Abstraction of Trusted Execution Environments as the Missing Layer for Broad Confidential Computing Adoption: A Systematization of Knowledge

QUENTIN MICHAUD, SAMOVAR, Télécom SudParis, Institut Polytechnique de Paris & Thales, cortAIx Labs, Palaiseau, France
SARA RAMEZANIAN, Lund University & Karlstad University, Lund, Sweden, & Karlstad, Sweden
DHOUHA AYED, Thales, cortAIx Labs, Palaiseau, France
OLIVIER LEVILLAIN, SAMOVAR, Télécom SudParis, Institut Polytechnique de Paris, Palaiseau, France
JOAQUIN GARCIA-ALFARO, SAMOVAR, Télécom SudParis, Institut Polytechnique de Paris, Palaiseau, France

Trusted Execution Environments (TEEs) protect sensitive code and data from the operating system, hypervisor, or other untrusted software. Different solutions exist, each proposing different features. Abstraction layers aim to unify the ecosystem, allowing application developers and system administrators to leverage confidential computing as broadly and efficiently as possible. We start with an overview of representative available TEE technologies. We describe and summarize each TEE ecosystem, classifying them in different categories depending on their main design choices. Then, we propose a systematization of knowledge focusing on different abstraction layers around each design choice. We describe the underlying technologies of each design, as well as the inner workings and features of each abstraction layer. Our study reveals opportunities for improving existing abstraction layer solutions. It also highlights WebAssembly, a promising approach that supports the largest set of features. We close with a discussion on future directions for research, such as how future abstraction layers may evolve and integrate with the confidential computing ecosystem.

ACM Reference Format:
Quentin Michaud, Sara Ramezanian, Dhouha Ayed, Olivier Levillain, and Joaquin Garcia-Alfaro. 2026. Abstraction of Trusted Execution Environments as the Missing Layer for Broad Confidential Computing Adoption: A Systematization of Knowledge. 1, 1 (October 2026), 35 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In the last decades, information technologies have seen a large increase in usage, and so has data collection, including private or sensitive information. Furthermore, the size and complexity of these systems makes difficult their management, which is often delegated to third parties. This makes the critical data of an entity (be it an individual, a company, or a government) accessible to these third parties, which are often untrusted.

To tackle this problem, various security mechanisms have successfully been designed to protect data at rest [28] and in transit [55]. However, as computing shifts towards public clouds, the ecosystem severely lacks a way to execute arbitrary

Authors' addresses: Quentin Michaud, quentin.michaud@telecom-sudparis.eu, SAMOVAR, Télécom SudParis, Institut Polytechnique de Paris & Thales, cortAIx Labs, Palaiseau, France; Sara Ramezanian, sara.ramezanian@kau.se, Lund University & Karlstad University, Lund, Sweden, & Karlstad, Sweden; Dhouha Ayed, dhouha.ayed@thalesgroup.com, Thales, cortAIx Labs, Palaiseau, France; Olivier Levillain, olivier.levillain@telecom-sudparis.eu, SAMOVAR, Télécom SudParis, Institut Polytechnique de Paris, Palaiseau, France; Joaquin Garcia-Alfaro, joaquin.garcia_alfaro@telecom-sudparis.eu, SAMOVAR, Télécom SudParis, Institut Polytechnique de Paris, Palaiseau, France.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2026 Association for Computing Machinery.

computations on data while protecting said data against the untrustworthy platform that is providing the computing resources. Confidential computing is a technology that aims to tackle these problems, by executing computations in a secure, hardware-based, attested space named a Trusted Execution Environment (TEE).

A TEE's reliance on hardware fractured the confidential computing ecosystem across hardware vendors who each proposed their own incompatible, in-house solution. Furthermore, most confidential computing solutions require the adaptation of preexisting software. This makes the development of generic, broadly available confidential-computing applications complex. It also forces the developers to focus on the specifics of confidential computing. This can also incur security risks if an application leveraging confidential computing does so in an unsafe way, possibly defeating the purpose of confidential computing.

In order to simplify the work of such developers, several technologies building upon confidential computing emerged. These technologies have various goals, ranging from allowing developers to target different confidential computing solutions at once to supporting common software distribution formats such as containers.

1.1 Motivation

In the age of a growing need for data security, confidential computing became a critical area of research and development. However, despite its potential, the adoption of confidential computing technologies remains hindered by complexities related to their implementation and integration. The main issue comes from the disparity of available hardware and technologies. In a survey on confidential computing research, Feng et al. [20] concluded that *the absence of standardized implementation methods has resulted in diverse approaches by different vendors. This variance introduces technical complexity, bringing challenges even for professional security staff.*

At the core of this challenge lies the need for additional technologies, that we may call abstraction layers, that simplify the interaction with underlying hardware and security mechanisms, making it easier for developers to deploy secure solutions without deep expertise in the low-level details of the underlying confidential computing solutions. These abstraction layers have the potential to bridge the gap between complex security infrastructure and practical applications, promoting broader adoption of confidential computing in a landscape of situations ranging from cloud computing to IoT.

This systematization of knowledge aims to explore and categorize the existing abstraction layers in confidential computing, offering insights into their strengths, weaknesses, and areas for future innovation. By identifying key trends, frameworks, and challenges, this study lays the foundation for a deeper understanding of how abstraction layers can evolve to make confidential computing more accessible and efficient.

1.2 Scope

As abstraction layers are mainly relevant in the way the confidential workloads are executed, this study is focusing on TEEs, ignoring other aspects of confidential computing such as attestation. More precisely, this study is focusing on TEEs implemented on Application-Specific Integrated Circuits (ASICs) that support multiple CPU modes (and as such are able to run OSes relying on preemptive multitasking) and excludes technologies that provide similar security guarantees in other hardware domains, such as GPU TEEs [17, 69], FPGA-based enclaves [18, 78], or microcontrollers TEEs [19, 46, 47]. While these technologies offer valuable security features, their architectures, threat models, and use cases differ significantly from CPU TEEs, making direct comparisons less meaningful. Additionally, GPU TEEs and other non-CPU secure enclaves are still in early research or limited to specialized applications, whereas CPU TEEs

are widely deployed in commercial processors and cloud environments, where having the possibility to develop one application that targets multiple technologies is relevant.

1.3 Contributions

Several surveys tackled the confidential computing ecosystem [20, 37, 43, 59, 71]. However, they mainly focus on the low-level details of the TEE solutions, and they often do not make a clear distinction between TEE technologies and software leveraging these TEEs. Furthermore, they fail to rigorously list and analyze existing abstraction layers, along with considering ways for improving them. This survey focuses on abstraction layers, but keep a discussion on TEEs to introduce concepts needed for discussing abstraction layers.

The main contributions of this survey are the following:

- A detailed overview of the TEE ecosystem, focused on how these solutions are designed. This paper gives essential insights in the security challenges of the different approaches.
- An in-depth systematization of knowledge on existing abstraction layers technologies and their uses, its approaches and limitations.
- A list of open issues and future challenges regarding the development and the deployment of portable applications protected by confidential computing.

1.4 Paper organization

The organization of this paper is summarized in Figure 1. Section 2 provides necessary background on confidential computing and technologies related to the concept of abstraction layers. Then, Section 3 presents the main TEE technologies existing in the ecosystem, with a quick overview of how they work. Section 4 then describes the abstraction layers, existing approaches for developing them, and their inner workings. Discussion presented in Section 5 explores the ups and downs of abstraction layers, especially regarding the approach used to develop then. Finally, we discuss the future of confidential computing abstraction layers in Section 6 before concluding in Section 7.

2 BACKGROUND

This section presents the necessary background for this systematization of knowledge. We first present the confidential computing ecosystem, focusing on the definition of TEE, before presenting various isolation and packaging technologies that are often leveraged by abstraction layers.

2.1 Confidential Computing and Trusted Execution Environments

Confidential computing is a generic term regrouping several mechanisms that we define and explain in the next sections.

2.1.1 Terminology. The Confidential Computing Consortium (CCC) defines confidential computing as *the protection of data-in-use by performing computation in a hardware-based, attested Trusted Execution Environment (TEE)* [16]. The CCC further defines a TEE as *an environment that provides a level of assurance regarding three properties: data confidentiality, data integrity and code integrity*. Confidential computing includes not only the environment in which the computation will take place (the TEE), but also other aspects such as attestation.

In previous work [59], the CCC definitions have been criticized as being imprecise. One of the issues is that the CCC distinguishes between hardware-based TEEs and virtualized, software TEEs, but no formal definition is given for these categories. Since the definition of confidential computing includes hardware-based TEE but not software TEEs, a

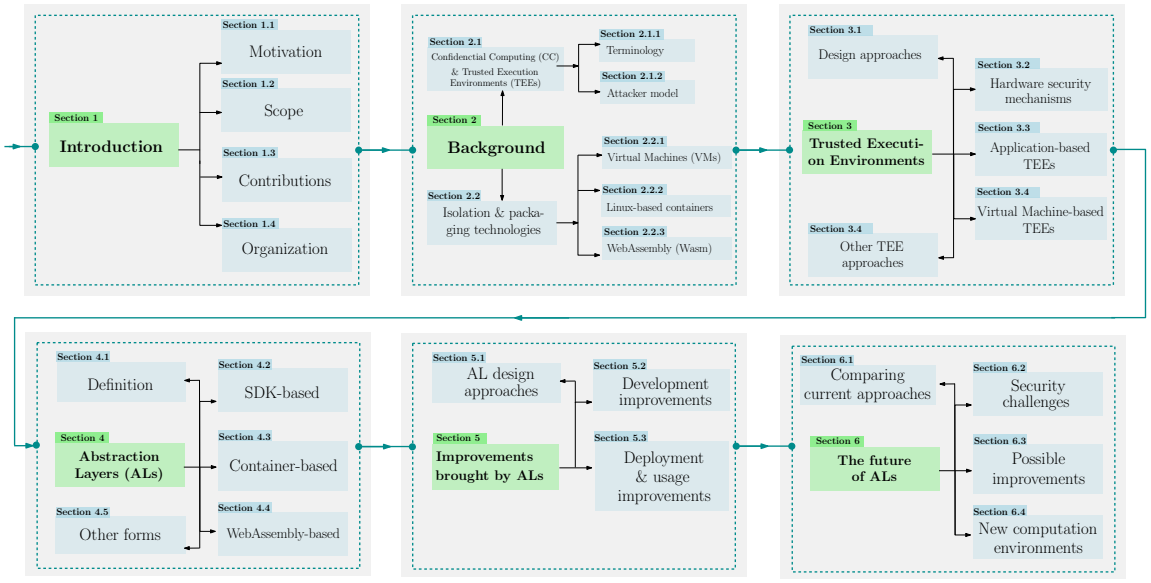

Fig. 1. Organization of the main sections.

lack of definition prevents an accurate listing of TEEs that can be included in the definition of confidential computing. Furthermore, it is unclear if "hardware-based" includes firmware, which can be considered a form of software.

For these reasons, this paper proposes a refined interpretation of the CCC definition of a TEE, that will be used throughout the paper. The goal is to make this definition as broad as possible, so that it encompasses the CCC definition as well as other approaches that may not fit the latter but remain relevant within the ecosystem. This definition is as follows: a hardware-based TEE is an environment that provides a level of assurance regarding data confidentiality, data integrity and code integrity, and which relies on assistance from the hardware to this aim. This means that we consider TEEs that are mainly software-based, but leverage hardware memory protections. In the rest of the paper, we use the term *enclave* to refer to a concrete instantiation of a TEE-protected application. While this term originates from Intel SGX, it is now broadly used, even for applications instantiated within other TEE technologies. The paper will prefer using this term generically instead of other TEE-specific terms.

2.1.2 Attacker model. Confidential computing considers a strong adversary that has access to the complete system software stack, including bootloaders, operating systems, firmware, and hypervisors. From these locations, the adversary can carry out all kind of attacks against the TEE that may compromise the confidentiality of the data inside the TEE. However, a TEE cannot protect the data it contains if the code running inside the TEE is vulnerable. Attacks gaining access to the TEE content by exploiting the application running inside the TEE are out of scope.

Hardware attacks are also out-of-scope. While some TEE technologies may provide additional protection against hardware attacks (e.g. by encrypting memory contents), this attack vector will not be considered as an evaluation factor in this study. Finally, a TEE cannot protect against Denial-of-Service (DoS) attacks, since it is impossible to prevent an adversary having access to low-level software stacks, such as the OS, from stopping the TEE by simply stopping the machine.

2.2 Isolation and packaging technologies

Virtual machines, Linux-based containers, and WebAssembly are popular software packaging and distribution technologies. These technologies are leveraged in different abstraction layers to enable easier development and deployment of confidential computing applications. In the following, we give a brief introduction to these technologies. These approaches are summarized in Figure 2, that illustrates the key elements and the boundaries of each isolation technology, alongside the mechanisms leveraged to interact outside the isolation perimeter.

2.2.1 Virtual machines. Virtual Machines (VMs) provide strong isolation by running full operating systems on virtualized hardware, making them a well-established solution for secure workload execution and multi-tenant environments. VMs offer strict isolation by leveraging hypervisors to separate workloads at the hardware level. While this comes with higher resource overhead and slower startup times compared to more recent technologies, VMs are the only technology that allows packaging and distribution of unmodified complex applications that are closely linked to a specific operating system.

2.2.2 Linux-based containers. Linux-based containers (containers for short) provide a lightweight and efficient approach to application deployment by encapsulating software and its dependencies in isolated user-space environments. Unlike traditional virtual machines, which require full operating system emulation, containers share the host Linux kernel while maintaining process-level separation through isolation mechanisms provided by the Linux kernel, such as namespaces and cgroups. This architecture enables rapid startup times, efficient resource utilization, and seamless portability across different Linux-based computing environments. However, applications unavailable on Linux or that require specific access to kernel features not available in a containerized environment will not be able to run in containers.

Containers have become the backbone of modern cloud-native applications, facilitating microservices architectures and scalable orchestration. Their flexibility and efficiency make them an essential component of modern software development and deployment strategies.

2.2.3 WebAssembly. WebAssembly (Wasm) [56] is emerging as a lightweight and secure alternative to traditional Linux-based containers, offering a portable execution environment that delivers near-native performance. Unlike containers, which rely on OS-level isolation, WebAssembly runs within a small, highly sandboxed, virtual machine, enforcing strict memory safety and reducing the attack surface. This makes it particularly well-suited for untrusted code execution, edge computing, and multi-tenant environments.

