Handling custom hardware can be even more challenging from the software perspective, if the hardware is being designed in parallel with the software - FPGA-friendly RISC-V ISA makes this a very common scenario, calling for proper verification of the software-hardware interoperability at all stages of development. As part of the Work Package 4, VEDLIoT is developing an FPGA based accelerator for Machine Learning payloads. Since this accelerator will be focused around the use of custom RISC-V instructions, proper testing will be set up with the Renode framework. Renode will be used to ensure the fitness and robustness of the SoC, allowing for testing of each iteration of software and FPGA payload without hardware in the loop.

6.1 Custom Function Units

A common approach to increase performance of Machine Learning processing is to use an external ML accelerator - either in the form of a separate generic processing unit that offloads parts of the computation from the main processor, or via dedicated expansion modules, dedicated to perform well in specific contexts. To benefit from the inherent configurability of the RISC-V ISA, and taking into account that Machine Learning workflows rely on highly specific patterns of computation, the RISC-V community formed the RISC-V FPGA Soft Processor Working Group led by Google and Antmicro and proposed a concept of Custom Function Units, or CFUs[23]. A CFU is a highly specialized extension to the CPU, tailored to speed up very specific fragments of processing, that would be otherwise expensive when executed on a generic processing unit. CFUs are mainly discussed in the context of FPGA processing, where a

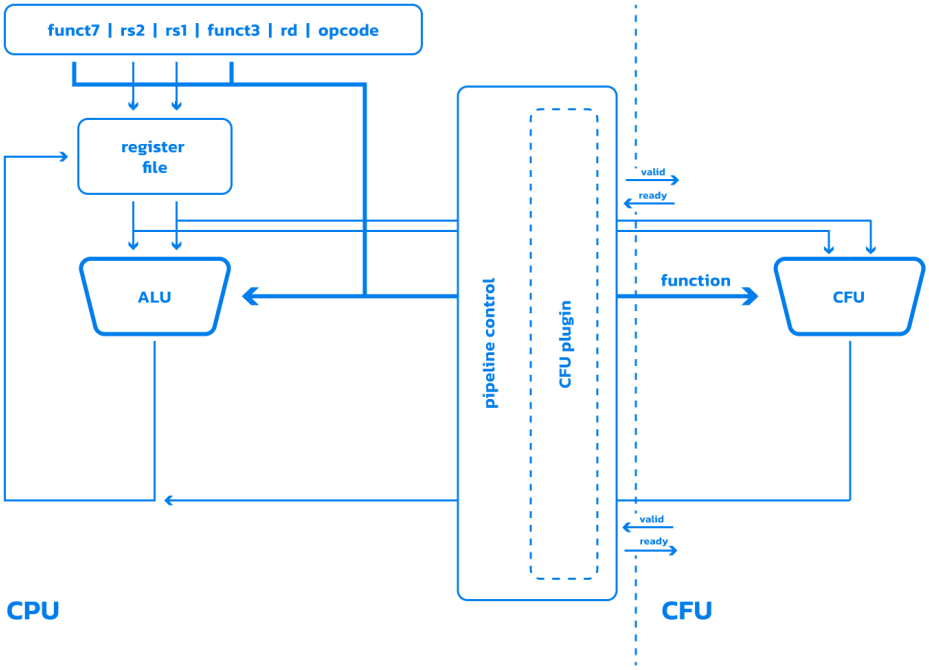


Figure 6.1. Connection of a RISC-V core with a CFU accelerator

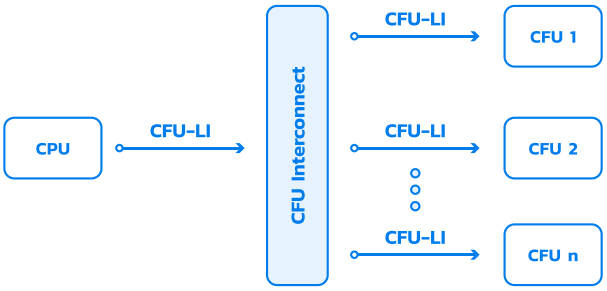


Figure 6.2. Composition of multiple CFUs

custom piece of one-off hardware can be defined easily, tested rapidly and, if needed, disposed of in favor of a better implementation.

CFUs are accessed via a set of standardized custom instructions, and are connected with the CPU via a well-defined interface (see 6.1). This interface is, typically, the only connection between the core and the CFU - in most cases there is no direct link with the memory, and even the internal state of the CPU (e.g. state of registers) is not visible for the Custom Function Unit. The instruction format allows for multiple CFUs to be used simultaneously, enabling composition of operations (see 6.2) for an even greater performance gain. For a more detailed description of the CFU interface, please see Deliverable 5.1[115].

6.2 Renode Simulation Framework and CFU Support

As stated before, Renode[24], an open source simulation framework, is the key component to ensure the quality and robustness of the implementation of the accelerator developed as part of the WP4.

6.2.1 Introduction to Renode

Renode is a functional simulator - it allows users to execute code targeting one of the supported ISAs (with RISC-V being one of the prominent examples) and accessing various peripherals. The simulation offers a high level of fidelity in resembling the real life hardware - this enables running exactly the same software that one would run on hardware. Peripherals are modeled on the register level, which means that their observable behavior is similar to the actual hardware. Both the core and peripherals, however, may differ from their real life counterparts with regard to timing, internal state etc. This approach allows for a reasonable balance between fidelity and performance of the simulation.

Renode is built with the concept of composable building blocks that constitute a simulation. A CPU can be accompanied by peripherals to compose an SoC (System-on-Chip). An SoC can be connected to various sensors and ports, creating a board, or, in Renode terminology, a machine. These machines can be connected with various mechanisms, ranging from a simple serial port, through a CAN (Controller Area Network) bus, to wireless BLE (Bluetooth Low Energy) connections. CPUs themselves can also be considered as created from various blocks, especially when discussing the RISC-V ISA [121]. A user would select appropriate instruction sets to be supported by the CPU, size of a word and additional features, to best reflect the emulated core.

6.2.2 Custom Instructions Support

One of the outstanding features of Renode is its support for custom instructions, which emerges from the need of the properties of the RISC-V ISA itself and which is very well suited for supporting CFUs. Traditionally, Renode users could define custom instructions in two ways: either by describing them in C# or in Python. To add a custom instruction in C#, the CPU would have to declare it in the following manner:

Listing 6.1. Custom instruction installation in C# code

```
InstallCustomInstruction(
    pattern: "0100001RRRRRSSSSS000DDDDD0110011",
    handler: opcode => MultiplyAccumulate(opcode),
    name: "p.mac rD, rs1, rs2"));
```

In the above example, the pattern is a string describing the format of the opcode (machine-code representation of the registered instruction): 0s and 1s define its static part, while other characters are traditionally used to denote opcode parameters (in the example: R, S and D). “Handler” is the actual implementation of the opcode. “Name” is the user-facing description, used for logging purposes.

Python implementations are similar, but more often used for debugging or mocking purposes. A sample instruction that only prints out a message when it’s called could be defined in the example below. The first string in the example is the opcode and the second one can be any Python code.

Listing 6.2. Custom instruction installation in Renode’s Monitor

```
sysbus.cpu InstallCustomInstructionHandlerFromString
    "10110011100011110000111111011AAAA"
    "cpu.DebugLog('custom instruction executed!')"
```

6.2.3 Verilator Co-Simulation

Since the concept of CFUs is usually discussed in the context of their FPGA implementations, another Renode feature should be mentioned: the ability to co-simulate with "verilated" HDL code. "Verilation" is a process of recompilation of Verilog code to C++, using the tool Verilator [142], the result being a cycle-accurate model of the original IP. Renode uses the Verilator co-simulation primarily in the context of peripherals. It supports various bus types (AXI4, AXI4-Lite, Wishbone) and interrupt connections. This gives the system designers a tremendous benefit of not having to reimplement their accelerators as Renode models. This leads to fewer bugs and faster turnarounds. The process of verilation leaves the user with C++ code. To connect it to Renode, the user has to implement a small shim layer that handles connecting signals from the bus to the peripheral. A library of examples is available on GitHub [25]. Originally, to guarantee portability over different host operating systems, verilated peripherals were connected to Renode via sockets. This solution comes with a performance cost due to the protocol handling layer and the network stack involvement, but is generic and acceptable for peripherals which are not accessed very often.

6.2.4 CFU Simulation

The performance cost of socket based connections is becoming more apparent when the communication with the verilated block is frequent. If this block represents a hot-path instruction, the slow-down may be prohibitively large. One of the tasks undertaken within the project was to implement an API-based communication mechanism between Renode and verilated blocks. This effort consisted of the following steps:

- Preparation of a build flow to build verilated peripherals as shared libraries. One of the goals was to retain the possibility of using the socket-based connections without the need to duplicate logic in the connection layer.
- Implementation of the library-based interface on the Renode side.
- Preparation of the user-facing API to allow easy use of both connection options.

The resulting speed-up of the processing was significant and the API-based connection became the default in verilated peripherals. The next step was focused on implementation of the CFU interface itself. The CFU specification is still under development, so the final shape of the interface is not yet known. However, current documentation and, what's even more important, current users of CFU accelerators, provide enough information to create a functional and useful implementation.

During the course of the project, an interface layer was implemented allowing users to use CFUs conforming to feature levels 0 and 1, described by the current standard as, respectively, "combinational" and "fixed latency, pipelined, no flow control". To create and use CFU extensions in Renode, the user has to follow the following steps:

- Implement the interface layer. To achieve that, the user has to use a library provided by Renode and connect the CFU signals.
- Compile the interface and the Verilog CFU implementation with Verilator.
- Prepare a simulation that uses the created verilated CFU.