Another advantage of Wasm is its portability, which allows applications to be compiled once and executed across different platforms without modification, unlike hardware-accelerated VMs or containers. This greatly helps the application developers, who do not have to distribute and test multiple binaries. Its small footprint and fast startup times make it an attractive option for computing on constrained devices, for example, in the far edge.

By default, Wasm applications are not able to interact with their outside environment because of sandboxing. The WebAssembly System Interface (WASI) is a modular set of system APIs designed to provide Wasm programs with secure access to resources such as files, clocks, pseudo-random number generators, and other resources.

WASI is capability-based, which means programs must be explicitly granted access to a specific resource. This enhances security by granting minimal privileges to Wasm applications by default. The WASI standards aim to be platform-independent, to allow Wasm applications to run consistently across operating systems and devices. WASI interfaces are designed with security in mind to ensure the integrity of the Wasm sandbox.

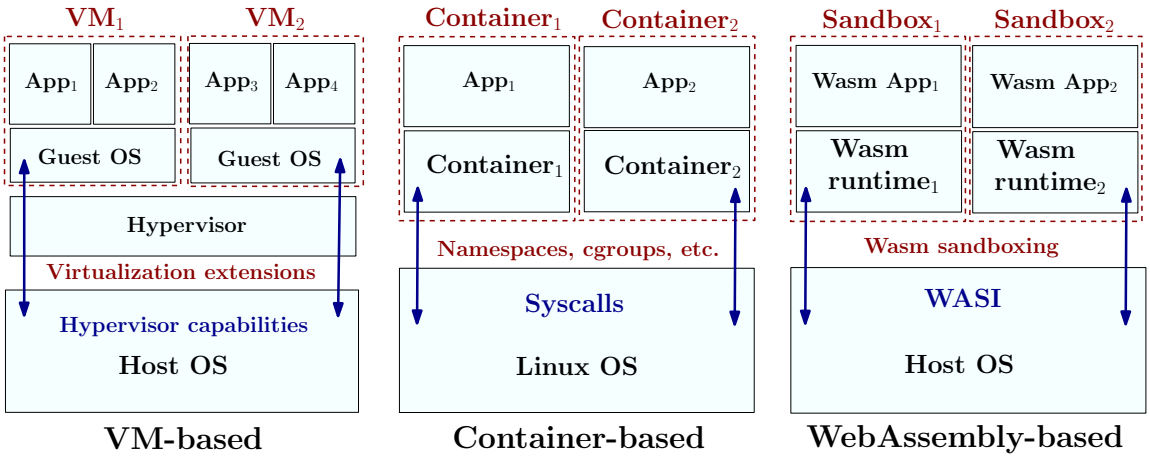

Fig. 2. An overview of each isolation technology and their interactions with the untrusted host. The red dashed color represents each isolation technology. The blue color represents the interfaces that interact outside the isolation solution.

3 TRUSTED EXECUTION ENVIRONMENTS

Trusted Execution Environments are the main components of confidential computing, tasked with executing the enclave application and ensuring the protection of the data while in use. This section presents the main existing TEE technologies within the scope described in the introduction. They are classified by their design approach, and their inner workings are briefly described.

3.1 Introduction to TEE design approaches

The design of a TEE is an important factor in the usability and simplicity of use for application developers. In some cases, the design can even restrict which applications are able to run on the TEE (e.g., multiprocess, access to dedicated hardware). Two main approaches emerge from the ecosystem which are modeled on the common application units used in software engineering. These two units are a process (for a single, standalone software) and a VM (for more complex applications requiring software and dependencies to work tightly together). The TEE using these approaches are designated as App-based TEEs and VM-based TEEs, respectively.

Other, less common approaches also exist. Most of them are coming from research projects without broad use or industry support. The notable exception is Arm TrustZone [5], which is one of the first TEE technologies and is commonly used in embedded and mobile applications. However, these approaches can fill specific use cases or tackle new challenges that current TEE technologies have not yet addressed.

3.2 Hardware security mechanisms

TEEs are built upon a specific CPU architecture, itself based on a specific Instruction Set Architecture (ISA). There are various ISAs that are commonly used in the computing ecosystem relevant for this paper. Most of these ISAs are proprietary (such as x86 or ARM), and the CPUs leveraging those ISAs are often proprietary too. This makes security research on these ISAs complex since the researchers have no solution but to reverse engineer the CPUs to explore their security features. This explains the lack of information on the hardware mechanisms of CPUs based on those ISAs.

A notable exception to this ecosystem is the RISC-V ISA. This ISA is open and belongs to a non-profit that allows anybody to use the RISC-V specifications to build CPUs. This results in an ecosystem where the security solutions are often designed openly, or even directly included in the RISC-V specifications. As such, we are able to explore the RISC-V security mechanisms deeper than those of other solutions.

There are multiple hardware security mechanisms inside the RISC-V specifications that are often used to build TEE solutions [39]. The first one is the Physical Memory Protection (PMP). It allows to define a finite (up to 16, which is the maximal number of PMP registers in the specification) number of memory regions on which access permissions are defined. The permissions are enforced by the hardware implementing PMP.

Another mechanism (that is not yet standardized) aims at enabling multiple hypervisors running in parallel, an essential building block for VM-based TEEs or RISC-V. It is the Supervisor Domain Access Protection. Its goal is to add the support for physical address space isolation, thus allowing for OSes to run securely in parallel.

3.3 Application-based TEEs

Application-based (app-based) TEEs aim to secure a single process in an untrusted Operating System (OS). They aim to protect designated chunks of memory inside a process. These TEEs generally allow for fine-grained control of the application memory, as well as the underlying TEE mechanisms. Most of the time, this control is made available through an SDK. The main benefit of this design is a small Trusted Computing Base (TCB), as the developer explicitly chooses what data to store into the TEE. As the TEE is protecting the minimal amount of memory required, this approach is also generally the fastest one.

However, this comes at a heavy development cost, as existing applications must be adapted to work on the underlying TEE. Furthermore, the security-relevant variables must be correctly identified and secured using the SDK. A wrong use of the SDK could result in the leak of variables into the untrusted host, defeating the purpose of the TEE. Thus, the developers need to be able to accurately identify which memory parts of their code need to be secured. Optimization of the SDK use also involves a performance aspect, as the interactions between classic and protected memory need to be optimized in order to avoid an important performance overhead.

3.3.1 Intel SGX. Intel Software Guard Extensions (SGX) [30] is one of the first TEE technologies, introduced in 2015. The first version of SGX (SGXv1) was initially deployed in Intel's 6th-generation Core processors. In SGXv1, the CPU enforces access control for the enclave memory, known as the Enclave Page Cache (EPC). However, SGXv1 had several drawbacks. The enclave memory was limited (approximately 128MB EPC shared among all enclaves), it lacked support for dynamic memory allocation (pages could not be added after enclave creation), and multithreading support was constrained and difficult to manage securely.

To facilitate application development for SGX, Intel provides a Software Development Kit (SDK) that simplifies the complexity of enclave creation and management. The SDK includes a set of tools, libraries, and example code for building enclave-based applications, and is primarily targeted at C and C++ developers.

SGX's threat model assumes that attackers may control the OS, hypervisor, BIOS, and even certain hardware components, but cannot breach the CPU package. It also explicitly excludes side-channel attacks. Many academic studies have demonstrated that enclaves are vulnerable to various cache, page-fault, and speculative execution attacks. Furthermore, SGXv1 suffered from various attacks weakening the solution [45], among which were hardware-based attacks that are impossible to patch (even if mitigations are available).

Intel announced SGXv2 in 2019 to address several of the practical limitations of SGXv1 and improve the security of SGX. SGXv2 made enclave management more flexible and scalable for modern applications. Key enhancements include dynamic memory management (allowing enclave memory to be dynamically allocated and deallocated at runtime) and improved thread support. Intel announced in 2022 that for the 11th and 12th generations of its processors, SGX would be deprecated for personal chips and that only the Xeon server processors would continue to support SGX.

3.3.2 Keystone. Keystone [36] is an open-source framework for building TEEs on top of the RISC-V instruction set architecture. It is designed to bring an App-based TEE to an open and extensible hardware platform. Keystone leverages the modularity of RISC-V to provide a flexible, formally verifiable TEE infrastructure that can be tailored to specific threat models and use cases.

Keystone introduces the notion of enclave-isolated execution contexts where code and data are protected from interference or inspection by any other software, including the operating system and hypervisor. It achieves this by using the RISC-V Physical Memory Protection (PMP). PMP allows the definition of memory access rules that prevent untrusted software from accessing enclave memory. These rules are managed by a minimal security monitor (SM). This small piece of trusted code running at the highest privilege level (M-mode in RISC-V) is responsible for enforcing isolation and managing enclave lifecycle events.

Keystone is significant for several reasons. First, unlike most x86 or ARM-based TEEs, which are proprietary and hardware-locked, Keystone is fully open-source and can run on open RISC-V hardware. This allows researchers and developers to design, inspect, modify, and formally verify the entire stack [54, 76]. It also allows modification of the solution for specific purposes. Second, Keystone targets standard RISC-V platforms, providing a vendor-neutral foundation for secure computing on a variety of general-purpose CPUs. The simple requirements needed for Keystone that consist of a hardware Root of Trust (RoT), PMP support, and support for the basic U, S, and M modes allow for running enclaves on a large scale of RISC-V hardware, making confidential computing more broadly available than with proprietary alternatives that require specific CPU versions. A notable limitation of Keystone is that its maximum number of enclaves running in parallel is limited to $N - 2$, with N being the number of available PMP entries (as the SM consumes two entries for its own protection, and one entry is required for each enclave).

3.3.3 Penglai. Penglai [21] is an open-source RISC-V TEE that enhances enclave scalability and performance by addressing three key metrics: the size and granularity of secure memory, the number of enclaves, and the startup latency of the enclaves. The hardware security mechanisms standardized in the RISC-V specifications, such as PMP, are limited in these regards, notably regarding the flexibility and the number of enclaves. As such, Penglai solves this problem by proposing two hardware extensions: the Guarded Page Table (GPT) and the Mountable Merkle Tree (MMT).

These custom hardware mechanisms extend the concepts of the standardized PMP while addressing its limitations. More precisely, GPT protects page table pages and enables memory isolation with page-level granularity, and MMT is a new abstraction to achieve on-demand and scalable memory encryption and integrity protection. Together, they remove the constraint of a fixed number of secure memory regions, and enable efficient management of a large number of enclaves. However, Penglai's reliance on GPT and MMT requires specific hardware support, making it less widely deployable than other RISC-V TEEs based only on PMP.

The system operates with a secure monitor in machine (M) mode and continues to rely on PMP for enforcing memory isolation. To reduce enclave initialization overhead, Penglai introduces shadow enclaves, which enable fast enclave instantiation by avoiding expensive memory clearing operations.

3.4 Virtual Machine-based TEEs

The other main approach, VM-based TEEs, aims to protect complete virtual machines instead of only one process. Thus, they allow for protecting complex applications or even multiple, independent applications at once. Their main drawbacks are a much larger TCB, as the trusted code includes the entire OS of the VM, and a generally slower performance because of the virtualization [10]. It is worth noting that, for most solutions, the TEE is built upon the virtualization acceleration solution of the hardware, that allows not to sacrifice too much performance. Their main benefit is that applications can benefit from the security provided by confidential computing without the need for modification. This approach allows for a quicker popularization of the use of confidential computing in various sectors.

3.4.1 AMD SEV. AMD Secure Encrypted Virtualization (SEV) [68] was announced in 2016 and is the first TEE solution for virtual machines. SEV enables the memory of each VM to be transparently encrypted with a unique key managed by a dedicated security processor known as the AMD Secure Processor (ASP). This ensures that only the VM itself can decrypt its memory contents, preventing unauthorized access even by privileged system software running on the host machine.

The security of SEV has been improved throughout the years with extensions, mainly in answer to several attacks [27, 74]. In 2017, SEV-ES (Encrypted State) added encryption and integrity protection to the CPU register states during VM exits, preventing hypervisors from inspecting or manipulating VM execution contexts. In 2020, SEV-SNP (Secure Nested Paging) introduced hardware-based memory integrity protection, strengthening defenses against a broader range of attacks, including replay and remapping attacks, by validating guest memory accesses through a trusted memory encryption engine. Finally, AMD announced SEV-TIO (Trusted Input/Output) in 2023, allowing the establishment of trust and communication between trusted devices.

3.4.2 AWS Nitro Enclaves. AWS Nitro Enclaves (NE) [2] is a confidential computing feature offered by Amazon Web Services (AWS) that enables the creation of isolated, hardened virtual machines within EC2 instances (AWS' virtual machine solution). Built on top of the AWS Nitro System [61], which is the underlying hardware and software platform for modern EC2 (VM) offering, Nitro Enclaves reserve a portion of the instance's CPU and memory to create a separate virtual machine with a reduced attack surface, communicating with the host instance only through a secure virtual socket interface. Applications are converted from an OCI container image to an enclave image using the `nitro-cli` command utility.

Unlike other TEEs, which are tied to specific CPU capabilities and enclave instructions, Nitro Enclaves rely on dedicated Nitro hypervisor technology to carve out secure memory regions within an EC2 instance. These enclaves have no persistent storage, no external networking, and no interactive access, minimizing the attack surface. Nitro Enclaves are available on Intel, AMD, or AWS Graviton (ARM-based CPUs developed by and for AWS) instances, making them theoretically cross-platform. However, they are deeply integrated into the AWS ecosystem and unavailable outside of it.

Although it is not clear which part the dedicated Nitro hardware takes in the protection of Nitro Enclaves, they may be considered a form of hardware-backed TEE, as AWS claims the isolation is enforced by the Nitro System at the hypervisor and memory-controller level. We choose to include Nitro Enclaves in this study as it is frequently mentioned in the confidential computing ecosystem and is generally considered as a TEE by the community.

3.4.3 IBM Secure Execution. IBM Secure Execution (SE) [12] is a TEE specifically designed by IBM for its offer of mainframes on its in-house z/Architecture ISA. Introduced with the IBM z15 and LinuxONE III systems, Secure Execution

provides strong confidentiality and integrity guarantees by creating a secure, encrypted enclave for each virtual machine, known as a Secure Virtual Server.