While this flow has already been used in Renode to connect to verilated peripherals, there are significant differences between Memory-Mapped I/O (MMIO) blocks and custom verilated instructions, stemming mainly from the performance requirements.

Clocking of CFU blocks: Verilated peripherals connected to Renode periodically receive information about time progression, in the form of the number of "ticks", which translate directly to changes of the clock signal of the peripheral. These inputs allow the logic of the simulated block to move forward along with the simulation. Regardless of the periodic time progression, verilated peripherals are clocked whenever they are accessed by the CPU. These additional signals help us bridge the gap between the worlds of the cycle-accurate Verilator simulation and the behavioral Renode simulation. CFU blocks, as defined in the specification and used in practice, do not require periodic clocking - they are not expected to perform any processing when not prompted by CFU instructions. For this reason, the flow of interaction was changed. When the CPU issues a CFU-triggering instruction, Renode starts the clock signal and observes two CFU signals: `req_ready`, indicating if the CFU block is ready to accept the request, and `resp_valid`, indicating that the response is ready and can be retrieved. As soon as the `resp_valid` signal is asserted, the clock is stopped. This flow has two important impacts: it reduces the load on the host (as the CFU model does not have to be evaluated when not needed) but also decouples timing of the CFU processing from the rest of the system. The latter is not considered to be a major issue, as Renode does not differentiate between different instructions in terms of time of execution.

Interface: CFUs are designed to shift the computational complexity from software to FPGA. They are usually used in the hot paths of the processing, maximizing their impact on the application's performance. Frequent communication between Renode and the CFU block using sockets, the original method of integration with verilated blocks, would be prohibitively inefficient. Not only would it require the network stack to be involved, it also would need to serialize and deserialize requests and responses. For these reasons we decided to support only the native, library-based interface. This mechanism is already used by Renode to execute regular CPU instructions, which naturally made it a good fit for CFU integration.

6.3 Renode CFU Support in Practice

As mentioned above, Renode CFU support is based on both, a draft specification and a practical implementation. This note-worthy use case of CFU is Google's CFU Playground [67], developed by Google in cooperation with Antmicro. The CFU Playground provides an open source framework which offers a methodology for reasoning about ML acceleration and developing Custom Function Units using FPGAs and simulation. It offers a range of examples and plenty of supported soft SoCs and CPUs. Although to use verilator for co-simulation purposes the code has to be written in Verilog, CFU Playground uses Amaranth (formerly known as nMigen) [117], allowing developers to create their accelerators in Python, subsequently translated to Verilog. Thanks to the modularity of Renode, we were able to autogenerate platform descriptions from CFU Playground samples. With this capability we set up a Continuous Integration environment, testing every change to CFUs, underlying SoCs or the ML software that runs on them. At the moment the CI setup in CFU Playground runs more than 220 different compilation targets in Renode, ensuring the robustness of implementation.

7 SIRE - Trusted Verifier Service

In this chapter we will explain further SIRE (acronym for truSted verIfieR sErvice, previously called TMS - Trusted Membership Service), a replicated infrastructure that supports multiple functionalities necessary to an IoT System. In Deliverable 5.1 [115], we outlined the main features of SIRE on a conceptual level as well as some of the challenges associated to its design (revisited in Section 7.1). In this deliverable, we will give further insight on the system's architecture and implementation details of its features (coordination and attestation) as well as other challenges that came to light as we developed it further.

As the interest in IoT grows, so does the need for well-rounded systems that can deal with various issues inherent to the area, such as the dynamism of the system composition, ensuring the security requirements are met etc. With the intent of solving this type of problems, we came up with SIRE, a replicated system that will support remote attestation, membership management, auditable integrity-protected storage and coordination primitives.

7.1 Concepts

In Deliverable 5.1, we explained a few concepts that SIRE is built upon. In order to recall them, we wrote a short summary of each:

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 [44]. To achieve this, the attester generates an evidence of its properties which the verifier will evaluate using a *policy* - set of rules that specify which devices can join the system. Typically, this is done by using some secure hardware components such as a Trusted Platform Module [136] or a processor-native trusted computing technology such as Intel SGX [54] and Arm TrustZone [116], deployed on the attester.

Coordination Services: Coordination Services provide a consistent and highly available data store with enough synchronization power for client processes to execute tasks such as mutual exclusion and leader election. Two examples of this kind of services are ZooKeeper [76], a crash-tolerant coordination service with a node structure, and DepSpace [41], a Byzantine fault-tolerant coordination service with a tuple space structure. Furthermore, these services can be extensible, i.e., they can allow the introduction of small pieces of code called extensions to be executed when certain operations are called [59].

State Machine Replication: In the *State Machine Replication* approach [127], i.e., SMR, an arbitrary number of client processes send requests to a set of replicas. These replicas implement a stateful service that receives these requests and updates its state depending on the operation contained in the clients' requests. Once enough replicas send matching replies to the client, its invocation returns the result computed by the service. The goal of this approach is to keep the service consistent, by making the service state maintained by each replica evolve in the same way.

Blockchains: A blockchain [108] 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 [135] consists of a sequence of blocks where each one contains the cryptographic hash of the previous block in the chain.

BFT-SMaRt: BFT-SMaRt [43] is an open-source library that implements a modular State Machine Replication (SMR) protocol [134], as well as features such as state transfer and group reconfiguration. This SMR protocol grants BFT-SMaRt the characteristics needed to achieve Byzantine-fault tolerance. We will use this library to provide fault and intrusion tolerance for SIRE.

7.2 SIRE - System Architecture

SIRE is a 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 7.1.

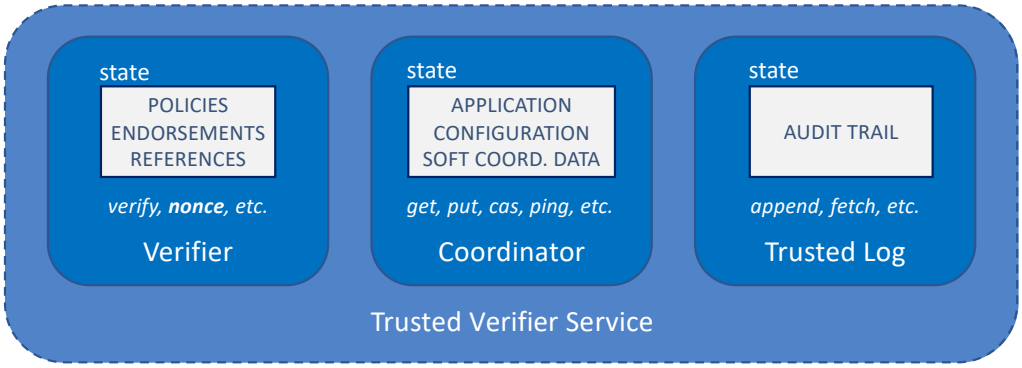


Figure 7.1. Trusted Verifier Service (SIRE) main modules

The first module, *Verifier*, is used to support remote attestation, following the ideas described in Section 7.1. This provides operations and functionalities for verifying the evidence supplied by a device for attestation. To do this, the module must be 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 an extensible 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. It will also be responsible for managing and executing extensions introduced by the application administrator.

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) [42]. This module will mostly be composed by the transaction ledger from BFT-SMaRt [42], adapted to SIRE's needs.

7.2.1 SIRE's Architecture

The SIRE system will be composed of two main entities: the SIRE Server and the SIRE Proxy; and two main types of clients: Devices and App Administrators as shown in Figure 7.2.

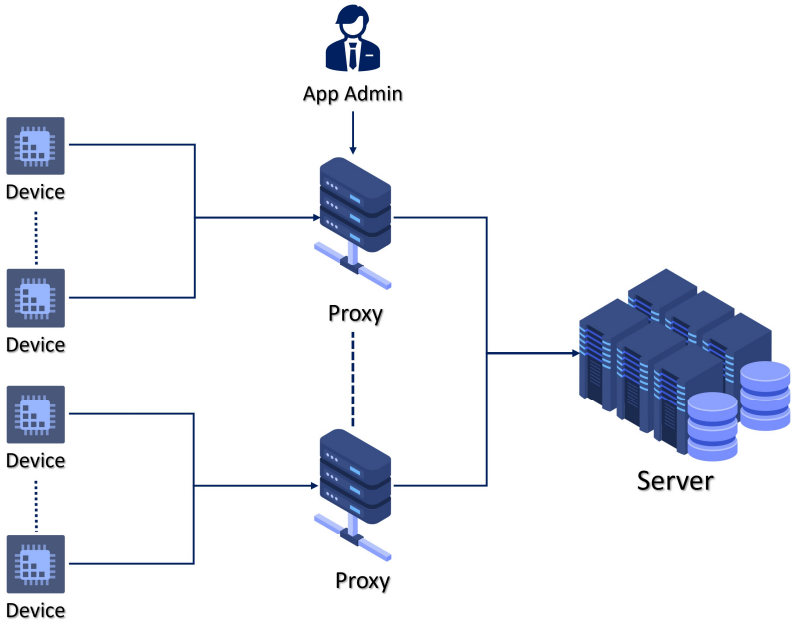


Figure 7.2. SIRE's Architecture

SIRE Proxy: The SIRE Proxy will mainly be responsible for translating both types of clients' requests to the SIRE Server, providing a simple and standard API and sparing the clients of interacting with BFT-SMaRT directly, which is fundamental to the rudimentary nature of some IoT Devices' hardware. The API will be composed of three interfaces:

1. *Management Interface* (Table 7.1): Used by Application Administrators to manage and configure extensions and policies. The user can access it through a Web Client or REST calls. All the function calls made through this interface will require prior authentication of the Application Administrator.