Under IBM Secure Execution, a VM is first prepared offline. The tenant uses SE-specific tooling to encrypt the guest kernel, initramfs, and boot parameters, and to bind the resulting image to the target system's SE keys. This ensures that only the platform's trusted firmware (the ultravisor) can decrypt it. When the VM is started, the standard hypervisor delegates launch control to the ultravisor, which verifies the image metadata, decrypts the payload, and instantiates the guest inside a hardware-enforced protected address space. During execution, all guest memory is isolated from the host and hypervisor: pages remain encrypted when stored externally, access by host software is blocked, and integrity protections prevent tampering. As a result, the VM runs with confidentiality and integrity guarantees against both a potentially malicious host OS and administrative users, while preserving normal VM lifecycle operations.

SE only supports Linux, as its architecture is based on KVM. It is supported by Linux distributions certified for IBM Z platforms, such as Red Hat Enterprise Linux, SUSE Linux Enterprise Server, and Ubuntu.

3.4.4 Arm CCA. Arm Confidential Compute Architecture (CCA) [6] was introduced in 2021 along with the Armv9 architecture. CCA is a key feature of the Armv9-A architecture and makes confidential computing available to a wider audience for ARM machines. The first Arm TEE, TrustZone (presented in 3.5.1), was limited, and developing for the TrustZone TEE requires a strong collaboration with Arm and acceptance of related collaboration terms. Arm CCA represents a significant evolution in secure computing for the Arm ecosystem, offering a scalable and flexible path to deploy confidential workloads, compared to its limited TrustZone predecessor.

At the core of Arm CCA is the concept of Realms, a new form of secure execution environment for protecting virtual machines. Realms are instantiated and managed by a Realm Management Monitor (RMM), a privileged firmware component that operates beneath the hypervisor. Realms execute in a dedicated CPU mode and memory space that is cryptographically isolated using Arm's Memory Tagging Extension (MTE) and Secure EL2 mechanisms, along with a hardware-based component known as the Realm Management Extension (RME).

Arm CCA itself does not provide any implementation of an RMM. Islet [32] is an open source implementation of the Realm Management Monitor (RMM) written in Rust. Islet is backed by the Arm company and other industrial actors and is considered as a reference implementation of the CCA RMM. Islet provides an open source SDK written in Rust. The Islet SDK enables developers to create applications for ARM CCA enclaves, offering built-in support for essential security primitives such as data sealing and secure channel establishment.

3.4.5 IBM PEF. The Protected Execution Facility (PEF) [29] is IBM's TEE designed for confidential computing on the Power ISA, introduced with the POWER9 architecture. The Power ISA is developed by a foundation led by IBM, and is used in many IBM products. PEF enables the creation of secure virtual machines (SVMs).

SVMs are managed by a minimal trusted firmware component, called the ultravisor. The ultravisor controls the transitions into and out of secure state, enforces access control policies, and manages the cryptographic protection of memory belonging to secure VMs. Memory pages belonging to a secure VM are subject to hardware-enforced encryption and integrity checks before they leave the secure state or are paged out to normal memory. The ultravisor ensures that any page movement from secure to non-secure requires encryption (confidentiality) and cryptographic integrity (detection of tampering) before being made accessible to the hypervisor or system software. In turn, when those pages are later brought back into secure state, they must pass the integrity check and be decrypted before guest access.

3.4.6 Intel TDX. Intel Trust Domain Extensions (TDX) [31] is a hardware-based confidential computing technology introduced by Intel in 2023 on its 4th generation of Intel Xeon processors to provide enhanced security for virtualized environments. It enables the creation of Trust Domains (TDs), which are hardware-isolated virtual machines.

At the core of Intel TDX is the Secure Arbitration Mode (SEAM), a new CPU mode that hosts the TDX module responsible for managing TDs. This module operates in a reserved memory region defined by the SEAM Range Register (SEAMRR) and functions alongside the existing virtualization infrastructure. Memory confidentiality and integrity for each Trust Domain are enforced using hardware mechanisms. TDX leverages Intel Total Memory Encryption, Multi-Key (TME-MK), so that each TD's memory is encrypted under its own key.

3.4.7 RISC-V CoVE. The RISC-V Confidential VM Extension (CoVE) [57], previously named Application Processor Trusted Execution Environment (AP-TEE), is an open standard designed to enable confidential computing on RISC-V platforms. It first appeared in 2022 and is in the process of being standardized as an official RISC-V extension. It provides a hardware-assisted framework for running TEE Virtual Machines (TVMs) that are isolated from untrusted software components, including the host operating system and hypervisor. This isolation is achieved through a combination of RISC-V privilege modes, new Supervisor Binary Interface (SBI) extensions, and a minimal security monitor known as the Trusted Security Manager (TSM).

CoVE necessitates several hardware extensions, such as the H-mode for enabling hardware virtualization support, Supervisor Domain Access Protection for isolating physical memory between different supervisor domains (a more complete and flexible approach than PMP, as PMP is insufficient for creating a VM-based TEE), and Advanced Interrupt Architecture (AIA) for fine-grained interrupt control. Some of these extensions are still in the process of being officially standardized.

While CoVE lays the groundwork for secure VM-based execution on RISC-V systems, its specifications are not completely ready for general use and no hardware exists yet for some extensions needed by CoVE. However, it is a significant development in the TEE ecosystem, making it relevant enough to be included in this study. Furthermore, a research project by IBM, named Assured Confidential Execution (ACE) [51] implemented CoVE's specification by emulating hardware features that are not yet available. This project also works on the provability of a TEE Security Monitor [50] (the trusted code responsible for managing enclaves and enforcing isolation).

3.5 Other TEE approaches

Some TEEs are using other approaches that do not fit into the application or virtual machine model. For the case of TrustZone, the approach is unique because, as the first TEE implementation, it made choices that were not reused for further iterations. Finally, HyperEnclave proposes a flexible design based on both VM- and App-based TEEs, without requiring specialized hardware.

3.5.1 Arm TrustZone. Arm TrustZone [5], introduced in 2004, represents Arm's first approach to hardware-based isolation. It is implemented in both the Arm Cortex-A and Cortex-M processor families, starting from ARMv6. It establishes two distinct execution environments, the secure world and the normal world. Each world has its own operating system, generally a rich operating system for the normal world, and a specialized secure operating system running in the secure world. The secure operating system governs the secure world and is responsible for managing access to protected resources.

TrustZone supports only a single secure world, with a fixed secure operating system. This limitation makes it unsuitable for use cases involving multiple mutually distrusting workloads or dynamic provisioning of secure services.

Additionally, TrustZone lacks support for generic virtualization, a feature only introduced later with Armv8.4-A through the addition of Secure EL2 (S-EL2), accessible solely from within the secure world. These limitations have motivated the development of more flexible secure computing architectures, such as Arm's Confidential Compute Architecture (CCA).

Despite its constraints, TrustZone has seen widespread deployment in consumer devices. It has been used to enhance the security of Android smartphones, with platforms like Samsung Knox [58] leveraging TrustZone to protect sensitive operations and data.

TrustZone on its own provides hardware isolation to create a TEE, but no software to leverage it. OP-TEE[1] is a widely used open-source runtime environment that is often used as the software layer on top of TrustZone. It provides an environment for creating trusted applications. OP-TEE adheres to the TEE architecture and the Global Platform API standard (GP API)[2], which is structured around three main components: a client application, a specialized Linux driver, and the OP-TEE OS. In this setup, the operating system in the normal world is also known as the rich execution environment (REE). Host applications operate within this normal world and act as clients to trusted applications (TAs) in the secure world. These host applications use client APIs for communication.

3.5.2 HyperEnclave. HyperEnclave [33] is a TEE technology that aims to overcome the limitations of existing vendor-specific TEEs like Intel SGX or AMD SEV. Instead of relying on specialized hardware features, HyperEnclave leverages standard virtualization extensions available on commodity processors to provide secure and isolated execution environments. It achieves this by running enclave processes inside lightweight virtual machines managed by a privileged security monitor called RustMonitor. This monitor, written in Rust, enforces strong isolation between the enclave and the potentially compromised operating system, ensuring that sensitive data and code remain protected.

A key advantage of HyperEnclave is its cross-platform design. It can run on x86 systems from different vendors without hardware-specific dependencies. The system also maintains compatibility with Intel SGX applications, allowing existing enclave code to be reused with minimal changes. This makes HyperEnclave a mixed approach between VM-based and app-based TEEs, as it relies on VMs for the isolation but supports the app-based approach of SGX.

3.6 TEEs takeaways

The key findings of this section are that today's TEE ecosystem remains highly fragmented, with vendors adopting divergent abstractions, programming models, and trust assumptions. An overview of TEE technologies presented in this section is available in Table 1. Two dominant approaches have emerged: application-based TEEs, which isolate individual programs or functions within lightweight enclaves, and VM-based TEEs, which protect entire virtual machines using hardware-enforced boundaries.

Application-based designs offer fine-grained isolation and smaller attack surfaces, but they often require substantial re-engineering effort and expose developers to platform-specific constraints. In contrast, VM-based solutions provide stronger compatibility with existing software stacks and simplify deployment, though at the cost of larger trusted computing bases and potentially higher overhead. While application-based TEEs offer better guarantees from a security point of view, VM-based approaches took over the cloud ecosystem thanks to their better usability. However, for other ecosystems such as the edge or embedded, the app-based TEEs remain essential because of their lightness. This dichotomy, along with other approaches that are also present in the ecosystem, underscores the lack of convergence in the field and highlights the need for unified abstractions and tooling to support secure and practical use of TEEs at scale.

[1] https://www.trustedfirmware.org/projects/op-tee
[2] https://globalplatform.org

Table 1. Overview of Trusted Execution Environments. The *Year* column refers to the availability of technology and not its announcement date, except for CoVE where no hardware is currently available.

Technology	Acronym	Year	Hardware	Approach	Maturity
Arm TrustZone [5]	TZ	2004	ARM	Other	Industry
Intel Software Guard Extensions [30]	SGX	2015	x86	App-based	Industry
AMD Secure Encrypted Virtualization [68]	SEV	2016	x86	VM-based	Industry
Keystone [36]	KS	2019	RISC-V	App-based	Research
AWS Nitro Enclaves [2]	NE	2020	x86, ARM	VM-based	Industry
IBM Secure Execution [12]	SE	2020	z/Architecture	VM-based	Industry
Arm Confidential Compute Architecture [6]	CCA	2021	ARM	VM-based	Industry
IBM Protected Execution Facility [29]	PEF	2021	PowerPC	VM-based	Industry
Intel Trust Domain Extensions [31]	TDX	2021	x86	VM-based	Industry
Penglai [21]	PG	2021	RISC-V	App-based	Industry
HyperEnclave [33]	HE	2022	x86	Other	Research
Confidential VM Extension [57]	CoVE	2022*	RISC-V	VM-based	Research

4 ABSTRACTION LAYERS

Abstraction layers are software solutions built on top of one or several TEEs to ease some aspects of developing and deploying TEE-protected applications. A more detailed context is given before presenting the abstraction layers ecosystem, categorized by the approach chosen for the software abstraction.

4.1 Introduction and definition

While TEEs provide strong hardware-enforced isolation for secure computation, their low-level nature and disparate inner workings present significant challenges for developers and system integrators. Programming directly with TEEs often requires expertise in cybersecurity, enclave development, and memory management, which can be complex and error-prone. Additionally, different TEE implementations have distinct APIs and security models, complicating portability and interoperability across platforms.

To address these issues, additional software layers, that we will refer to as *abstraction layers*, are needed to simplify the development and the deployment of confidential computing applications. These abstraction layers are software solutions that can provide additional features, including but not limited to multiplatform or multi-architecture deployment, support for additional programming languages, or deployment solutions. These features are aimed at improving the easiness of either the development or the deployment of TEE-protected application. Abstraction layers are built upon at least one TEE, but frequently support several of them.

This enables developers to focus on application logic rather than low-level security details, and makes confidential computing more broadly available for various applications. Furthermore, these abstraction layers can facilitate integration with existing software ecosystems, making confidential computing more accessible and reducing the risk of implementation errors that could compromise security.

The main goal of an abstraction layer is to enable the execution of an unmodified application, packaged into a designed format, inside a TEE. The interactions between an abstraction layer, a TEE (in the two main approaches), and an application are detailed in Figure 3. An abstraction layer intervenes at two main steps of an application's lifecycle: first, when the application is being processed from a standard application to one supported by the abstraction layer. Second, on top of the supported TEEs, making the intermediary between the processed app and the TEE solution.

There are various approaches for abstracting a TEE. In the following, we detail the SDK-based, the container-based, and the WebAssembly-based approaches, as well as other, less common approaches.

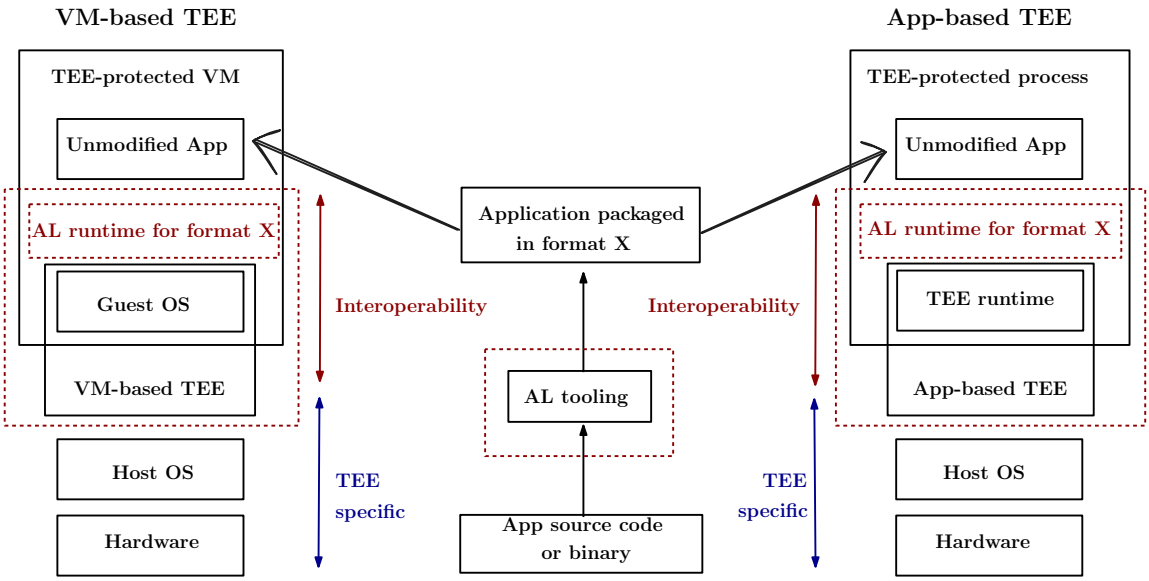

Fig. 3. Representation of where the abstraction layer lays inside various TEE approaches. Dotted red represents the areas where abstraction layers are intervening. Blue arrows represent the classic TEE protection.

4.2 SDK-based abstraction layers

SDK-based abstraction layers provide a software solution that simplifies the development of secure applications by exposing a high-level API and runtime environment, while relying on the underlying TEE hardware for isolation and protection. These solutions typically follow a client-enclave model, where an application executed on the untrusted OS communicates with a secure enclave (or trusted application) using well-defined interfaces.

This approach is inspired by app-based TEEs that, for the majority, rely on an SDK to build an application that can run on said TEE. The main downside of the SDK-based approach is that applications still need to be developed for a TEE environment, without the possibility to port existing applications out of the box.

4.2.1 Open Enclave SDK. Open Enclave SDK [60] is an open-source, hardware-agnostic SDK designed to unify and simplify the development of TEE applications across multiple TEE platforms. The SDK allows building applications using C or C++, and currently supports SGX and TrustZone. Developed originally by Microsoft, it provides a consistent, high-level programming model that abstracts away the hardware-specific details of each TEE, enabling developers to write portable and maintainable enclave applications.

The core functionalities of Open Enclave SDK mirror those of typical TEE SDKs: it supports enclave creation, secure communication between trusted and untrusted components, cryptographic services, and secure storage. However, it introduces several enhancements over using vendor-specific TEE SDKs directly. Most notably, it offers TEE portability, allowing a single codebase to target multiple hardware backends without modification. This is achieved through a unified API and a pluggable architecture that routes platform-specific behavior to appropriate backends at runtime or

compile time. In addition, Open Enclave SDK improves the development and testing workflow with features such as a simulation mode, which allows enclaves to be executed and debugged without hardware support.

4.2.2 Asylo. Asylo [9] is an open-source framework developed by Google to simplify the creation of applications running inside TEEs. It only supports Intel SGX, but the project was aiming to support multiple TEE solutions with its flexible design. Its design aims to abstract away low-level, hardware-specific complexities and offer a high-level, portable programming interface for secure applications.

Unlike TEEs that follow a split-world model, requiring developers to manually partition code into trusted and untrusted components with narrowly defined communication interfaces, Asylo supports a more unified approach. It enables the compilation of complete applications like Redis or SQLite into a TEE with minimal or no changes to the codebase, thanks to its support of the C and C++ standard libraries.

Internally, Asylo uses a sandboxed runtime and a pluggable backend system that allows the same application to run either in a real enclave or in a simulation mode for easier debugging. It also includes support for data sealing and enclave lifecycle management. Asylo is notable for its whole-application enclave model, which stood in contrast to the classic approach of modular and interface-driven SDKs found in the confidential computing ecosystem.

4.2.3 Teaclave. Teaclave [8] is an open-source collection of developer-friendly SDKs, aimed at enabling direct development of custom TEE-protected applications. It was originally designed as a general-purpose secure computing framework for TEE-protected Function as a Service (FaaS) applications. This framework was relying on WebAssembly as its general-purpose execution target, but the Teaclave community valued direct SDK development because FaaS applications often have a very small codebase.

This collection of SDKs allow developers to use Rust, with an experimental SDK targeting Java. The Rust SDKs are currently supporting SGX and TrustZone, and the Java SDK supports SGX only, but the project aims to extend support to other TEEs.

4.3 Container-based

Container-based solutions built on top of TEEs extend the familiar container abstraction to support TEE workloads. These solutions aim to combine the ease of deployment and isolation properties of containers with the strong hardware-backed security guarantees offered by TEEs. They allow users to run entire containerized applications inside a protected enclave. These solutions are based on widely used standards such as the OCI runtime spec [49] and image spec [48].

This approach features compatibility with standard container tooling, allowing developers to build and package applications using familiar tools. Combined with support for encrypted I/O and storage, this enables applications to securely process sensitive data without exposing it to the host. Some systems also integrate with Kubernetes, offering TEE-aware scheduling, and may provide enhanced logging, auditing, and policy controls. It provides an intuitive way to deploy and scale confidential workloads, abstracting away the low-level complexity of enclave programming while leveraging modern DevOps practices and hardware-enforced security.

4.3.1 SCONE. SCONE (Secure CONtainer Environment) [7] is a secure container framework designed to run containerized applications inside SGX enclaves with minimal changes. First introduced in 2016, it is presented as a practical approach to securing cloud workloads using TEEs.

SCONE targets containerized applications and enables them to run in an SGX enclave without requiring extensive refactoring. The core idea is to provide a transparent execution environment by integrating a custom libc inside the

enclave, which becomes part of the TCB. This libc intercepts and mediates all system calls, securely forwarding them to the untrusted host operating system using an asynchronous system call mechanism. This design allows applications to benefit from SGX protections without being aware of enclave-specific programming models. The SCONE libc also protects data at rest and in-transit by decrypting / encrypting data going through file descriptors, protecting files that are stored outside the enclave, and using TLS to secure communications.

To support efficient execution, SCONE also employs a kernel module on the host side to facilitate communication between the enclave and the OS, while maintaining confidentiality and integrity guarantees. This setup allows SCONE to handle I/O operations, signals, and thread management securely and efficiently, despite the limited functionality provided by the SGX hardware.

4.3.2 Gramine. Gramine [15, 52, 64] is a lightweight library OS designed to run unmodified Linux applications in a TEE. It provides a minimalist kernel-like environment that emulates Linux system services inside an enclave, enabling legacy applications to benefit from confidentiality and integrity protections without extensive rewriting. Gramine is primarily targeting Intel SGX, with recent work [34] porting Gramine to TDX. Furthermore, this study mentions that the TEE-agnostic way the Gramine TDX porting is designed should allow for easy porting to other VM-based TEEs such as SEV.

Gramine operates by packaging the application together with a small runtime into a manifest file that specifies how the application interacts with the outside world. Inside the enclave, Gramine handles syscalls, file accesses, and I/O through a secure wrapper, ensuring that all interactions with the untrusted host OS are tightly controlled and, where necessary, encrypted or authenticated.

In addition to direct SGX execution, Gramine has a specialized tool called the Gramine Shielded Container (GSC) tool, which integrates with Docker to allow the conversion of regular OCI containers into SGX-enabled containers. This makes it easier to deploy confidential workloads using familiar container workflows without needing to manually manage the enclave configuration.

4.3.3 libkrun. Libkrun [38] is a lightweight library designed to run processes inside a minimal virtual machine using KVM. Coupled with crun, a low-level container runtime developed by Red Hat, libkrun is able to run OCI-compatible workloads. Libkrun acts as a low-level runtime that enables strong isolation by encapsulating containers within virtual machines, while preserving compatibility with OCI container specifications through crun. Its goal is to offer a secure and efficient alternative to traditional container runtimes, especially in environments that demand stronger isolation than what classic Linux containers provide.

A specialized variant, libkrun-sev, extends this model by integrating SEV support. With libkrun-sev, container workloads not only benefit from virtualization-based isolation but also gain confidential computing protection for SEV. Support is currently limited to SEV, and it is unclear whether future versions will include other TEEs. One notable improvement to libkrun-sev is its integration into Buildah, a popular container image-building tool also developed by Red Hat. A new command, `mkcw`, allows users to convert a standard OCI container into an SEV-protected confidential container, streamlining the process of deploying secure workloads without altering application code.

4.3.4 CoCo. Confidential Containers (CoCo) [65] is an open-source project that brings confidential computing capabilities to the cloud-native ecosystem, enabling Kubernetes-based workloads to run with strong hardware-backed isolation. CoCo builds on earlier efforts from the Kata Containers project, which introduced lightweight virtual machines to improve container isolation by leveraging KVM-based virtualization. While Kata Containers provided an effective

foundation, it became clear that its architecture was not suitable for creating a confidential computing framework supporting multiple TEEs. This realization led to the creation of CoCo as a broader and more modular solution for supporting diverse confidential hardware technologies.

Internally, Confidential Containers orchestrate secure workloads by running each container inside a lightweight TEE-backed virtual machine. CoCo integrates with Kubernetes and container runtimes to transparently handle secure bootstrapping and secret provisioning, aiming to keep the developer and operator experience close to standard cloud-native tooling. A key goal of the project is to abstract the differences between TEE implementations, enabling developers to deploy confidential workloads without needing to adapt their code for each platform. CoCo currently supports SGX, SEV, TDX, and SE, with planned support for PEF.

4.3.5 Anjuna Seaglass. Anjuna Seaglass [4] is a commercial confidential computing platform designed to simplify the deployment of secure applications using hardware-backed TEEs, with support for SGX, SEV, and NE. Seaglass supports deployment in containerized environments, including Kubernetes, using unmodified applications packaged in OCI containers. Its runtime stack handles the secure provisioning of secrets and secure I/O operations transparently. It also handles security policies, which govern how applications interact with sensitive data and ensure compliance with said policies throughout the lifecycle of a workload. These policies are enforced at runtime through integration with the platform's secure provisioning process.

4.4 WebAssembly-based

In this section, we present *Wasm Trusted Runtimes* that we consider in this study. A Wasm Trusted Runtime refers to a secure and isolated environment for executing Wasm modules in TEEs. This environment is often designed to ensure confidentiality, integrity, and attestation of computations.

A Wasm Trusted Runtime can be viewed as a *two-way sandbox environment*, where the TEE provides hardware-level security, protecting the application from the host, while the Wasm runtime that is coupled with the TEE enforces software-level security, and protects the TEE and the host from the application. The following Wasm trusted runtimes and frameworks are developed to run untrusted or third-party Wasm code safely in a TEE.

Work on assessing Wasm runtimes has already been conducted in the past [72, 77]. While these surveys do not focus on trusted runtimes explicitly, they provide us with a base analysis of the runtimes that have been modified to make them trusted runtimes in the solutions presented below.

4.4.1 Enarx. Enarx [22] is an open-source framework that provides a Wasm trusted runtime. The current implementations (Enarx 0.7.1, January 2023)[3] include support for Intel SGX and AMD SEV.

The Enarx project was initially conceived with the objective of developing a CPU architecture-independent runtime that facilitates application development without necessitating code modifications for compatibility across diverse hardware platforms. Just as other confidential computing platforms, the core of Enarx's design philosophy is to provide protection for data-in-use, referring to data residing within the processor during program execution.

Rather than asserting the trustworthiness of the host environment, Enarx adopts an architectural approach that minimizes the necessity to trust the surrounding system components. It does so by operating under the assumption that all entities external to the enclave (a *keep* in Enarx's terminology) are inherently untrusted. In other words, only the enclave and the processor along with its firmware are trusted. Consequently, components such as the hypervisor,

[3]https://github.com/enarx/enarx/releases

operating system kernel, and user-space services are denied access to both the application code and the data encapsulated within the enclave.

Enarx uses the following components in its architecture: (1) A Loader (in app-based TEEs) or a Virtual Memory Manager (in VM-based TEEs). (2) A Linux microkeonlyrnel (shim). (3) A Wasm runtime (Wasmtime). (4) A WASI Interface.

The goal of the Enarx components is to ensure that workloads are isolated and protected via an abstraction layer. Enarx uses WASI and the Wasm runtime, specifically Wasmtime, to provide APIs necessary for secure code execution and communication, that allows code portability across platforms without compromising security.

To securely execute a code in Enarx, an enclave is initiated on the host's machine. The following three steps are then executed: (1) **Authenticity Check.** Enarx verifies the authenticity of the application that the user intends to deploy. (2) **Packaging.** Then, Enarx uses cryptographic algorithms to encrypt the user's application and data. (3) **Provisioning.** Finally, application and data are executed in the enclave.

Enarx monitors code execution for abnormal behavior, such as unauthorized system calls or disallowed memory requests. If such behavior is observed, the enclave is terminated.

4.4.2 TWINE. TWINE (trusted Wasm in enclave) [40] is an open-source, lightweight, and embedded Wasm virtual machine running in Intel SGX. It was developed by Ménétrey et al. in 2021. The authors expanded their work in 2023 and evaluated the performance of TWINE [42].

TWINE provides hardware security by using the Intel SGX enclaves and software security by using Wasm to run applications, in a two-way sandbox environment. TWINE provides an environment to execute Wasm applications inside Intel SGX and consists of a Wasm runtime (that runs inside the enclave) and a WASI (that acts as the middleman between the runtime and the operating system).

In order to implement TWINE, several runtimes were evaluated as potential candidates. These runtimes were Wasmtime, Wasmer, Lucet[4], WAVM[5], Wasm3[6] and WAMR. Most of these runtimes were excluded due to limited capabilities, such as large trusted computing base and therefore, more potential vulnerabilities, or lack of built-in TEE compatibility. Only WAMR met the requirements for a trusted WebAssembly runtime.

One of the reasons to use WAMR in TWINE is its "out-of-the-box" implementation capabilities for Intel SGX. The WAMR toolkit includes an ahead-of-time (AOT) compiler, allowing Wasm applications to be compiled into native code using LLVM before they are sent to TWINE's enclave. As a result, TWINE does not include a Wasm interpreter and is only capable of running AOT-compiled applications. The key advantage of this approach is that native code execution is faster than code interpretation. Additionally, the Wasm runtime has a smaller memory footprint compared to an interpreter, a vital consideration for SGX environments and cloud/edge computing scenarios. Embedding a just-in-time (JIT) compiler was not pursued in TWINE by its developers, because incorporating LLVM into an enclave would require adapting the code to meet SGX's restrictions. The implementation of TWINE has since been incorporated in the upstream WAMR project.