2. *Map Interface* (Table 7.2): Zookeeper-like [76] interface used to access the generic key-value store contained in the SIRE Server. This store will be used to maintain application configurations, membership state and any other data the user might need, hence the use of generic values. Any function in this interface might trigger the execution of an extension, if defined by the Application Administrator. Used by both the Application Administrators, through REST calls or the Web Interface, and the Devices through RPC or REST calls. Application Administrators will need to authenticate before calling any of these interface's functions.

3. *Membership Interface* (Table 7.3): Interface to be used by the devices to perform the attestation and coordination protocols. Any function in this interface might trigger the execution of an extension, if defined by the Application Administrator. Used only by the Devices through RPC or REST calls.

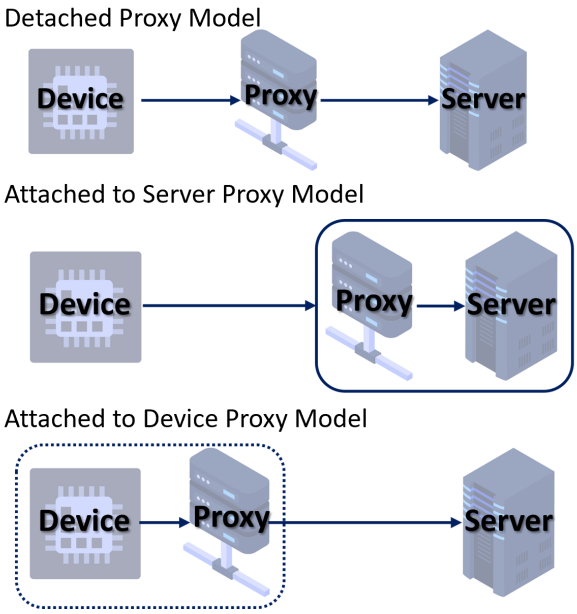


Figure 7.3. SIRE Proxy Configuration Models

With this approach, a wider compatibility of IoT devices with SIRE can be much easily achieved. Each server replica will have a proxy attributed to it, to a certain extent and can be configured in the following ways:

- *Detached Proxy Model:* The Proxy is in a different machine and can be at the edge or at the same location as the server. In this case, the proxy is not dedicated to any device or server and should be the most common model.
- *Attached to Server Proxy Model:* The Proxy is in the same machine as the server and only acts as a library for it. Can be useful when the server itself is close to the devices.
- *Attached to Device Proxy Model:* The Proxy is at the same place as the device and is dedicated to it. This model is particularly useful when we have a critical device.

The different proxy configuration models are also illustrated in Figure 7.3. The Proxy will also be responsible to send each message it receives to all the server replicas, not only to the one attributed to it. The Proxy is assumed to be untrusted, containing only soft state and will not have access to any sensitive data.

SIRE Server: The SIRE server will be implemented on top of BFT-SMaRT [43] and will be composed by the three modules described before. This will grant SIRE fault tolerance which is an essential property to coordination systems [59]. The leader-generated timestamp and the total-order multicast will also be useful features for the coordination module. The server will also be the one storing application states and configurations (including extensions and policies).

Devices: SIRE will be able to support a multitude of devices, but we are more specifically aiming for those with an Arm processor supporting the TrustZone TEE, since that's what will be used in the VEDLIoT project. Due to the rudimentary nature of the TrustZone TEE, the devices will only be able to communicate with SIRE through

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

Table 7.1. Management Interface

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

Table 7.2. Map Interface

sockets in raw bytes, but nonetheless, we intend to also support a REST interface with Protocol Buffers [68] serialization for a wider compatibility. Protobuf is a data format used to serialize structured data. It uses a very simple descriptive language to define the structure of the data, which is then used to automatically generate code for a variety of programming languages.

Application Administrators: The App Administrators will be able to manage and configure policies and extensions, as well as view the state of their application's devices. They will do this by communicating with the proxy, which will then forward the requests to the server replicas. They will be able to do this either by a REST interface or a Web Interface that we will develop using Swagger [133].

7.2.2 Implementation Details

While the SIRE system isn't fully developed yet, it has been defined and planned, as well as having a few parts of it implemented, which will be further detailed in this section. SIRE will be developed in Java along with Groovy for extensions.

Attestation Protocol

The attestation protocol we are using is based on the WaTZ framework [107] but adapted to be used with a distributed verifier. WaTZ is an efficient and secure runtime environment for trusted execution of WebAssembly code in Arm's TrustZone TEE [116] with the purpose of performing remote attestation [107]. The WaTZ's attestation protocol (Figure 7.4) can be reduced to basically four messages:

(a) Message 0 (attester→verifier): The attester generates a session key pair $\langle a, G_a \rangle$ and sends the public part G_a to the verifier.

(b) Message 1 (attester←verifier): When the verifier receives msg_0, it generates a

Function	Description
join(deviceid, msg0)	Join the system and start the attestation protocol
leave(poif)	Leave the system
ping(poif)	Assures the system that the device is still running
getView(poif)	Get current membership of system from the app associated to *poif*

Table 7.3. Membership Interface

session key pair $\langle v, G_v \rangle$ and computes a shared secret from the public session key of the attester G_a and its private session key v, which results in G_{av}. This shared secret key is then derived into a *Key Derivation Key* (KDK), which in turn is further derived into another two shared secrets: K_m to calculate MACs and K_e to encrypt subsequent messages in this session. These derivations are the same as in the Intel SGX attestation protocol [54]. Message 1 is then composed by G_v, the verifier's ECDSA public key V and a signature of both public session keys G_v and G_a. This is then signed with K_m, creating a MAC which is appended to the message.

(c) Message 2 (attester→verifier): The attester verifies the signature of the public session keys to prevent the possibility of a masquerading or replay attack, and verifies the MAC of msg_1. It also checks whether the public key V matches the hardcoded key in the Wasm application, to ensure that it's communicating with the intended service, preventing possible malicious attackers from altering that key, since it is part of the code measurement. The attester then makes the necessary computations to generate G_{av} in the same way the verifier did when processing message 0. The attester then creates msg_2 by concatenating its public session key G_a with a newly-generated evidence signed by itself, where the anchor of the transport layer is the hashed concatenation of the public session keys. Finally, it appends a MAC.

(d) Message 3 (attester←verifier): The verifier checks the MAC of message 2 and the verifies that G_a matches the one it has previously received. It also verifies if the anchor matches the public session keys it has, to possibly reveal a masquerading or replay attack. It extracts the evidence's public attestation key and checks if it is contained in its list of endorsed public keys to determine whether this is a known device. If it is present in the list, the digital signature of the evidence will be checked, which reveals if the hardware whether the hardware is genuine. Finally, the code measurement claim is compared with a list of reference values to verify that the Wasm application is trustworthy. If all verifications pass, the verifier sends msg_3 with the confidential data encrypted with AES-GCM, which requires iv (an initialisation vector).

Since WaTZ's attestation protocol was designed for a centralized server, we had to adapt it to the distributed nature of SIRE. For the device, it will be basically the same, it will send two messages and receive another two, with only the signing method being changed. The biggest difference will be the added steps between the proxy and the servers:

- When processing message 0, the key derivations are done in the proxy which then forwards the G_a and G_v to the server for it to sign using a distributed signing method. In this case, we opted for Schnorr signatures [128, 129].
- When processing message 3, the proxy will receive the evidence and its signature and forward it to the verifier to validate the evidence. Since the proxy is

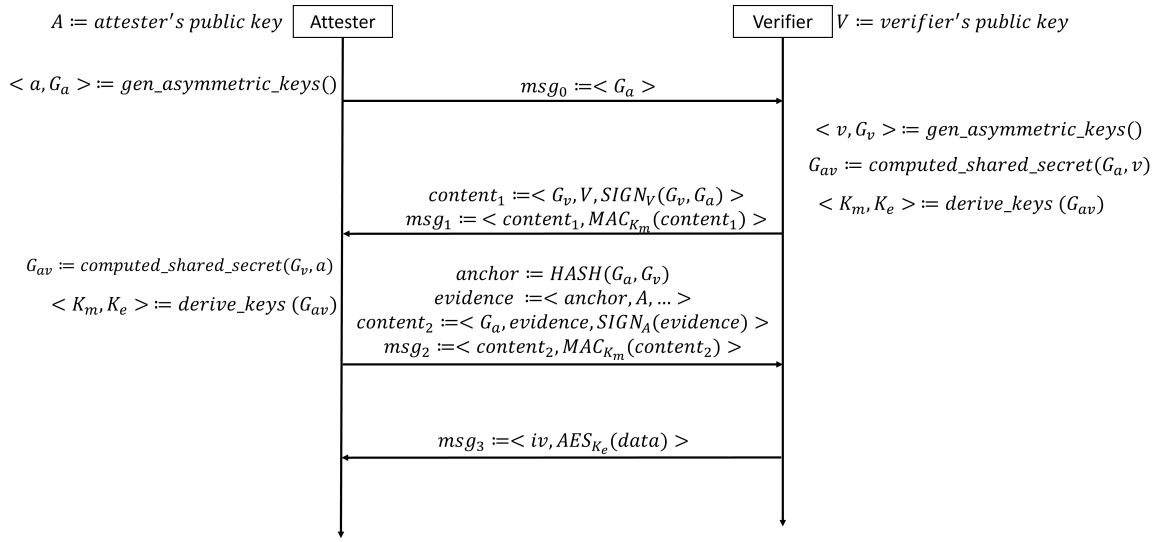


Figure 7.4. WaTZ Protocol [107]

Incoming claim set

```
version= 1.0;
authorizationrules {
      [ type=="x-ms-sgx-is-debuggable", value==false ]
       && [ type=="x-ms-sgx-product-id", value==<product-id> ]
       && [ type=="x-ms-sgx-svn", value>= 0 ]
       && [ type=="x-ms-sgx-mrsigner", value=="<mrsigner>"]
    => permit();
};
issuancerules {
      c:[type=="x-ms-sgx-mrsigner"] => issue(type="<custom-name>", value=c.value);
};
```

Outgoing claim set

```
[type="<custom-name>", value=c.value, valueType="*", issuer="AttestationPolicy"]
```

Figure 7.5. Example Policy from Microsoft Azure Attestation [40]

untrusted, we need to make sure it can't read the attestation result or tamper with it. To achieve this, the verifier will encrypt the attestation results with the attester's public key, sign it with its public key and forward it to the proxy. The proxy will in turn encrypt what it received from the verifier and send it to the device along with the initialization vector iv.

Policy Definition

A policy is a rule that is applied over the evidence to produce the attestation result [44]. There are two ways that policies can be implemented: (1) using scripts that are populated with the values from the evidence (e.g. Intel SGX Remote Attestation [79]) or (2) using logical expressions evaluated with the values from the evidence (e.g. Microsoft Azure Attestation [40]). We intend to implement it using the logical expression approach. An example policy can be seen in Figure 7.5.

Extensions

Extensions are small pieces of custom code that will be executed, when certain criteria are met, at the server side, making the coordination kernel more versatile and efficient without sacrificing its simplicity [59]. The extensions will be implemented in the form of Groovy classes [26]. We chose Groovy because it is already included in JDK (Java Development Kit) and can be directly ran on the JVM (Java Virtual Machine). The extension is sent to SIRE by the app administrator in the form of a string of text, which in turn is compiled by the extension manager present in the SIRE server and instantiated.

The extensions will be run whenever an operation from the Membership (*join, leave, ping, getView*) or Map (*put, get, getList, delete, cas*) Interface is called. When the extension is first introduced in the system it should be associated with a key, composed by $appId$ + $extType$ + key, with the possibility of $extType$ and key being null. The $extType$ parameter corresponds to the type of operation that will trigger the execution of the extension and the key parameter the key associated with the operation that will trigger the extension (e.g, an extension associated with application $app1$, operation type $extPut$ and key $exampleKey$ will be executed whenever $put(app1, exampleKey, exampleValue)$ is called). Since some parameters can be null, the extension manager first checks for extensions matching $appId$ + $extType$ + key, then for any matching $appId$ + $extType$ and lastly for $appId$.

Key-Value Store and Proof of Identification

The key-value store will have a similar interface to ZooKeeper [76], as explained in Section 7.2.1, but it has a few differences that distinguish SIRE's key-value store from it. It is generic, allowing a client to store any kind of objects they might need. Due to the serialization techniques used, it was necessary to store it in the format of byte arrays since Google's Protocol Buffers doesn't support generic types and, in the case of communication through sockets with raw bytes, no changes are needed. Since SIRE runs on top of BFT-SMaRt, so does the key-value store which makes it tolerant to Byzantine faults as opposed to crash faults in the case of ZooKeeper.

Besides this, SIRE will have multiple applications running alongside each other, while ZooKeeper only assumes one, and so, there is a need to distinguish which devices belong to which applications, since they should only be able to access and change the data of the one they belong to. This correspondence of device to application needs to be authentic, otherwise a compromised device can pretend to belong to another application and get unauthorized access to data from its victim. To prevent this, we added another parameter, Proof of Identification ($poif$), which consists of the device ID, app ID and a nonce encrypted using the derived key K_e from the shared secret G_{av}. If the device belongs to the application it claims to, the server will be able to decrypt the nonce, check if it matches and verify the device's identity. This nonce will be incremented after every interaction between the device and the server to prevent replay attacks. If a device belongs to more than one application in SIRE, it will need to have a different ID for each application.

7.3 Wrapping-up SIRE

In this chapter, we presented SIRE and a few of its developed features. However, it isn't fully developed and we are proceeding as planned, having only a few deviations

from the original idea, which were needed to deal with new-found challenges and problems. In the future, we intend to develop SIRE further by improving the attestation protocol, since it is adapted from other VEDLIoT's partners work. Since this was one of the last features to be implemented there are still things to improve.

Furthermore, we will research on policy standards, depending on the needs of SIRE's users, e.g., the kind of properties they want to attest, the format of the properties, etc. Another fundamental direction of this work is the testing of the system for its security and scalability and make adequate changes if required. We also plan on shifting from mock devices to a functional device client on a real IoT devices, to test it as close to its core use scenario as possible. Along with major additions like the Trusted Log, we will also continue to work on minor improvements like time-to-live for devices, encryption, and serialization methods, in the remainder of the project.

8 Robustness

Currently, autonomous systems employ advanced machine learning (ML) models for data analytic and decision making based on planning, control and other actions. Many of these systems, such as self-driving cars and remote surgery are critical, operate in a dynamic, uncertain and challenging environment and have strict requirements (e.g., timings, resource constraints and mission context). Furthermore, the training and testing data that are collected may contain noise (e.g. abnormal data, incorrect labels and incomplete information) and adversarial examples [147]. This requires high robustness of ML models to make reliable decisions for applications.

The VEDLIoT project aims to guarantee the security of a safety-critical distributed system through an end-to-end trust mechanism, a communication based on mutual attestation and secure execution of critical code. In addition, it aims to develop monitoring and adaptation mechanisms to increase the ability to control safety and robustness in distributed AI systems. This work contributes to the realization of a solution for the latter goal.

The following chapter is structured as follows: first, we illustrate the helpful literature for our work, then we explain the initial tests to improve the performance of a model and finally, we expose our proposal to solve the problem.

8.1 Related Work and Relevant Concepts

In ML, robustness is the model's capacity to generate correct output even in presence of input corruption and/or faults. To improve the robustness of a ML model, it is possible to act in different ways. For example, hardware problems can lead to transient faults, typically manifest as single bit-flips. This faults propagate along a Deep Neural Networks (DNNs) and can result in failures with related safety violations. As show in Figure 8.1 some techniques aim to prevent these phenomena [50].
Moreover, in recent years it has been realized that applied ML requires managing *uncertainty*. There are many sources of uncertainty in a ML project, including variance in the specific data values, the sample of data collected from the domain and in the imperfect nature of any models developed from such data. It is clear that with incomplete information reliable results cannot be given. To deal with that, tools and techniques from probability were adopted. Model's parameters are considered as distributions instead of fixed weights and we learn the weights' distribution [63]. This is useful to face the *robustness to distributional shift*, one of the main safety issues in ML, which discusses how to avoid having ML systems make bad decisions when given inputs that are potentially very different than what was seen during training [20]. Statistical distance measures can be considered as a method to measure distributional shift [39]. Conventional ML algorithms often adapt poorly to domain shifts. The modern ML community has many different strategies to attempt to gain better *domain adaptation* [97].
In addition to taking robustness into account, it is important to measure the performance of our model. If any technique to improve robustness decreases the accuracy of a model, then the technique should be reevaluated. An object detector was used for our work and we measured its accuracy with the *mean average precision* metric.

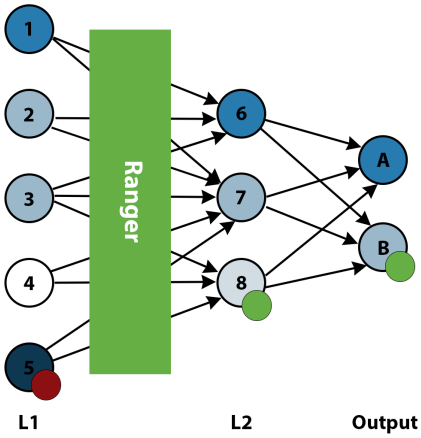


Figure 8.1. Fault prevention from propagating and generating incorrect output

Object detectors predict the location of objects of a given class in an image with a certain confidence score. Every detection is a set of three attributes: the object class (i.e., person), the corresponding bounding box (i.e., [63, 52, 150, 50]) and the confidence score (i.e., 0.583 or 58.3%). This score for each prediction box is computed as:

$$confidence\ score\ =\ conditional\ class\ probability\ \times\ box\ confidence\ score \tag{8.1}$$

So, it measures the confidence on both the classification and the localization. Box confidence score reflects how likely the box contains an object (objectness) and how accurate the boundary box is. To know this, we introduce the concept of Intersection Over Union (IoU), which is equal to the area of the overlap (intersection) between the predicted bounding box (red in Figure 8.2) and the ground-truth bounding box (green) divided by the area of their union. The greater the IoU value, the better the accuracy of the boundary box.