4.4.3 WaTZ. In 2022, a framework called WaTZ [41] was proposed by Ménétrey et al. to securely execute Wasm codes inside Arm TrustZone. There is an overlap in scholars who developed WATZ and TWINE. WaTZ shares many design

[4]https://bytecodealliance.github.io/lucet/
[5]https://github.com/wavm/wavm
[6]https://github.com/wasm3/wasm3

choices with TWINE, including the use of WASI, the reliance on WAMR, and the use of the Ahead-Of-Time (AOT) runtime model. However, WaTZ was developed independently of TWINE to adapt to TrustZone's unique design.

WaTZ has a significantly small footprint that reduces the attack surface, offering fewer opportunities for attackers to exploit compared to larger programs. WaTZ is specifically designed to provide a Wasm runtime for small, edge-scale Arm processors.

WaTZ uses the OP-TEE runtime. This utilization, however, had one drawback: OP-TEE's memory management API lacks the ability to adjust page protections to designate them as executable. To address this, the developers of WaTZ modified OP-TEE to enable such functionality for Trusted Applications.

4.4.4 Other Wasm Trusted Execution Environments.

Se-Lambda. Se-Lambda [53] was developed in 2018 and uses Intel SGX to provide the two-way sandboxed environment. Se-Lambda provides serverless computing framework and remote authenticity check mechanisms. It was built on top of OpenLambda [26].

AccTEE. AccTEE [23] uses Intel SGX as its TEE. AccTEE runtime is open-source, and it offers a two-way sandboxing environment. AccTEE was developed in 2019, and as an "early" trusted runtime relied on the Node.js JavaScript runtime as a building component. Moreover, to execute JavaScript and Wasm, the developers of AccTEE used Chrome's JavaScript engine V8.

Veracruz. Veracruz [67] is a framework that provides a Wasm trusted runtime. Veracruz is focusing on VM-based TEE technologies for future development[7], such as Arm CCA and AWS Nitro Enclaves. Veracruz also uses WASI, as it provides a system interface for tasks such as accessing Veracruz's in-memory filesystem, generating random bytes, and similar operations. Veracruz presents a framework to develop Wasm applications in a TEE for privacy-preserving application scenarios [14]. Wasmtime is chosen by the developers of Veracruz as the Veracruz Wasm engine.

4.5 Other forms of abstraction layers

4.5.1 Constellation. Constellation [63], developed by Edgeless Systems, is a Kubernetes distribution designed to run on top of confidential computing technologies. It ensures that all data within a Kubernetes cluster remains encrypted at rest, in transit, and during processing, effectively shielding the entire cluster from the underlying infrastructure.

The core of Constellation's security model involves running all Kubernetes nodes, including control plane and worker nodes, inside TEE-protected Virtual Machines using SEV or TDX. To maintain end-to-end encryption, Constellation transparently encrypts all network traffic between pods using WireGuard-based encryption and secures persistent storage, including state disks and cloud storage services. Cryptographic keys are managed within the CVMs, and key management is integrated into the cluster's lifecycle, ensuring that keys are securely generated, stored, and rotated without exposing them to the underlying infrastructure.

Constellation is a CNCF-certified Kubernetes distribution, ensuring compatibility with existing Kubernetes tooling and workflows. By integrating confidential computing into the Kubernetes ecosystem, Constellation enables a way to securely migrate sensitive workloads to the cloud while still leveraging the cloud-native ecosystem, especially around Kubernetes.

[7] https://github.com/veracruz-project/veracruz/issues/330

4.5.2 Occlum. Occlum [62] is a memory-safe, multi-process library OS (an OS where functionalities are provided to the application as libraries, and compiled into an unikernel) designed to run applications within SGX enclaves. It enables unmodified or minimally modified Linux applications to execute securely inside SGX by providing support for compiling unmodified C or Rust code directly to a binary supported by Occlum. The build and deployment process in Occlum is inspired by common container tools, but Occlum does not support OCI images.

Traditional solutions built on top of SGX adopt designs that allocate a separate enclave for each process. On the contrary, Occlum adopts a single-enclave, multi-process architecture. This approach allows multiple processes to share the same enclave, significantly improving performance by reducing startup latency and inter-process communication overhead.

To maintain strong isolation between processes within the shared enclave, Occlum employs Software Fault Isolation (SFI) [70]. SFI is a software instrumentation technique that inserts runtime checks before every potentially dangerous operation, for sandboxing untrusted software modules within a single address space. This ensures that each process operates securely without compromising the integrity of others. More specifically, Occlum's SFI scheme is specifically adapted for SGX, so that multiple processes can live within a same enclave without interfering with each other.

Occlum is implemented in Rust, a memory-safe programming language, which inherently reduces the risk of common memory vulnerabilities inside the solution. The system supports various file systems, including read-only hashed file systems for integrity protection, writable encrypted file systems for confidentiality, and untrusted host file systems for convenient data exchange between Occlum and the host OS.

4.5.3 Oak. Oak [24] is an open-source platform designed to build distributed systems that are externally verifiable and rooted in hardware-based trust through TEEs. Oak is targeted at VM-based TEEs, and currently supports AMD SEV and Intel TDX.

To minimize the size of the TCB, Oak adopts a split architecture that separates applications into two components: an enclave application and a host application. The enclave component is executed within a TEE and handles security-critical operations, while the host component, which operates outside the enclave, serves as a frontend by exposing a gRPC endpoint.

Oak offers two execution environments for enclave applications. The first is the Oak Restricted Kernel, a minimal microkernel designed to run a single application on a single CPU core. This option results in a minimal TCB and provides strong isolation, but it requires careful partitioning of the application to isolate the trusted logic. The second option is the Oak Container, which embeds a full Linux kernel and a complete userspace environment, and supports OCI images.

4.5.4 nitriding. Nitriding [75] is an open-source toolkit designed to abstract the constrained development model imposed by Nitro Enclaves. While NE traditionally requires minimal network access and complex build processes, nitriding simplifies this workflow, allowing developers to run unmodified Linux applications inside enclaves with secure and seamless Internet connectivity. Additionally, nitriding emphasizes transparency and trust by making its entire codebase externally verifiable.

To bridge the enclave's strict I/O constraints, nitriding provides a secure communication proxy that facilitates encrypted data exchange between the enclave and the external Internet via the untrusted host, without exposing the enclave to direct network access. This architecture decouples enclave development from the limitations of Nitro's native toolchain by supporting conventional application packaging, process management, and resource configuration.

Table 2. Overview of abstraction layer technologies. The *Year* column refers to the availability of technology and not its release date.

Technology	Year	Approach	Supported TEEs	Maturity	Open source
SCONE [7]	2016	Container	SGX	Industrial	✗
Gramine [15, 52, 64]	2017	Container	SGX, TDX	Industrial	✓
Oak [24]	2018	Container, Other	SEV, TDX	Research	✓
Open Enclave SDK [60]	2018	SDK	SGX, TZ	Industrial	✓
Se-Lambda [53]	2018	WebAssembly	SGX	Research	✗
Asylo [9]	2018	SDK	SGX	Research	✓
AccTEE [23]	2019	WebAssembly	SGX	Research	✓
Occlum [62]	2020	Other	SGX	Industrial	✓
Teaclave [8]	2020	SDK	SGX, TZ	Industrial	✓
Enarx [22]	2021	WebAssembly	SEV, SGX	Research	✓
libkrun [38]	2021	Container	SEV	Industrial	✓
TWINE [40]	2021	WebAssembly	SGX	Industrial	✓
CoCo [65]	2022	Container	SEV, SGX, TDX, SE	Industrial	✓
Constellation [63]	2022	Other	SEV, TDX	Industrial	✓
Veracruz [67]	2022	WebAssembly	NE, CCA	Research	✓
WaTZ [41]	2022	WebAssembly	TZ	Research	✓
Anjuna Seaglass [4]	2023	Container	NE, SEV, SGX	Industrial	✗
Nitriding [75]	2023	Other	NE	Industrial	✓

4.6 Abstraction layers takeaways

This section described an extensive list of abstraction layers. An overview is available in Table 2. Various approaches are used by these technologies, but three main ones can be identified. The SDK approaches propose a way of writing code that are compatible with multiple TEEs at once, or that are able to compile already existing applications without the need for modifications adapting them to TEEs. The other two main approaches, containers and WebAssembly, leverage well-known packaging and isolation technologies to execute software on various TEEs.

Abstraction layers can have very different features depending on what the abstraction layer goals are. Some of these features, along with discussion on how to implement them and their compatibility with the main approaches identified in this section, will be discussed in the next section.

5 IMPROVEMENTS BROUGHT BY ABSTRACTION LAYERS

In the previous sections, this study established a comprehensive and up-to-date database of abstraction layers, serving as a foundation for this section. Abstraction layers provide improvements to bare TEEs across several domains, which will be described and analyzed below, along with the tradeoffs involved in using an abstraction layer. The analysis of this section uses the underlying TEE as a baseline, focusing exclusively on features that the TEE cannot provide by itself.

Abstraction layers intend to improve the development and deployment of confidential computing applications. These improvements can take place anywhere in the lifecycle of a confidential computing application, and may vary from an abstraction layer to another. Figure 4 presents this lifecycle, that goes from the initial applications contents (binaries or source code, depending on the abstraction layers) to the final deployment of TEE-protected applications in various TEEs. It also highlights where the improvement brought by abstraction layers take place inside this lifecycle. Each improvement listed in Figure 4 (multiple input formats, portability, easy deployment, and improved security) will be

discussed in this section. Abstraction layers support for each of these improvements, along with other criteria, are indicated in Table 3.

This section begins by analyzing how different abstraction layer approaches can efficiently target the main TEE approaches. It then examines the improvements these layers introduce throughout the lifecycle of a TEE-protected application, focusing on its two main phases of its life: the development, and the deployment and use. Finally, this section reviews how abstraction layers strive to preserve the security guarantees of TEEs.

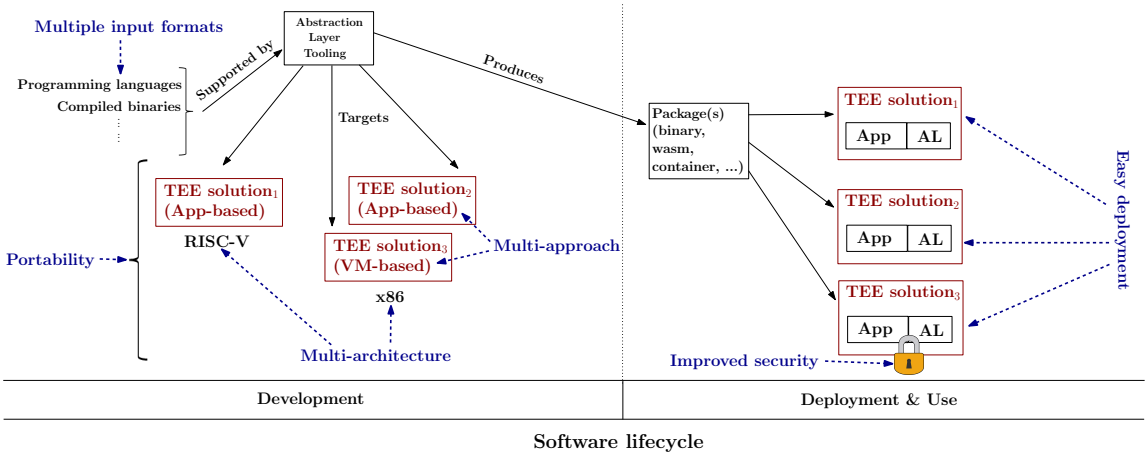

Fig. 4. Representation of the lifecycle of a TEE-protected application leveraging an abstraction layer. Improvements proposed by abstraction layers are indicated using blue dashed arrows.

5.1 Approaches for an abstraction layer design

Implementing an abstraction layer requires working with the restrictions of TEEs and their various approaches. The abstraction layer approach will have an impact on its ability to target various approaches of TEEs. Furthermore, depending on the abstraction layer design, some may use strictly one enclave per application, where others will regroup several applications in one same enclave.

5.1.1 Abstraction layers approaches targeting app-based TEEs. The design of app-based TEEs are often aimed at running only one app per enclave. Therefore, it is difficult for an abstraction layer to leverage these TEEs in order to make them run multiple applications per enclave. This requires the development of a small operating system within the constraints of the TEE to be able to deploy multiple applications on top of it. The challenge of developing even a small secure operating system are well-known, and this OS will be added to the TCB of the solution, making its security even more critical.

SDK-based abstraction layers, by design, only target one app per enclave, making their development on app-based TEEs relatively easy. Furthermore, most app-based TEEs are already providing their own SDK, allowing SDK-based abstraction layers to potentially reuse part or the totality of these SDKs.

Supporting app-based TEEs is way harder for container-based abstraction layers. Indeed, even if the goal of the abstraction layer is to run only one application per enclave, running a container requires a lot of effort. By design, the isolation of containers relies on the Linux kernel, the container needs a filesystem, the ability to run ELF files, etc.

Table 3. List of improvement support for each abstraction layer.

Technology	Approach	Multiple input formats	Portability	Multi-architecture	Multi-approach	Easy deployment	Improved security	Number of supported TEEs
SCONE [7]	Container	●	○	○	○	○	●	1
Gramine [15, 52, 64]	Container	●	●	○	●	○	●	2
Oak [24]	Container, Other	●	●	○	○	○	●	2
Open Enclave SDK [60]	SDK	○	●	●	●	○	○	2
Se-Lambda [53]	WebAssembly	●	○	○	○	○	●	1
Asylo [9]	SDK	◐	○	○	○	○	○	1
AccTEE [23]	WebAssembly	●	○	○	○	○	●	1
Occlum [62]	Other	◐	○	○	○	○	◐	1
Teaclave [8]	SDK	◐	●	○	○	○	◐	2
Enarx [22]	WebAssembly	●	●	○	●	○	●	2
libkrun [38]	Container	●	○	○	○	●	●	1
TWINE [40]	WebAssembly	●	○	○	○	○	●	1
CoCo [65]	Container	●	●	●	●	●	●	4
Constellation [63]	Other	●	●	○	○	●	○	2
Veracruz [67]	WebAssembly	●	●	●	○	○	●	2
WaTZ [41]	WebAssembly	●	○	○	○	○	●	1
Anjuna Seaglass [4]	Container	●	●	●	●	●	●	3
Nitriding [75]	Other	●	○	○	○	●	◐	1

Even with a preprocessing step that adapts the container images before they are being executed, there is still a sizable amount of software that needs to be run into the enclave to support the features of containers.