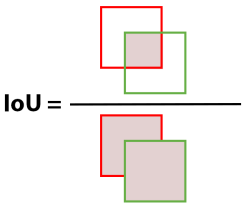


Figure 8.2. The intersection over union

Finally, the conditional class probability is the probability that the detected object belongs to a particular class. Now we can complete the formula in this way:

$$box\ confidence\ score\ \equiv\ P(object)\cdot IoU \tag{8.2}$$

$$conditional\ class\ probability\ \equiv\ P(class_i|object) \tag{8.3}$$

$$confidence\ score\ \equiv\ P(class_i)\cdot IoU \tag{8.4}$$

$$confidence\ score\ =\ conditional\ class\ probability\ \times\ box\ confidence\ score \tag{8.5}$$

The mean average precision metric is based on the concepts of *precision* and *recall*. The first tells us how many times it has made a correct prediction when it makes a prediction (8.6), the second tells us whether it has made the correct prediction as many times as it should have, or has lost some object (8.7).

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} \tag{8.6}$$

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} \tag{8.7}$$

The confidence score can be considered as precision and recall calculations, by considering only those whose confidence is larger than a confidence threshold to be positive detections.

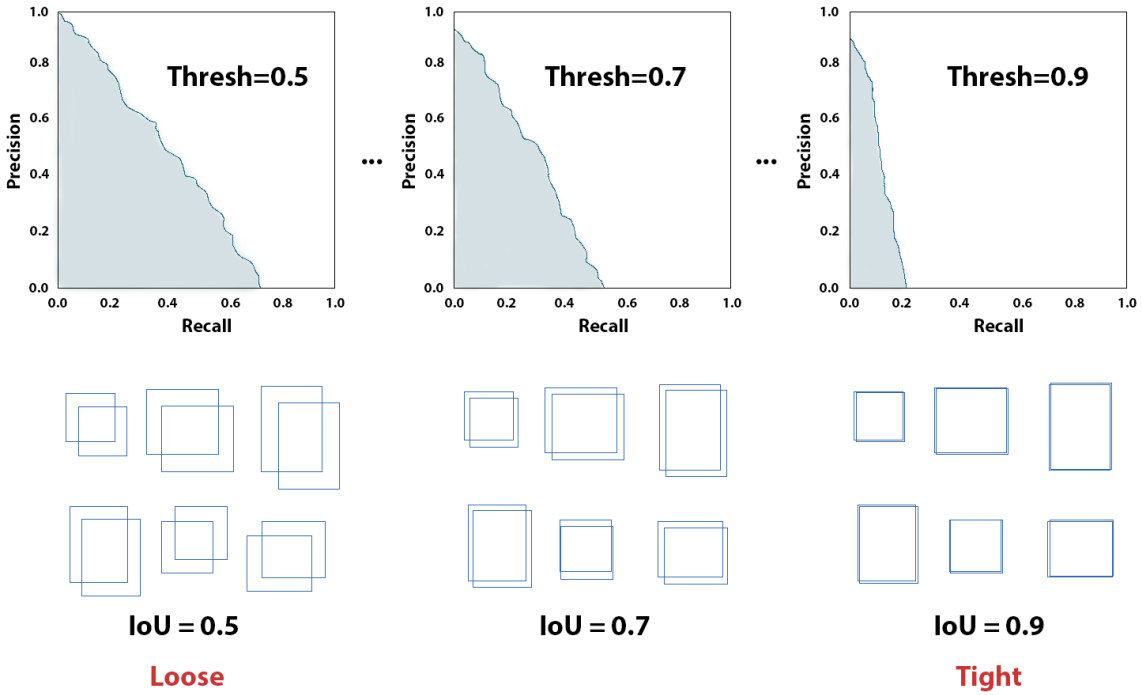


Figure 8.3. We can see three different PR curves and they differ by the IoU threshold used in the precision-recall calculation

As we can see in the Figure 8.3, the Precision-Recall (PR) curve is a plot of precision as a function of recall, which shows the trade-off between the two metrics for varying confidence values for the model detection. With AP@α we indicate the Area Under of Precision-Recall Curve (AUC-PR), which means Average Precision (AP) at the IoU threshold of α. Therefore AP@0.50 and AP@0.75 are the AP at IoU threshold of 50% and 75% respectively. In the end, we calculate the meanAP as an average of all APs for all classes. Since the AP corresponds to the area under the curve, it is obvious that a loose IoU threshold results in a higher AP score than a strict IoU threshold.

8.2 Initial Experiments

In this section, we present the experimental setup and initial results to answer the questions whether a model trained on a specific domain's sub dataset performs better than a model trained on the whole dataset.

A. Experimental setup

Research Question. We evaluated the hypothesis of using a multi-layer machine learning strategy to improve the performance of a model. In particular, evaluate the idea of split a dataset into specific domains to train specific models for better performance. We evaluate this idea by asking:
Does a model trained on a specific domain's sub dataset perform better than a model trained on the whole dataset?

ML Model. To carry out the experiments, *YOLO version 4* was used, a real-time object detector [47]. It resizes the input frames and passes them through a convolutional neural network to predict class and position of objects as showed in Figure 8.4.

Hardware Setup. Our experiments were conducted on an Ubuntu Linux 20.04.3 LTS system with, 2 Tesla T4 GPUs with 64GB RAM.

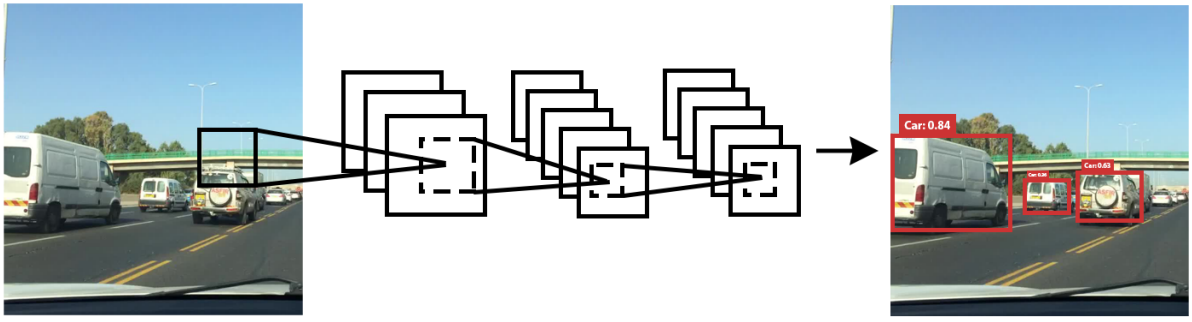


Figure 8.4. YOLOv4, object detector

Dataset and data preparation. The *BDD100k driving dataset* developed by the University of Berkeley, well known in the literature, was used [148]. It presents a total of 100 thousand images, of which 70 thousand for training, like those shown in the Figure 8.5, 10 thousand for validation and 20 thousand for testing.

Figure 8.5. Some images from BDD100k dataset

The dataset contains 10 classes for object detection (e.g., pedestrian, rider, car, etc.) and all of them have been considered. The experiment consists in comparing the performance of a machine learning model trained on daytime and night images with the same model trained first only on daytime images and then only on night images. So only daytime and night images were considered. The dataset presents json files where there are various labels and annotations including the weather conditions,

with which it was possible to discriminate between day and night images and all the others. For a correct experiment, the same number of images was used for the training, except of course the monolithic model which instead uses the sum of the training images of the other two more specific models. The same sub datasets are used for testing. The different models were trained and tested as shown in Table 8.1.

Model	Training Set	Testing Set
Day	27,967 daytime images	3,929 daytime images 3,929 night images
Night	27,967 night images	3,929 night images 3,929 daytime images
Day and night	55,934 daytime + night images	7,858 daytime + night images 3,929 night images 3,929 daytime images
Random	27,967 random images	3,929 random images 3,929 night images 3,929 daytime images

Table 8.1. Datasets used for testing

B. Results

The results, in Table 8.2, showed no improvement in performance with more specific models. Indeed, the monolithic model have better performance on all tests. An experiment was also proposed by training a model with random images between day and night, further demonstration that there is no improvement in performance by training models on specific domains.

Results in meanAP@0.50				
Model	**Day images**	**Night images**	**Day and night**	**Random**
Day	48.59%	41.75%		
Night	39.59%	47.82%		
Day and night	**50.58%**	**49.90%**	50.38%	
Random	48.59%	47.28%		47.35%

Table 8.2. Experiments results

8.3 Proposed Approach

The main goal of this work is designing and implementing the local monitoring of a device within a more complex framework that ensures the safety requirements of autonomous systems during their run-time execution. In Figure 8.6 we have a global vision of the framework architecture:

- **Trusted Membership Service**, a service located in the cloud, checks if a device or a service belongs to the global system, therefore considered truthful.
- **Software Service**, a service located in the cloud, manages a repository containing the updates for device software. Therefore, new models are trained offline and saved here.

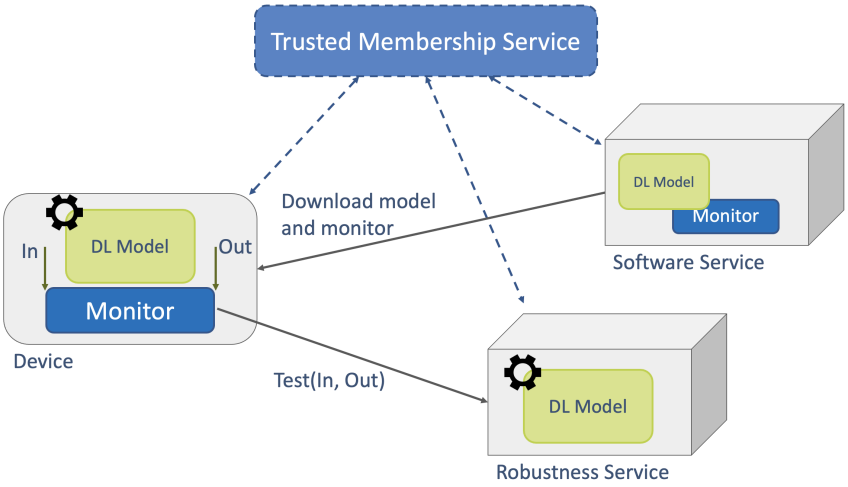


Figure 8.6. Robustness monitoring framework

- **Device**, the system may include several physical devices that contain ML models and their respective monitors.
- **Robustness Service**, a service located in the cloud, checks if the model in the device is performing correctly or needs an update by the Software service.

Figure 8.7 illustrates how we imagine a possible method to implement the monitor. Note that in this case we are focusing on a classifier implemented with a DNN. However, we believe that this approach can easily be extended to other types of machine learning approach.

In the flowchart (Figure 8.7) we have a first phase **off-line** where we train the model with a reliable starting dataset (top half of Figure 8.7). Once trained, we can apply a technique to make the classifier robust against hardware transient faults propagation [50]. The performances of the model and the statistical distributions of each class will be estimated and stored to be used for a future comparison in the second phase. We now move on to an **on-line** phase in which the model comes into operation in our system (bottom half of Figure 8.7). Here inputs and outputs are saved and once stored sufficiently, we can carry out the calculation of the statistical distribution and performances to compare with those made previously.

A large difference between the statistical distributions of training data and model predictions, such that the system is receiving inputs for which it has not been trained, means that the system is working in a context that we had not foreseen (e.g., different weather condition). If this difference is very low, but performances has decreased, also in this case, the system needs to be replaced or re-trained with new (better) data that contains the new contexts. If not, we can be sure that our system is working safely. In this context, the topic of unsupervised domain adaptation can be an interesting improvement to allow the model to improve its performances on a new statistical distribution not very different from the initial one.

8.4 Upcoming Work on Robustness

In this chapter, we have demonstrated the initial experiments, results and proposal for local monitoring of devices in complex frameworks enabled with a robustness service. The future of this work lies in the implementation of techniques in defence

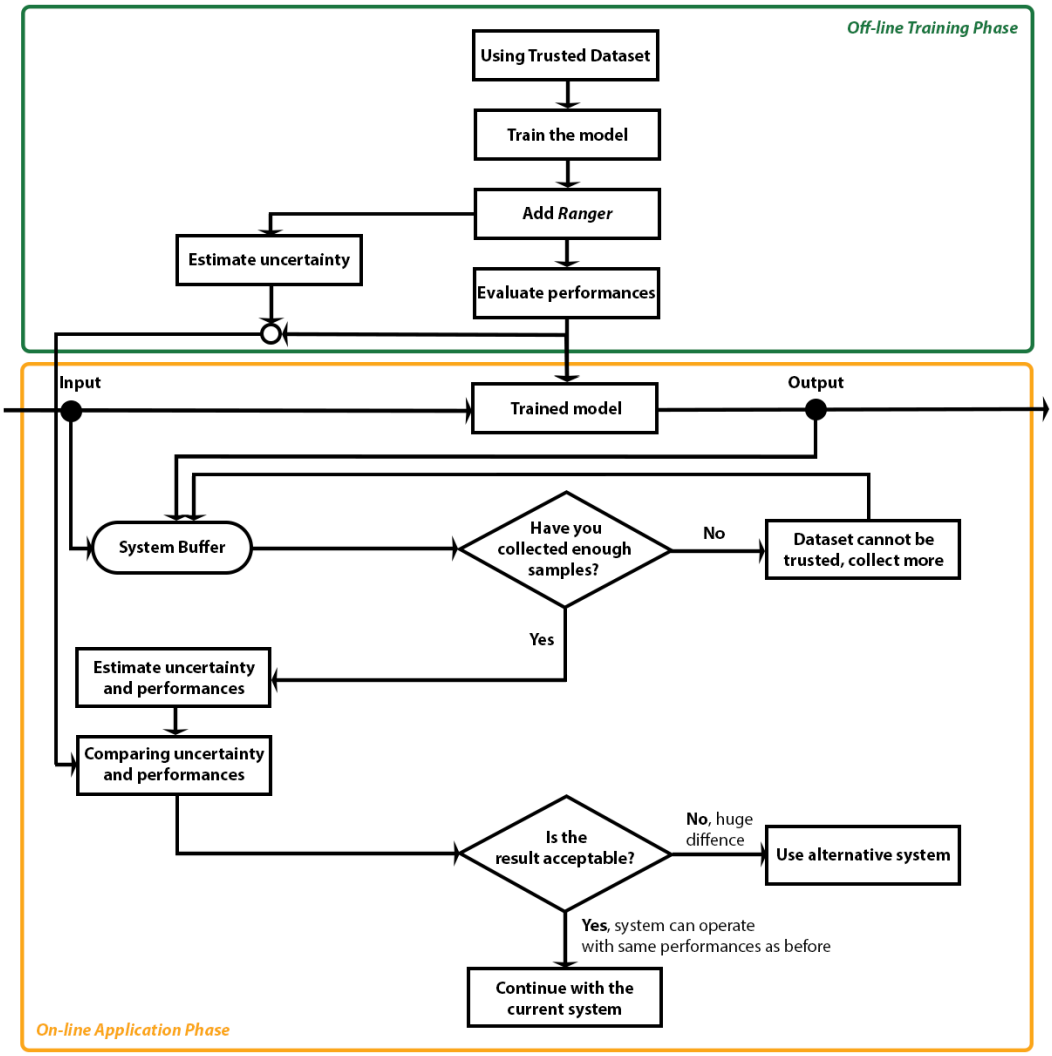


Figure 8.7. Flowchart of the proposed approach

against adversarial examples and noisy input, well known in the literature, to make further contributions to the robustness service. We will continue and implement this proposed architecture during the rest of the project.

9 Overall Achievements and Future Work

We continued to work in Tasks 5.1, 5.2, 5.3 and 5.5 in this period. This deliverable document is organized in 9 chapters including 6 chapters on technical progress. We have explained the structure and per-task progress of the work packages in Chapter 2 (Introduction). Highlights and future plan of our work in different tasks to support the VEDLIoT goals is discussed in the following paragraphs.

In Chapter 3, we presented the collective work of partners on implementing remote attestation using Trusted Execution Environments in IoT devices. Our presented survey comparing state-of-the-art remote attestation schemes, leveraging hardware-assisted TEEs, will prove helpful for deploying and running trusted applications in VEDLIoT use cases. The survey has also shed some light on the limitations of state-of-the-art TEEs and identifies promising directions for future work. To the best of our knowledge, we have also implemented and evaluated the first Wasm runtime running entirely inside Arm TrustZone with full support for remote attestation, optimised explicitly for Wasm to establish trust on hosted applications. We are also designing an automated remote attestation and certification scheme building on standards like RATS for remote attestation to eliminate the TOCTOU problem in certificates.

The Chapter 4 explained our efforts on security support for execution of critical software and communication. With the novel use of MPU, we have provided lightweight solutions to critical problems in TEEs for constrained IoT devices (i.e., TrustZone-M). The preliminary efforts and proposals for a memory protection unit for open-source CPUs (RISC-V based) requires further investigations leading to potential implementation and evaluation.

We discussed initial and ongoing work on incorporating the aspect of safety standards in ML specification, verification, inference and learning in Chapter 5. We will continue to work in identifying the need of a monitoring solution capable of evaluating the performance of AI/ML models, in- and output data quality, communication robustness and capacity in real-time. We also plan on working on learning data selection tool that guarantees the correct number of datasets and the distribution of scenario variability over the datasets for each function/feature.

In Chapter 6, we present our extensive work on the Renode simulator. Due to the modularity of Renode, we were able to autogenerate platform descriptions from CFU Playground samples. With this capability we set up a Continuous Integration environment, testing every change to CFUs, underlying SoCs or the ML software that runs on them.

The architecture and implementation of our trusted verifier service, which is continued from the first 9 months of the project, is included in Chapter 7. In the future, we intend to improve the features of the attestation protocol, adjust some finer details like the time-to-live for devices and implement the Trusted Log beyond conceptualization.

Our last technical chapter (Chapter 8) discusses the architecture and implementation plan and some results of bringing robustness in the VEDLIoT platform. A concrete next step for this work is the complete implementation of the proposed architecture.

10 References

[1] AWS nitro enclaves. https://aws.amazon.com/ec2/nitro/nitro-enclaves/.

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

[3] Genann. https://github.com/codeplea/genann.

[4] GlobalPlatform. https://globalplatform.org.

[5] OP-TEE: *Can heap memory contain executable code?* https://github.com/OP-TEE/optee_os/issues/4396.

[6] Status of Intel SGX and Arm TrustZone support. https://github.com/veracruz-project/veracruz/issues/330.

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

[8] WAMR. https://github.com/bytecodealliance/wasm-micro-runtime.

[9] WASI. https://wasi.dev.

[10] RISC-V ISA and privileged architecture specs. https://bit.ly/3LN9Ili, 2019.

[11] State of the IoT 2020: 12 billion IoT connections, surpassing non-IoT for the first time, November 2020.

[12] ISO/IEC 11889-1: 2015. Information technology–trusted platform module library–part 1: Architecture. 2015.

[13] ISO/PAS 21448:2019. Iso/pas 21448:2019 road vehicles — safety of the intended functionality. 2019.

[14] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *ACM CCS'05*.

[15] Tigist Abera, Raad Bahmani, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Matthias Schunter. Diat: Data integrity attestation for resilient collaboration of autonomous systems. In *NDSS*, NDSS '19, 2019.