WebAssembly-based abstraction layers require a WebAssembly runtime that is running inside the enclave. Since WebAssembly is very lightweight, it is used to target small embedded devices. This resulted in the development of small, portable WebAssembly runtimes suitable for being integrated into an app-based TEE. The challenges described above about the execution of multiple binaries in parallel still applies to WebAssembly. However, the simplicity of its binary format and ISA, compared to more complex formats like ELF and x86, makes the development of such an operating system slightly easier.

5.1.2 Abstraction layers approaches targeting VM-based TEEs. While software aimed at VM-based TEEs is often portable due to standardized VM image formats, slight differences in functionalities (support for memory encryption, protection against memory corruption, etc.) exist [25]. Such differences need to be taken into account while designing an abstraction layer that need to be portable across these TEEs.

VM-based TEEs are mainly aimed at running a whole virtual machine. While this approach is well-suited for public cloud ecosystems, where a virtual machine commonly serves as a unit of computing, it may not be optimal for more general use cases. Indeed, including a whole OS in the TCB is not desirable from a security point of view. Abstraction layers may thus aim at reducing the OS size, or even removing the OS altogether and replacing it with a standalone application.

However, if the goal is to deploy a large number of applications protected by TEEs, it may be simpler from a system administrator point of view to include the deployment and orchestration solution inside the TEE directly, especially if said solution has sensitive data to protect on its own. Taking Kubernetes as an example, protecting its secrets and other sensitive content is not trivial and adds another layer of security on top of the TEE management, whereas integrating Kubernetes directly into the TEE removes this constraint. This approach is adopted by Constellation [63].

From a cloud developer point of view, one of the main benefit of an abstraction layer is allowing them to easily, or even transparently, exploit the advantages of a TEE. VM-based TEEs can protect several applications in a single enclave. Furthermore, they can allow integrating a cloud solution (e.g., Kubernetes) within a TEE, enabling developers to transparently use features from the cloud solution while being protected inside a TEE.

Another argument in favor of embedding the orchestration solution inside the TEE is the number of enclaves. When deploying numerous applications with each its own enclave, the performance may be affected. The number of applications may even be capped, as most TEEs have a limit on the number of enclaves that can be run in parallel. By running the orchestration solution and all applications inside the same enclave, it reduces the performance overhead and removes the potential limitation on the number of applications. On the other hand, running all applications in the same enclave prevents from separating the applications from each other, which may be useful in the case of multi-tenant or multi-criticality environments.

This difference in approach stems from the same difference in philosophy that led to the app-based and VM-based TEE approaches distinction; either the emphasis is placed on fine-graining the security to each application, which may incur more costs in terms of design and development of the security solution, or the emphasis is made on the simplicity of use, gathering all applications and a large TCB inside one enclave. Both have advantages and disadvantages. In this context, one also has to take into account not only the developers experience but also the system administrators experience.

5.2 Improvements on development

Abstraction layers can simplify the development process by providing higher-level programming models and APIs that conceal the complexities of low-level TEE interactions. This allows developers to focus more on application logic rather than on TEE specifics, to have faster development cycles and to reduce time-to-market for new applications. They allow a broader range of developers to engage with TEE technology, including those who may not have deep hardware or cybersecurity knowledge.

5.2.1 Multiple input formats. An abstraction layer may propose several possibilities for tooling to develop an application. These possibilities may be proposed alongside or on top of existing solutions coming from the TEEs underneath. Having more options for creating an application using an abstraction layer allows to leverage the skills of a bigger pool of software developers. This also lowers the barrier to accessing the benefits of confidential computing.

Furthermore, as observed in Section 3, most TEE solutions that require an adaptation of the software for their solution provide an SDK that only supports the C programming language. C is a memory-unsafe language that is more and more criticized among the cybersecurity community, especially for the development of new applications. Abstraction layers bring the possibility of using newer, memory-safe languages as an alternative to C/C++, such as Rust or Go. For example, both these languages are able to compile static binaries used in containers. For WebAssembly, developers can use any programming language that compiles to Wasm via LLVM [35] or other compatible toolchains, thus relaxing language constraints for secure execution.

Another possibility brought by abstraction layers is to consume already packaged applications (as compiled binaries, container images, or other formats) and adapt them for confidential computing use. SCONE is an example of such practice, where an off-the-shelf container image can be adapted to run on SGX. This allows to package not only in-house software but also well-known software packaged as container images (such as web servers or databases) to use along with said software. This improves the ease of development along with development times by allowing the leverage of tried and tested software.

Some abstraction layers are even able to run a binary not designed for running inside a TEE without any modification. Indeed, SCONE modifies the libc embedded inside the container to adapt it to SGX. This modification may introduce subtle changes in the behavior of the application that may be difficult to troubleshoot. Furthermore, reproducing such bugs requires providing the developer with access to corresponding hardware, which may not be easily available. Abstraction layers relying on WebAssembly and WASI such as TWINE [40] are able to execute Wasm binaries targeting WASI without any modification. This brings a great testability, allowing the developer to test on its development machine and eliminating the need for additional, costly hardware.

5.2.2 Portability, multi-approach and multi-architecture. An essential feature brought by abstraction layers is portability, i.e., the possibility to support multiple TEE technologies without having to rewrite TEE-specific code. This is relevant to avoid vendor lock-in, to improve the availability of an application, or to distribute it across different environments. Portability can be more or less flexible depending on how the abstraction layer is implemented. The assessment of an abstraction layer's portability can be split into two sub-criteria. The first one, multi-approach, is the ability to support multiple TEE design approaches as presented in Section 3. The second one, multi-architecture, is the possibility of supporting multiple hardware ISAs. Not supporting one of these criteria limits the abstraction layer possibility of extending support to a large range of TEEs.

A multi-approach is relevant for developers aiming at targeting various computing environments with the same application. While VM-based TEE applications may be easily ported from one technology to another, doing so from a VM-based TEE to an app-based TEE or between two app-based TEEs is complex and often requires a rewrite of the application. Abstraction layers remove these constraints and allow using one codebase for all approaches.

Multi-architecture is an important feature for distributing a confidential computing application across various hardware targets. This is especially a challenge for edge applications, which are often required to run on more varied hardware platforms than the ones commonly found in clouds. The multi-architecture constraint breaks the portability of VM-based applications, as most VM-based platforms support only VMs that are using the same ISA as the host. If they can run VMs for other ISAs than the one of the host, it often comes at a huge performance cost because of emulation.

The SDK, container, and WebAssembly-based abstraction layers are all able to support each of the two main TEE approaches, although at different levels of efficiency, as already discussed in Section 5.1. However, designing an abstraction layer that can efficiently support both approaches in a unified manner is more complex. As app-based TEEs are usually more constrained than VM-based TEEs, abstraction layers that aim to support both approaches often adopt designs that are constrained by app-based TEEs, before moving the design to VM-based TEEs.

For a multi-approach abstraction layer, one of the most important part of its design is the definition of interfaces through which the application interacts with its environment. These interfaces must be as portable as possible in order to integrate well with all approaches. SDK-based abstraction layers are offering a lot of flexibility in this regard, as the interfaces provided by the SDK can be tailored by the abstraction layers creators. Containers are less flexible as

they rely on various Linux mechanisms to interact with their environment, mechanisms that can be difficult to port to an abstraction layer. Finally, for WebAssembly, the use of WASI as interface allows to decouple application logic from specific TEE implementations. As long as a TEE (such as Intel SGX) can support or interpret Wasm modules conforming to WASI, applications can be executed securely across different enclave technologies, promoting broader TEE compatibility than containers.

For the multi-architecture support however, containers and SDK fall short compared to the WebAssembly approach. Indeed, if containers can support multiple ISAs, one usually needs to create one container per ISA, multiplying the number of artifacts the developer has to maintain and test. SDK-based abstraction layers can in theory support a large number of ISAs, but in practice this large support is constrained by the design of the abstraction layer and the tooling used to transform the source code into something usable by the TEE. Indeed, providing robust support for a large set of ISAs, even for a single language, is not a simple feat, and abstraction layers not relying on a well-known compiler framework (such as LLVM), or regularly integrating the improvements of such a framework into the abstraction layer tooling, will fall behind WebAssembly on ISA support. Furthermore, SDK-based abstraction layers will still produce different binaries for different ISAs. On the other hand, WebAssembly-based abstraction layers produce one WebAssembly binary for all supported approaches and ISAs.

5.3 Improvements on deployment and use

Abstraction layers can ease the difficulty of deploying TEE-protected applications across multiple, potentially complex, environments. They can also help with improving the security guarantees offered by the TEEs.

5.3.1 Easy deployment. One of the key features of abstraction layers is to ease the amount of work to deploy TEE-protected applications in production. Depending on the use case, production deployments must take into account various criteria such as availability, monitoring, available hardware, and integration with existing solutions. This last point is essential since the integration of TEE-protected applications into an already existing infrastructure should be as smooth as possible. In this regard, SDK-based abstraction layers often rely on the deployment technologies provided by the underlying TEEs, and thus are difficult to integrate with other deployment technologies. On the other hand, both container and WebAssembly technologies are well-integrated with cloud-native deployment technologies. Abstraction layers based on them consequently allow for an easier management of distributed applications, bringing out-of-the-box lifecycle management, monitoring, and other features provided by orchestration platforms such as Kubernetes.

WebAssembly goes even further thanks to its portability and lightness, that gives WebAssembly-based abstraction layers the possibility to be deployed on a large list of constrained devices. This allows system administrators to leverage TEEs present on edge and far-edge devices, which are becoming more and more relevant as the need to process sensitive data near their production points grows. Another point where abstraction layers ease the deployment of TEE solutions is by improving the interoperability between different TEE implementations. This enhances flexibility for system administrators to switch hardware or deploy across different environments without re-architecting their solutions.

However, while abstraction layers can provide numerous advantages when deploying an application in a TEE, they can also introduce drawbacks, such as performance overheads, complexity, layering issues, and dependency on the abstraction layer. Indeed, the abstraction will most of the time lead to additional latency or resource consumption compared to bare-metal execution. Furthermore, the additional layers that provide the abstraction layer capabilities can complicate the system architecture and potentially mask issues. Finally, an application's reliance on a specific

abstraction layer's features may limit future flexibility and adaptability if this abstraction layer does not follow already existing and well-used specifications.

5.3.2 Improved security. While the abstraction layers must take care of preserving the security guarantees offered by the underlying TEEs, they can also augment the security of the application in several ways. First, the applications themselves can be verified for correctness (e.g., absence of undefined behavior) and protected (e.g., addition of security mechanisms) in order to avoid the compromission of the abstraction layer or the environment outside the TEE. For SDK-based abstraction layers, verifications can be made at compile time, but the code is often not isolated during execution.

In container- and WebAssembly-based approaches however, the underlying technologies are providing by design an isolation between the application and the abstraction layer. This isolation is especially welcome when the abstraction layer implies that several applications run in parallel within a single enclave. WebAssembly goes even further as some protections such as the Control-Flow Integrity (CFI) [1] can protect the application from malicious actors interacting with it.

By isolating the application from the abstraction layer and the underlying TEE, these approaches allow considering a stronger attacker model than the one commonly used for confidential computing and described in Section 2.1.2. Indeed, attackers coming from within the TEE are not considered in the classic confidential computing attacker model, while it can happen in practice (e.g., a TEE-protected vulnerable web service that gets compromised). Sandboxing technologies like WebAssembly or containers enable a new protection of the TEE host from the guest application, using a two-way sandbox. This is particularly interesting for abstraction layers that regroup several applications in one same enclave, as a compromised application will not necessarily result in a compromise of the totality of the enclave contents.

Abstraction layers on top of TEEs can improve security by constraining how applications interact with the TEE mechanisms. By construction, they provide a dedicated interface for TEEs, which helps to simplify and systematize the correct use of their low-level mechanisms. Since these layers are explicitly designed with TEE constraints and threat models in mind, they can enforce consistent and secure usage patterns and reduce the likelihood of developer errors. In addition, they can ensure that all TEE events are consistently handled according to the expected security model. In contrast, applications built directly on top of the TEE may overlook or mishandle some of these events, thereby introducing subtle but critical security weaknesses.

6 THE FUTURE OF ABSTRACTION LAYERS

Surveying the ecosystem of abstraction layers shows a large ecosystem with multiple solutions aiming at solving different problems. However, not all challenges are addressed by abstraction layers yet, and new challenges may arise in the near future.

6.1 Comparing current approaches

The three main approaches identified to build an abstraction layer are the SDK-based, the container-based, and the WebAssembly-based approaches. They all have their advantages and disadvantages. In addition, each abstraction layer is building on top of these approaches but does not necessarily implement all the features or follow the specifications of these base approaches, potentially breaking some advantages brought by them. However, in general, the approaches pros and cons that have an impact on the abstraction layer features can be identified and are summarized in Table 4.

Table 4. Comparison of the abstraction layers main approaches

	SDK-based	Container-based	WebAssembly-based
Multiple input formats	only source code	support multiple, complex apps	high, lack expressiveness
Portability	no features	portable but with constraints	very portable and light
Easy deployment	no features	strong cloud ecosystem support	partial cloud ecosystem support
Improved security	some security mechanisms	strong sandboxing	very strong sandboxing

WebAssembly stands out as the approach providing the best portability and security improvements. Containers are following close behind, but the fact that they are heavier than WebAssembly applications, less portable especially across ISAs, and requires a Linux kernel (or another solution implementing its isolation features) make them less ideal than WebAssembly, especially on edge devices. However, containers approaches remain the best choice for packaging complex sets of applications, since WebAssembly still lacks expressiveness for advanced features. Containers also benefit from a strong cloud ecosystem support, making them the best choice for targeting TEEs on public clouds. Finally, SDK-based approaches are lagging behind, mostly because they use an outdated design and mindset, that was relevant when confidential computing was a niche technology with a small number of applications designed explicitly for specific needs. The confidential computing ecosystem and its usages have grown in such a way that makes SDK-based abstraction layers mainly irrelevant for applications that are not explicitly targeting a TEE.