[16] Julius Adebayo, Justin Gilmer, Michael Muelly, Ian Goodfellow, Moritz Hardt, and Been Kim. Sanity checks for saliency maps. *Advances in neural information processing systems*, 31, 2018.

[17] Jaehwan Ahn, Il-Gu Lee, and Myungchul Kim. Design and implementation of hardware-based remote attestation for a secure internet of things. *Wireless Personal Communications*, 114(1):295–327, 2020.

[18] AMD. Strengthening VM isolation with integrity protection and more. *White Paper*, 2020.

[19] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in ai safety. *arXiv preprint arXiv:1606.06565*, 2016.

[20] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul F. Christiano, John Schulman, and Dan Mané. Concrete problems in AI safety. *CoRR*, abs/1606.06565, 2016.

[21] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *ACM HASP'13*.

[22] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13. ACM New York, NY, USA, 2013.

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

[24] Antmicro. Renode. https://renode.io. Accessed on: Mar. 4, 2022.

[25] Antmicro. Renode-verilator integration examples. https://github.com/antmicro/renode-verilator-integration/. Accessed on: Mar. 4, 2022.

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

[27] Apple. Apple unleashes M1. https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/, November 2020.

[28] Arm. The realm management extension (RME), for Armv9-A. Technical report.

[29] Arm. TrustZone technology for Armv8-M architecture. https://bit.ly/3LNHtTB, 2018.

[30] ARM. ARM security technology, building a secure system using TrustZone technology. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html, 2019.

[31] Arm. Initial attestation verifier. https://tf-m-user-guide.trustedfirmware.org/tools/iat-verifier/README.html, 2020.

[32] Arm. Confidential compute-architecture (CCA). https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture, 2021.

[33] Arm. TrustZone for Cortex-M. https://www.arm.com/technologies/trustzone-for-cortex-m, 2021.

[34] Arm. Switching between secure and non-secure states. https://developer.arm.com/documentation/100690/0201/Switching-between-Secure-and-Non-secure-states, 2022.

[35] ARM Ltd. Arm® Musca-A Test Chip and Board Technical Reference Manual. https://static.docs.arm.com/101107/0000/arm_musca_a_test_chip_and_board_technical_reference_manual_101107_0000_00_en.pdf, 2018.

[36] ARM Ltd. TrustZone technology for the ARMv8-M architecture. https://static.docs.arm.com/100690/0200/armv8m_trustzone_technology_100690_0200.pdf, 2019.

[37] ARM Ltd. Trusted firmware-m documentation. https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/tfm/index.html, 2021.

[38] ARM Ltd. Musca-a test chip board. https://developer.arm.com/tools-and-software/development-boards/iot-test-chips-and-boards/musca-a-test-chip-board, 2022.

[39] Koorosh Aslansefat, Sohag Kabir, Amr Abdullatif, Vinod Vasudevan, and Yiannis Papadopoulos. Toward improving confidence in autonomous vehicle software: A study on traffic sign recognition systems. *Computer*, 54(8):66–76, 2021.

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

[41] 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.

[42] 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.

[43] 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.

[44] H. Birkholz, D. Thaler, M. Richardson, N. Smith, and W. Pan. Remote attestation procedures architecture. https://tools.ietf.org/id/draft-ietf-rats-architecture-12.html, 2021.

[45] Henk Birkholz, Dave Thaler, Michael Richardson, Ned Smith, and Wei Pan. Remote Attestation Procedures Architecture. Internet-Draft draft-ietf-rats-architecture-02, Internet Engineering Task Force, March 2020. Work in Progress.

[46] Henk Birkholz, Dave Thaler, Michael Richardson, Ned Smith, and Wei Pan. Remote attestation procedures architecture. Technical Report draft-ietf-rats-architecture-12, Internet Engineering Task Force, April 2021.

[47] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *CoRR*, abs/2004.10934, 2020.

[48] Ferdinand Brasser, Kasper B. Rasmussen, Ahmad-Reza Sadeghi, and Gene Tsudik. Remote attestation for low-end embedded devices: the prover's perspective. In *IEEE DAC'16*.

[49] Ernie Brickell and Jiangtao Li. Enhanced privacy ID: A direct anonymous attestation scheme with enhanced revocation capabilities. In *Proceedings of the 2007 ACM workshop on Privacy in electronic society*, WPES '07, pages 21–30, 2007.

[50] Zitao Chen, Guanpeng Li, and Karthik Pattabiraman. Ranger: Boosting error resilience of deep neural networks through range restriction. *CoRR*, abs/2003.13874, 2020.

[51] George Coker, Joshua Guttman, Peter Loscocco, Amy Herzog, Jonathan Millen, Brian O'Hanlon, John Ramsdell, Ariel Segall, Justin Sheehy, and Brian Sniffen. Principles of remote attestation. *IJIS*, 10(2):63–81, 2011.

[52] International Electrotechnical Commission. Iec 61508-1:2010. `https://webstore.iec.ch/publication/5515`, 2010.

[53] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016.

[54] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[55] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium*, USENIX Security 16, pages 857–874, Austin, TX, August 2016. USENIX Association.

[56] Cas J. F. Cremers. The scyther tool: Verification, falsification, and analysis of security protocols. In Aarti Gupta and Sharad Malik, editors, *Springer, CAV'08*.

[57] Cas J. F. Cremers and Sjouke Mauw. *Operational semantics and verification of security protocols*. Springer, 2012.

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

[59] 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.

[60] D. Dolev and A. Yao. On the security of public key protocols. *IEEE TIT*, 29(2):198–208, 1983.

[61] Tommaso Dreossi, Daniel J Fremont, Shromona Ghosh, Edward Kim, Hadi Ravanbakhsh, Marcell Vazquez-Chanlatte, and Sanjit A Seshia. Verifai: A toolkit for the formal design and analysis of artificial intelligence-based systems. In *International Conference on Computer Aided Verification*, pages 432–442. Springer, 2019.

[62] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. Smart: Secure and minimal architecture for (establishing dynamic) root of trust. In *NDSS*, volume 12, pages 1–15, 2012.

[63] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning, 2016.

[64] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. Sprobes: Enforcing kernel code integrity on the trustzone architecture. *arXiv preprint arXiv:1410.7747*, 2014.

[65] GlobalPlatform. *TLS specification of TEE sockets API specification*, February 2021.

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

[67] Google. Cfu playground. `https://github.com/google/CFU-Playground/`. Accessed on: Mar. 4, 2022.

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

[69] Christian Göttel, Pascal Felber, and Valerio Schiavoni. Developing secure services for IoT with OP-TEE: a first look at performance and usability. In *IFIP DAIS'19*.

[70] Christian Göttel, Pascal Felber, and Valerio Schiavoni. Developing secure services for IoT with OP-TEE: A first look at performance and usability. In José Pereira and Laura Ricci, editors, *Distributed Applications and Interoperable Systems*, DAIS '19, pages 170–178, Cham, 2019. Springer International Publishing.

[71] Sven Gowal, Krishnamurthy Dvijotham, Robert Stanforth, Rudy Bunel, Chongli Qin, Jonathan Uesato, Relja Arandjelovic, Timothy Mann, and Pushmeet Kohli. On the effectiveness of interval bound propagation for training verifiably robust models. *arXiv preprint arXiv:1810.12715*, 2018.

[72] 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 *ACM PLDI'17*.

[73] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security 08*.

[74] Matthias Hein, Maksym Andriushchenko, and Julian Bitterwolf. Why relu networks yield high-confidence predictions far away from the training data and how to mitigate the problem. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 41–50, 2019.

[75] Guerney D. H. Hunt, Ramachandra Pai, Michael V. Le, Hani Jamjoom, Sukadev Bhattiprolu, Rick Boivie, Laurent Dufour, Brad Frey, Mohit Kapur, Kenneth A. Goldman, Ryan Grimm, Janani Janakirman, John M. Ludden, Paul Mackerras, Cathy May, Elaine R. Palmer, Bharata Bhasker Rao, Lawrence Roy, William A. Starke, Jeff Stuecheli, Enriquillo Valdez, and Wendel Voigt. Confidential computing for OpenPOWER. In *Proceedings of the 16th European Conference on Computer Systems*, EuroSys '21, page 294–310, New York, NY, USA, 2021. Association for Computing Machinery.

[76] 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.

[77] Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Shaza Zeitouni. SeED: secure non-interactive attestation for embedded devices. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 64–74, 2017.

[78] Intel. XuCode. `https://intel.ly/3rYAhMI`, 2021.

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

[80] Intel Corporation. SGX remote attestation end-to-end example. `https://intel.ly/3AlpKxF`, 2018.

[81] International Organization for Standardization. *ISO 26262:2018: Road vehicles — Functional safety*. International Organization for Standardization, Geneva, 2018.

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

[83] David Kaplan. Protecting VM register state with SEV-ES. Technical report, February 2017.

[84] Arm Keil. Using TrustZone for Armv8-M. `https://www.keil.com/pack/doc/CMSIS/Core/html/using_TrustZone_pg.html`, 2020.

[85] Anum Khurshid, Sileshi Demesie Yalew, Mudassar Aslam, and Shahid Raza. Shield: Shielding cross-zone communication within limited-resourced iot devices running vulnerable software stack. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[86] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. TrustLite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[87] Hugo Krawczyk. Sigma: The 'sign-and-mac' approach to authenticated Diffie-Hellman and its use in the IKE protocols. In Dan Boneh, editor, *Springer CRYPTO'03*.

[88] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *IEEE CGO'04*.