6.2 Security challenges

Even if abstraction layers added on top of TEEs can enhance security guarantees of TEEs, the effectiveness of the abstraction layers in preserving security can vary widely depending on the specific TEE technology and implementation and various other factors, including the design and the implementation of the abstraction layer itself, how it interacts with the underlying TEE, and potential vulnerabilities introduced by this additional layer. If the abstraction layer is correctly implemented, it can significantly contribute to the security of the application, offering both reinforcement of the security guarantees offered by TEEs, and additional security features for the application.

6.2.1 Preservation of security guarantees. One of the primary security guarantees of TEEs is the isolation of secure code and data from the host operating system and other applications. When an abstraction layer is introduced, it must maintain this isolation. If the abstraction layer inadvertently exposes sensitive data or allows improper access to the secure environment, this could compromise security. Proper access controls must be integrated within the abstraction layer to ensure that only authorized applications can communicate with or invoke secure operations within the TEE. If these controls are weak or improperly implemented, they can frame a potential breach of security guarantees.

Abstraction layers can potentially increase the attack surface by introducing additional interfaces and pathways for interaction. Each additional layer can provide new opportunities for exploitation or new vulnerabilities if not designed and implemented carefully. As such, the robustness of the abstraction layer's design is critical for maintaining security. If the abstraction layer facilitates the transfer of sensitive data to and from the TEE, it must employ strong cryptographic measures to protect this data-in-transit. Weak data handling practices can expose sensitive information. For example, if the layer fails to verify the origin of requests or inputs intended for the TEE, it could lead to unauthorized access or manipulation of secure operations. A well-designed abstraction layer that incorporates layered security principles, such as segmentation, strong authentication, and regular security assessments, can help to preserve the security guarantees of the TEE. The security of the abstraction layer can also be supported by standardized practices and thorough auditing.

Layers that are widely used, peer-reviewed, and subjected to rigorous security assessments are more likely to maintain the guarantees of the underlying TEE.

By design, an abstraction layer will increase the TCB. The abstraction layer code that is being executed in the enclave needs to be trusted in addition to the TEE code that is executed in the enclave. While there is no guarantee that the abstraction layer and TEE code will be without any vulnerabilities, measures can be taken to minimize the risk, such as security audits. Another stronger possibility is to formally verify the code, as this has already been researched by the ACE study on CoVE [50]. This verification work can be extended to any software of firmware that takes part of a TEE TCB.

6.2.2 Leveraging TEE features. Different TEE implementations can support different sets of features (security or otherwise). These features can even be outside the scope of confidential computing, bringing additional protections against other attacks (e.g., supporting memory encryption to protect against hardware attacks). For an abstraction layer, supporting multiple TEE solutions may lessen the security offered by the TEEs. Indeed, when covering several solutions with different features it is difficult for the abstraction layer to cover them all while keeping one unified way of packaging the application. The abstraction layer can aim at covering all the features proposed by each supported TEE, which will result in an additional burden for the developer and complexity in the design of the abstraction layer. It may also require the abstraction layer to provide a way to abstract the support of features, meaning that the abstraction layer will have to provide a fallback in case a TEE does not support the feature used by the application. Another approach is to cover only the intersection of the features, but then the abstraction layer will fail to accurately exploit all the protection offered by each supported TEE.

Furthermore, providing ways to abstract the TEE features may come with a security risk. Indeed, the application may want to know if a specific security feature is available or not and act in consequence. This goes against the idea of an abstraction layer providing all features transparently and without alerts, yet failing to provide the same level of security or efficiency. Indeed, for features such as memory encryption, the hardware-backed implementation provided by a TEE will often be more secure than the software-only implementation proposed by an abstraction layer as a failover. This security risk is even more present when the abstraction layers are used in conjunction with automatic orchestration or deployment solutions, because then the TEE-protected application may be redeployed on another TEE at any moment, without warning and without requiring an explicit approval from the system administrator.

In order to protect against the security risks posed by such discrepancies, abstraction layers should provide a way to list the characteristics of the TEE the application is currently running on, along with a way to be warned about a TEE change if the abstraction layer supports such a feature. Furthermore, for system administrators it is important to list the ways the TEE-protected application may be redeployed, on another TEE or not. If using automatic deployment or orchestration systems, it is important to check the parameters of the mechanisms that may trigger a redeployment, ensuring that such an action can only happen on the same TEE, or cannot happen without an approval or a warning to the system administrator. These requirements could be fulfilled by an application manifest, where the application can express which features are required and which ones are desirable, so the abstraction layer and the potential orchestrating mechanisms are able to avoid executing the application on a TEE where the required conditions are not met. The contents of this manifest could come from either a manual human source (the developers or the system administrators) or automatically generated from the application.

6.2.3 Ensuring the continuity of data protection. Confidential computing aims to protect data in-use, but doesn't guarantee the protection of this data when it is stored on disk or transported through a network. It is still up to the

developer to ensure that the modification of the data state and its displacement are done securely. This is especially problematic for applications that are not designed for running inside a TEE, and therefore are potentially not handling file or network encryption. By allowing to execute generic applications in TEEs, abstraction layers become responsible for the continuity of the data protection, when it is being stored at-rest or transferred over the network.

An abstraction layer can help during the data transformation by proposing an integrated way to either store or transport data securely that has been produced by the application running inside an enclave. Furthermore, it can propose a transparent data encryption mechanism so that the developers don't have to think about data security at all. A transparent data encryption mechanism also makes it possible to leverage the protection of a TEE in an application that was not conceived for a TEE in the first place, by porting it as is into the TEE using the abstraction layer, and let the transparent encryption seamlessly protect the data. Ideally, to allow generic applications to benefit from the protection of TEEs while ensuring the complete protection of the manipulated data, an abstraction layer should be able to handle the continuity of both data at-rest and data in-transit transparently.

6.3 Possible improvements for a unified confidential computing support

The most notable contributions of abstraction layers to confidential computing as of today are to the execution environments. Therefore, this study is purposely scoped to the execution environment only and ignores other aspects of confidential computing. However, in the future, abstraction layers may be able to abstract not only the TEE solution but other aspects of confidential computing.

6.3.1 Unified, abstracted attestation. One important aspect of confidential computing is attestation. Attestation allows providing a proof to the TEE beneficiary that its application is indeed being executed in a TEE. While the principle of attestation is the same across various TEE implementations, they differ enough in practice to make them incompatible between each other. Indeed, use of different cryptographic schemes, reliance on an external party or not, and discontinuous hardware support are various factors leading to a fragmented attestation ecosystem.

Recent efforts aim to create and standardize a unified attestation protocol for TEEs. The standardization of the Remote ATtestation procedureS (RATS) architecture [11] or Weinhold et al. [73] work on integrating remote attestation into TLS and comparing it with other existing protocols are examples of the interest in developing an attestation protocol that answers the needs of various TEE designs and user needs. However, as of today, most TEEs implement their own attestation solution with little to no support for these new unified protocols. This creates a gap that could be filled by attestation abstraction layers that could provide a unified attestation solution by uniformizing the existing solutions and merging them into a global, unified protocol.

6.3.2 Improved underlying technologies for abstraction layers. In the technologies used in the main approaches, improvements are also possible. Improving the base technologies may result in an improvement on abstraction layers based on said technology, and open the way to new abstraction layers implementations leveraging these new features. Notably, the WebAssembly ecosystem is in a huge transformation, with a major release of the WebAssembly spec that occurred in September 2025, and constant improvements being made in the WASI specifications, resulting in WebAssembly binaries gaining more and more capabilities. This specification and standardization effort is followed by important work on the compiler and developing toolchains, making compiling complex code to WebAssembly increasingly easier. In this context, we may see appear abstraction layers that are supporting WebAssembly and WASI specifications and allow to transparently use a WebAssembly binary inside a TEE, while some WebAssembly-based abstraction layers require recompilation of the code with their custom compiler as they do not entirely support these standards.

The evolution of the container ecosystem is slower but still happening. The main improvements are taking place in the cloud ecosystem, especially in the domains of deploying, orchestrating, and monitoring containers. In this context, there is a possibility of developing abstraction layers that seamlessly integrate TEE-backed applications alongside regular ones, allowing application developers and cloud engineers to exploit the advantages of confidential computing while retaining their already existing cloud environments. New ways of expressing the need for applications targeting confidential computing, such as a manifest expressing the needs of said applications, may be integrated into the existing container standards. This standardization effort would allow the container and cloud ecosystem to fully embrace confidential computing for regular workloads, reaching a new height in data protection and privacy standards.

Finally, SDK-based abstraction layers are constantly being improved by the programming languages ecosystems they rely on. Languages and compilation toolchains improvements result in more secure and more performant binaries being produced. New languages with features relevant for the confidential computing ecosystem can also emerge and be adopted by existing or new SDK-based abstraction layers.

6.3.3 Alternative methods of protecting data-in-use. Confidential computing is competing with other technologies such as Fully Homomorphic Encryption (FHE) for protecting data-in-use. While FHE is vastly different from confidential computing, in some cases applications may benefit from both technologies at once, or a solution may require to support both confidential computing and FHE. In this context, abstraction layers that support both solutions may appear, leveraging one or the other when appropriate or porting a subset of what is possible in confidential computing into FHE-supported solutions.

6.4 New computation environments

The concept of confidential computing started with generic-purpose computing platforms. However, as the use of confidential computing spreads across vastly different ecosystems, the technologies need to adapt. Far edge devices have very little compute power or memory, and some of them are not even fit to run a WebAssembly runtime. TEE solutions are emerging in very small devices and microcontrollers, with a design quite different from the ones listed in this study. On the other end of the spectrum, some applications require huge amounts of computing servers, with the improvements in the High Performance Computing (HPC) ecosystem and the related design changes bringing new challenges. In these kinds of scenarios, each loss of computing power is extremely impactful. However, as the supercomputers are very costly and complex to build, they are most of the time accessible remotely and shared between users. In this situation, users may require the protection of their data, therefore turning towards confidential computing. Abstraction layers in the future may be able to tackle both ends of this spectrum by allowing the leverage of TEEs on either extremely constrained or extremely powerful systems, while ensuring that the least amount possible of computing power is lost.

Furthermore, more and more applications with specific needs require doing computations on peripherals other than the main CPU of the machine. For example, AI applications often leverage Neural Processing Units (NPUs) or Graphics Processing Units (GPUs) to speed up their computations. As the protection of a classical TEE only encompasses the main CPU, the data computed in these external peripherals is at risk of being compromised. While some efforts have already been made on extending TEEs to GPUs [71], the ecosystem still lacks TEEs for NPUs and other accelerators, and the ones existing for GPUs are also fragmented and incompatible, resulting in the same problem as the current TEE ecosystem. An abstraction layer able to leverage a specific family of accelerators TEEs, bringing the advantages of an abstraction layer to a completely new family of applications, may be a game changer in the near future.

Another family of specialized hardware peripherals that TEE applications may need to leverage in order to do some computations are Trusted Platform Modules (TPMs) and Hardware Security Modules (HSMs). These devices are able to securely store cryptographic keys and perform cryptographic computations with them securely, without the keys leaving the hardware. TPMs and HSMs may be considered as very restricted TEEs that are only able to do very specific operations. Abstraction layers being able to securely handle the communication between various generic TEE solutions and TPMs or HSMs would allow the secure handling of custom cryptographic keys alongside the protection offered by a TEE, improving the security of the handled data.

Some TEEs can provide additional protection for device communication, allowing the enclave to exchange data with an external hardware device securely. Such features are available in e.g. SEV-TIO [3] or the IO-PMP extension of RISC-V [44]. If the use of a hardware peripheral is requested by the user application, an abstraction layer should be able to detect if a TEE is able to use additional hardware peripheral protection, and if yes, correctly use this feature while keeping its use transparent for the user application. If a TEE is not compatible with such a feature, depending on the situation, the abstraction layer should be able to either use the peripheral without protection, with or without warning to the user application, or block its use if the TEE integrity is at risk. In the case of external devices, e.g., a GPU, an abstraction layer may simulate in software the presence of a GPU if a secure communication channel cannot be established, or if the currently used TEE does not support such a feature, but it will come with tremendous performance costs that will negatively impact the application.

Finally, quantum systems are a different computing environment, but they still face the same needs that led to the creation of confidential computing. Indeed, quantum computing is very complex and not likely to be generally available in the near future. Using a quantum computer most often requires the user to rent one through public offerings, making its data at risk. Quantum Blind Computing [13, 66] is a form of quantum computing that ensures the privacy of the computation. From this point of view, Quantum Blind Computing is close to Confidential Computing. There are several ways of achieving Quantum Blind Computing, meaning that in the future, abstraction layers for quantum blind computing may appear, bringing some of the advantages described in this study.

7 CONCLUSION

With more and more TEE solutions using various approaches and proposing different features, abstraction layers aim to unify the ecosystem, allowing application developers and system administrators to leverage confidential computing as broadly and efficiently as possible. This systematization of knowledge first listed the main TEE technologies available and described them succinctly. Emphasis has been made on summarizing this quite complex TEE ecosystem, notably because of a large amount of proprietary technologies that are not or scarcely documented. This ecosystem has also been classified in several categories depending on their main design choices. Then, abstraction layers have been listed, categorized on the main technology they rely on, and their inner workings and features are described. Finally, this study gives an extensive analysis of the ups and downs of the existing abstraction layers and on the approaches used for their implementation. One main result is that WebAssembly seems a promising approach for designing abstraction layers that are portable, secure, and support a large set of software. Future improvement possibilities have been discussed based on an analysis of the ecosystem evolution, aiming at understanding how future abstraction layers will evolve, and addressing the gaps identified in this study.

Acknowledgements: This research received funding from the Smart Networks and Services Joint Undertaking (SNS JU) under the European Union's Horizon Europe programme: ELASTIC (GA 101139067).