[89] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the 15th European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[90] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Dawn Song, and Krste Asanović. Keystone: A framework for architecting tees. *arXiv preprint arXiv:1907.10119*, 2019.

[91] He Li, Kaoru Ota, and Mianxiong Dong. Learning IoT in edge: Deep learning for the internet of things with edge computing. *IEEE network*, 32(1), 2018.

[92] Wenhao Li, Haibo Li, Haibo Chen, and Yubin Xia. Adattester: Secure online mobile advertisement attestation using trustzone. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, page 75–88, New York, NY, USA, 2015. Association for Computing Machinery.

[93] Linaro. OP-TEE. `https://op-tee.org`.

[94] Linaro. *Frequently Asked Questions: Which API's have been implemented in OP-TEE?* `https://optee.readthedocs.io/en/latest/faq/faq.html`.

[95] Zhen Ling, Junzhou Luo, Yiling Xu, Chao Gao, Kui Wu, and Xinwen Fu. Security vulnerabilities of internet of things: A case study of the smart plug system. *IEEE Internet of Things Journal*, 4(6), 2017.

[96] Kaizheng Liu, Ming Yang, Zhen Ling, Huaiyu Yan, Yue Zhang, Xinwen Fu, and Wei Zhao. On manually reverse engineering communication protocols of Linux-based IoT systems. *IEEE Internet of Things Journal*, 8(8), 2021.

[97] Ziwei Liu, Zhongqi Miao, Xingang Pan, Xiaohang Zhan, Stella X. Yu, Dahua Lin, and Boqing Gong. Compound domain adaptation in an open world. *CoRR*, abs/1909.03403, 2019.

[98] G. Lowe. A hierarchy of authentication specifications. In *ACM CSFW'97*.

[99] Pieter Maene, Johannes Götzfried, Ruan de Clercq, Tilo Müller, Felix Freiling, and Ingrid Verbauwhede. Hardware-Based Trusted Computing Architectures for Isolation and Attestation. *IEEE Transactions on Computers*, 67(3):361–374, 2018.

[100] André Martin, Cong Lian, Franz Gregor, Robert Krahn, Valerio Schiavoni, Pascal Felber, and Christof Fetzer. ADAM-CS: Advanced asynchronous monotonic counter service. In *IEEE DSN'21*.

[101] Arm MBED. PSA initial attestation. `https://os.mbed.com/docs/mbed-os/v6.15/apis/psa-initial-attestation.html`, 2021.

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

[103] Sina Mohseni, Mandar Pitale, Vasu Singh, and Zhangyang Wang. Practical solutions for machine learning safety in autonomous vehicles. *arXiv preprint arXiv:1912.09630*, 2019.

[104] Sina Mohseni, Haotao Wang, Zhiding Yu, Chaowei Xiao, Zhangyang Wang, and Jay Yadawa. Practical machine learning safety: A survey and primer. *arXiv preprint arXiv:2106.04823*, 2021.

[105] Jämes Ménétrey. WaTZ runtime and experiments. `https://github.com/jamesmenetrey/unine-watz`, 2022.

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

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

[108] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://bitcoin.org/bitcoin.pdf`, 2009.

[109] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. TrustZone explained: Architectural features and use cases. In *IEEE CIC'16*.

[110] Zhenyu Ning, Fengwei Zhang, Weisong Shi, and Weidong Shi. Position paper: Challenges towards securing hardware-assisted execution environments. In *Proceedings of the Hardware and Architectural Support for Security and Privacy*, pages 1–8. 2017.

[111] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. VRASED: A verified hardware/software co-design for remote attestation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1429–1446, Santa Clara, CA, August 2019. USENIX Association.

[112] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Norrathep Rattanavipanon, and Gene Tsudik. On the toctou problem in remote attestation. *arXiv preprint arXiv:2005.03873*, 2020.

[113] NXP. *Security reference manual for i.MX8M*, November 2018.

[114] Pedro A. Ortega and Vishal Maini. Building safe artificial intelligence: specification, robustness, and assurance. `https://deepmindsafetyresearch.medium.com/building-safe-artificial-intelligence-52f5f75058f1`, 2018.

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

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

[117] Amaranth Project. Amaranth hdl (previously nmigen). `https://github.com/amaranth-lang/amaranth`, 2021.

[118] Weizhong Qiang, Zezhao Dong, and Hai Jin. Se-Lambda: Securing privacy-sensitive serverless applications using SGX enclave. In Raheem Beyah, Bing Chang, Yingjiu Li, and Sencun Zhu, editors, *Springer SecureComm'18*.

[119] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. Voltjockey: Breaching TrustZone by software-controlled voltage manipulation over multi-core frequencies. In *ACM CCS'19*.

[120] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, Magnus Nystrom, David Robinson, Rob Spiger, Stefan Thom, and David Wooten. fTPM: A Software-Only implementation of a TPM chip. In *25th USENIX Security Symposium*, USENIX Security 16, pages 841–856, Austin, TX, August 2016. USENIX Association.

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

[122] Rick Salay, Rodrigo Queiroz, and Krzysztof Czarnecki. An analysis of iso 26262: Using machine learning safely in automotive software. *arXiv preprint arXiv:1709.02435*, 2017.

[123] Samsung. TEEGRIS. `https://developer.samsung.com/teegris/overview.html`.

[124] Muhammad Usama Sardar, Christof Fetzer, et al. Towards formalization of enhanced privacy ID (EPID)-based remote attestation in Intel SGX. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 604–607. IEEE, 2020.

[125] Muhammad Usama Sardar, Saidgani Musaev, and Christof Fetzer. Demystifying attestation in Intel trust domain extensions via formal verification. *IEEE Access*, 9:83067–83079, 2021.

[126] Vinnie Scarlata, Simon Johnson, James Beaney, and Piotr Zmijewski. Supporting third party attestation for Intel SGX with Intel data center attestation primitives. *White paper*, 2018.

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

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

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

[130] Sanjit A Seshia, Ankush Desai, Tommaso Dreossi, Daniel J Fremont, Shromona Ghosh, Edward Kim, Sumukh Shivakumar, Marcell Vazquez-Chanlatte, and Xiangyu Yue. Formal specification for deep neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 20–34. Springer, 2018.

[131] Carlton Shepherd, Raja Naeem Akram, and Konstantinos Markantonakis. Establishing mutually trusted channels for remote sensing devices with trusted execution environments. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ARES '17, New York, NY, USA, 2017. Association for Computing Machinery.

[132] Carlton Shepherd, Konstantinos Markantonakis, and Georges-Axel Jaloyan. LIRA-V: Lightweight remote attestation for constrained RISC-V devices. In *2021 IEEE Security and Privacy Workshops*, SPW '21, pages 221–227, 2021.

[133] SmartBear. Swagger: Api documentation and design tools for teams. https://swagger.io/.

[134] 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.

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

[136] Trusted Computing Group. Trusted platform module library specification, family ?2.0? https://trustedcomputinggroup.org/work-groups/trusted-platform-module/, 2019.

[137] Furkan Turan and Ingrid Verbauwhede. Propagating trusted execution through mutual attestation. In *Proceedings of the 4th Workshop on System Software for Trusted Execution*, SysTEX '19, New York, NY, USA, 2019. Association for Computing Machinery.

[138] UGOT. Final report on the methods for requirement, specification, performance metrics and verification of context limited ai processing systems. https://vedliot.eu/deliverable/deliverable-d25/, 2022.

[139] UGOT and VEONEER. First report on the methods for requirement, specification, performance metrics and verification of context limited ai processing systems. https://vedliot.eu/deliverable/deliverable-d21/, 2021.

[140] Kush R Varshney. Engineering safety in machine learning. In *2016 Information Theory and Applications Workshop (ITA)*, pages 1–5. IEEE, 2016.

[141] Sébastien Vaucher, Rafael Pires, Pascal Felber, Marcelo Pasin, Valerio Schiavoni, and Christof Fetzer. SGX-aware container orchestration for heterogeneous clusters. In *2018 IEEE 38th International Conference on Distributed Computing Systems*, ICDCS '18, pages 730–741. IEEE, 2018.

[142] Verilator authors. Verilator. https://www.veripool.org/verilator/. Accessed on: Mar. 4, 2022.

[143] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *ACM SOSP '93*.

[144] Yunchao Wang, Zehui Wu, Qiang Wei, and Qingxian Wang. Neufuzz: Efficient fuzzing with deep neural network. *IEEE Access*, 7:36340–36352, 2019.

[145] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. TIMBER-V: Tag-isolated memory bringing fine-grained enclaves to RISC-V. In *NDSS*, NDSS '19, 2019.

[146] Daniel S Weld and Gagan Bansal. The challenge of crafting intelligible intelligence. *Communications of the ACM*, 62(6):70–79, 2019.

[147] Yulei Wu. Robust learning-enabled intelligence for the internet of things: A survey from the perspectives of noisy data and adversarial examples. *IEEE Internet of Things Journal*, 8(12):9568–9579, 2021.

[148] Fisher Yu, Wenqi Xian, Yingying Chen, Fangchen Liu, Mike Liao, Vashisht Madhavan, and Trevor Darrell. BDD100K: A diverse driving video database with scalable annotation tooling. *CoRR*, abs/1805.04687, 2018.

[149] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[150] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y. Thomas Hou. TruSense: Information leakage from TrustZone. In *IEEE INFOCOM'18*.

[151] Shijun Zhao, Qianying Zhang, Yu Qin, Wei Feng, and Dengguo Feng. SecTEE: A software-based approach to secure enclave architecture using TEE. In *ACM CCS'19*.