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.* 13, 1, Article 4 (Nov. 2009), 40 pages. https://doi.org/10.1145/1609956.1609960

[2] Amazon Web Services. 2020. AWS Nitro Enclaves. https://aws.amazon.com/ec2/nitro/nitro-enclaves/

[3] AMD. 2023. *AMD SEV-TIO: Trusted I/O for Secure Encrypted Virtualization.* Technical Report. Advanced Micro Devices, Inc. https://www.amd.com/content/dam/amd/en/documents/developer/sev-tio-whitepaper.pdf

[4] Anjuna. 2018. Seaglass. https://www.anjuna.io/product/seaglass

[5] Arm. 2002. Arm TrustZone. https://www.arm.com/technologies/trustzone-for-cortex-a

[6] Arm. 2021. Arm Confidential Compute Architecture. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture

[7] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 689–703. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[8] The Apache Software Foundation (ASF). 2020. Teaclave. https://teaclave.apache.org/

[9] Asylo. 2018. https://asylo.dev

[10] Jonatan Baumgartner, Christophe Lillo, and Sébastien Rumley. 2023. Performance Losses with Virtualization: Comparing Bare Metal to VMs and Containers. In *High Performance Computing*, Amanda Bienz, Michèle Weiland, Marc Baboulin, and Carola Kruse (Eds.). Springer Nature Switzerland, Cham, 107–120.

[11] H. Birkholz, D. Thaler, M. Richardson, N. Smith, and W. Pan. 2023. RFC 9334: Remote ATtestation procedureS (RATS) Architecture.

[12] C. Bornträger, J. D. Bradbury, R. Bündgen, F. Busaba, L. C. Heller, and V. Mihajlovski. 2020. Secure your cloud workloads with IBM Secure Execution for Linux on IBM z15 and LinuxONE III. *IBM Journal of Research and Development* 64, 5/6 (2020), 2:1–2:11. https://doi.org/10.1147/JRD.2020.3008109

[13] Anne Broadbent, Joseph Fitzsimons, and Elham Kashefi. 2009. Universal Blind Quantum Computation. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*. 517–526. https://doi.org/10.1109/FOCS.2009.36

[14] Mathias Brossard, Guilhem Bryant, Basma El Gaabouri, Xinxin Fan, Alexandre Ferreira, Edmund Grimley Evans, Christopher Haster, Evan Johnson, Derek Miller, Fan Mo, et al. 2023. Private delegated computations using strong isolation. *IEEE Transactions on Emerging Topics in Computing* 12, 1 (2023), 386–398.

[15] Chia che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 645–658. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai

[16] Confidential Computing Consortium. 2022. *A technical analysis of Confidential Computing v1.3.* Technical Report. https://confidentialcomputing.io/wp-content/uploads/sites/10/2023/03/CCC-A-Technical-Analysis-of-Confidential-Computing-v1.3_unlocked.pdf

[17] Yunjie Deng, Chenxu Wang, Shunchang Yu, Shiqing Liu, Zhenyu Ning, Kevin Leach, Jin Li, Shoumeng Yan, Zhengyu He, Jiannong Cao, and Fengwei Zhang. 2022. StrongBox: A GPU TEE on Arm Endpoints. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 769–783. https://doi.org/10.1145/3548606.3560627

[18] M. E. S. Elrabaa, M. Al-Asli, and M. Abu-Amara. 2021. Secure Computing Enclaves Using FPGAs. *IEEE Transactions on Dependable and Secure Computing* 18, 2 (2021), 593–604. https://doi.org/10.1109/TDSC.2019.2933214

[19] Espressif. 2025. ESP-TEE: User Guide. https://docs.espressif.com/projects/esp-idf/en/latest/esp32c6/security/tee/tee.html

[20] Dengguo Feng, Yu Qin, Wei Feng, Wei Li, Ketong Shang, and Hongzhan Ma. 2024. Survey of research on confidential computing. *IET Communications* 18 (04 2024), n/a–n/a. https://doi.org/10.1049/cmu2.12759

[21] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2021. Scalable Memory Protection in the PENGLAI Enclave. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 275–294. https://www.usenix.org/conference/osdi21/presentation/feng

[22] Confidential Computing Consortium from the Linux Foundation. 2019. Enarx, Confidential Computing with WebAssembly. https://enarx.dev Accessed: 2025-04-22.

[23] David Goltzsche, Manuel Nieke, Thomas Knauth, and Rüdiger Kapitza. 2019. Acctee: A webassembly-based two-way sandbox for trusted resource accounting. In *Proceedings of the 20th International Middleware Conference*. 123–135.

[24] Google. 2018. The Oak project. https://github.com/project-oak/oak

[25] Roberto Guanciale, Nicolae Paladi, and Arash Vahidi. 2022. SoK: Confidential Quartet - Comparison of Platforms for Virtualization-Based Confidential Computing. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. 109–120. https://doi.org/10.1109/SEED55351.2022.00017

[26] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Serverless computation with OpenLambda. In *8th USENIX workshop on hot topics in cloud computing (HotCloud 16)*.

[27] Felicitas Hetzelt and Robert Buhren. 2017. Security Analysis of Encrypted Virtual Machines. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Xi'an, China) *(VEE '17)*. Association for Computing Machinery, New York, NY, USA, 129–142. https://doi.org/10.1145/3050748.3050763

[28] J. Hughes. 2004. IEEE standards for encrypted storage. *Computer* 37, 11 (2004), 110–112. https://doi.org/10.1109/MC.2004.210

[29] 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. 2021. Confidential computing for OpenPOWER. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 294–310. https://doi.org/10.1145/3447786.3456243

[30] Intel. 2015. Intel Software Guard Extensions. https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html

[31] Intel. 2021. Intel Trust Domain Extensions. https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/trust-domain-extensions.html

[32] Islet. 2023. The Islet project. https://github.com/islet-project/islet

[33] Yuekai Jia, Shuang Liu, Wenhao Wang, Yu Chen, Zhengde Zhai, Shoumeng Yan, and Zhengyu He. 2022. HyperEnclave: An Open and Cross-platform Trusted Execution Environment. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 437–454. https://www.usenix.org/conference/atc22/presentation/jia-yuekai

[34] Dmitrii Kuvaiskii, Dimitrios Stavrakakis, Kailun Qin, Cedric Xing, Pramod Bhatotia, and Mona Vij. 2024. Gramine-TDX: A Lightweight OS Kernel for Confidential VMs. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 4598–4612. https://doi.org/10.1145/3658644.3690323

[35] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.

[36] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. 2020. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*.

[37] Mengyuan Li, Yuheng Yang, Guoxing Chen, Mengjia Yan, and Yinqian Zhang. 2024. SoK: Understanding Design Choices and Pitfalls of Trusted Execution Environments. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security* (Singapore, Singapore) *(ASIA CCS '24)*. Association for Computing Machinery, New York, NY, USA, 1600–1616. https://doi.org/10.1145/3634737.3644993

[38] libkrun. 2020. https://github.com/containers/libkrun

[39] Tao Lu. 2021. A Survey on RISC-V Security: Hardware and Architecture. arXiv:2107.04175 [cs.CR] https://arxiv.org/abs/2107.04175

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

[41] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. 2022. WaTZ: A trusted WebAssembly runtime environment with remote attestation for TrustZone. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1177–1189.

[42] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, Valerio Schiavoni, Giovanni Mazzeo, Arne Hollum, and Darshan Vaydia. 2023. A Comprehensive Trusted Runtime for WebAssembly with Intel SGX. *IEEE Transactions on Dependable and Secure Computing* (2023).

[43] Fan Mo, Zahra Tarkhani, and Hamed Haddadi. 2024. Machine learning with confidential computing: A systematization of knowledge. *ACM computing surveys* 56, 11 (2024), 1–40.

[44] Jien Hau Ng, Chee Hong Ang, and Hwa Chaw Law. 2022. A Realization of IO Physical Memory Protection for RISC-V Systems. In *2022 IEEE 15th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. 375–380. https://doi.org/10.1109/MCSoC57363.2022.00066

[45] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. 2020. A Survey of Published Attacks on Intel SGX. arXiv:2006.13598 [cs.CR] https://arxiv.org/abs/2006.13598

[46] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. 2013. Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base. In *22nd USENIX Security Symposium*. USENIX Association, 479–494.

[47] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. 2017. Sancus 2.0: A Low-Cost Security Architecture for IoT Devices. *ACM Transactions on Privacy and Security (TOPS)* 20, 3 (September 2017), 7:1–7:33.

[48] Open Container Initiative. 2024. OCI Image Format Specification v1.0.1. https://specs.opencontainers.org/image-spec/?v=v1.0.1

[49] Open Container Initiative. 2024. OCI Runtime Specification v1.0.2. https://specs.opencontainers.org/runtime-spec/?v=v1.0.2

[50] Wojciech Ozga. 2023. Towards a Formally Verified Security Monitor for VM-based Confidential Computing. In *Proceedings of the 12th International Workshop on Hardware and Architectural Support for Security and Privacy* (Toronto, Canada) *(HASP '23)*. Association for Computing Machinery, New York, NY, USA, 73–81. https://doi.org/10.1145/3623652.3623668

[51] Wojciech Ozga, Guerney D. H. Hunt, Michael V. Le, Gaeher Lennard, Avraham Shinnar, Elaine R. Palmer, Hani Jamjoom, and Silvio Dragone. 2025. ACE: Confidential Computing for Embedded RISC-V Systems. https://arxiv.org/pdf/2505.12995.

[52] The Gramine project. 2017. https://gramineproject.io

[53] Weizhong Qiang, Zezhao Dong, and Hai Jin. 2018. Se-lambda: Securing privacy-sensitive serverless applications using sgx enclave. In *Security and Privacy in Communication Networks: 14th International Conference, SecureComm 2018, Singapore, Singapore, August 8-10, 2018, Proceedings, Part I*. Springer, 451–470.

[54] Kévin Quénéhervé, William Pensec, Philippe Tanguy, Rachid Dafali, and Vianney Lapôtre. 2024. Exploring Fault Injection Attacks on CVA6 PMP Configuration Flow. In *2024 27th Euromicro Conference on Digital System Design (DSD)*. 43–50. https://doi.org/10.1109/DSD64264.2024.00015

[55] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. https://doi.org/10.17487/RFC8446

[56] Andreas Rossberg. 2025. WebAssembly Specification. https://webassembly.github.io/spec/core/

[57] Ravi Sahita, Vedvyas Shanbhogue, Andrew Bresticker, Atul Khare, Atish Patra, Samuel Ortiz, Dylan Reid, and Rajnesh Kanwal. 2023. CoVE: Towards confidential computing on RISC-V platforms. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*. 315–321.

[58] Samsung. 2021. *Samsung Knox Whitepaper v1.5*. Technical Report. https://image-us.samsung.com/SamsungUS/samsungbusiness/solutions/topics/iot/071421/Knox-Whitepaper-v1.5-20210709.pdf

[59] Muhammad Usama Sardar and Christof Fetzer. 2023. Confidential computing and related technologies: a critical review. *Cybersecurity* 6, 1 (May 2023), 10. https://doi.org/10.1186/s42400-023-00144-1

[60] Open Enclave SDK. 2018. https://openenclave.io/sdk/

[61] Amazon Web Services. 2022. The Security Design of the AWS Nitro System. https://docs.aws.amazon.com/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.html

[62] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 955–970. https://doi.org/10.1145/3373376.3378469

[63] Edgeless Systems. 2022. Constellation. https://www.edgeless.systems/products/constellation

[64] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. 2014. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 9, 14 pages. https://doi.org/10.1145/2592798.2592812

[65] Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Christophe de Dinechin, Pau-Chen Cheng, and Hani Jamjoom. 2024. Crossing Shifted Moats: Replacing Old Bridges with New Tunnels to Confidential Containers. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 1390–1404. https://doi.org/10.1145/3658644.3670352

[66] J van Dam, G Avis, Tz B Propp, F Ferreira da Silva, J A Slater, T E Northup, and S Wehner. 2024. Hardware requirements for trapped-ion-based verifiable blind quantum computing with a measurement-only client. *Quantum Science and Technology* 9, 4 (aug 2024), 045031. https://doi.org/10.1088/2058-9565/ad6eb2

[67] Veracruz. 2020. https://veracruz-project.com

[68] AMD Secure Encrypted Virtualization. 2016. AMD. https://www.amd.com/en/developer/sev.html

[69] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. 2018. Graviton: Trusted Execution Environments on GPUs. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 681–696. https://www.usenix.org/conference/osdi18/presentation/volos

[70] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient software-based fault isolation. *SIGOPS Oper. Syst. Rev.* 27, 5 (Dec. 1993), 203–216. https://doi.org/10.1145/173668.168635

[71] Qifan Wang and David Oswald. 2024. Confidential Computing on Heterogeneous CPU-GPU Systems: Survey and Future Directions. arXiv:2408.11601 [cs.CR] https://arxiv.org/abs/2408.11601

[72] Wenwen Wang. 2022. How Far We've Come – A Characterization Study of Standalone WebAssembly Runtimes. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. 228–241. https://doi.org/10.1109/IISWC55918.2022.00028

[73] Carsten Weinhold, Muhammad Usama Sardar, Ionuţ Mihalcea, Yogesh Deshpande, Hannes Tschofenig, Yaron Sheffer, Thomas Fossati, and Michael Roitzsch. 2025. Separate but together: integrating remote attestation into TLS. In *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '25)*. USENIX Association, USA, Article 77, 8 pages.

[74] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2019. The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security* (Auckland, New Zealand) *(Asia CCS '19)*. Association for Computing Machinery, New York, NY, USA, 73–85. https://doi.org/10.1145/3321705.3329820

[75] Philipp Winter, Ralph Giles, Moritz Schafhuber, and Hamed Haddadi. 2023. Nitriding: A tool kit for building scalable, networked, secure enclaves. arXiv:2206.04123 [cs.CR] https://arxiv.org/abs/2206.04123

[76] Florian Zaruba and Luca Benini. 2019. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 11 (2019), 2629–2640. https://doi.org/10.1109/TVLSI.2019.2926114

[77] Yixuan Zhang, Mugeng Liu, Haoyu Wang, Yun Ma, Gang Huang, and Xuanzhe Liu. 2025. Research on WebAssembly Runtimes: A Survey. *ACM Trans. Softw. Eng. Methodol.* 34, 8, Article 239 (Oct. 2025), 47 pages. https://doi.org/10.1145/3714465

[78] Mark Zhao, Mingyu Gao, and Christos Kozyrakis. 2022. ShEF: shielded enclaves for cloud FPGAs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 1070–1085. https://doi.org/10.1145/3503222.3507733