Karlsruhe Institute of Technology

Confidential Computing in Distributed Systems

Bachelor's Thesis of

Wenzhe Vincent Cui

at the Department of Informatics
SCC – Steinbuch Centre for Computing

Reviewer:	Prof. A
Second reviewer:	Prof. B
Advisor:	M.Sc. C
Second advisor:	M.Sc. D

11. Month 2021 – 02. Month 2022

I declare that I have developed and written the enclosed thesis completely by myself. I have submitted neither parts of nor the complete thesis as an examination elsewhere. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. This also applies to figures, sketches, images and similar depictions, as well as sources from the internet.

PLACE, DATE

. . . . . .

(Wenzhe Vincent Cui)

Todo list

Abstract

Distributed systems, like grid and cloud computing, have become essential for sharing of and accessing to remote resources. These systems are often used for economic reasons or to simplify collaboration and exchange of information. Because the maintenance and management of these distributed systems and the hardware it is running on is very complex and expensive, these tasks are often outsourced to service providers.

However, certain industries may have stricter security and trust requirements and governments too are growing more concerned with the privacy and confidentiality of user data. Confidential computing is an upcoming technology that seeks to address this trust issue by guaranteeing both privacy and integrity of data and code while enabling remote attestation.

DSGVO?

This paper will explore different platforms that facilitate sharing of resources while make use of confidential computing in order to preserve the privacy of client data. It will also investigate limitations imposed by the use of confidential computing and finally showcase challenges of realizing a secure joint distributed system.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Motivation

1.2. Goal

Comparison/Evolution to non-confidential

Remove support development of application

This thesis focuses on two main goals:

I Management of infrastructure and orchestration of application workloads running on this infrastructure. Provide application developers services that support them with the development of applications and the deployment of application workloads.

II Make application workloads deployed by these services confidential (definition in section 2.1.2) and allow verification of this confidentiality by clients and third-parties.

Services mentioned in goal **I** could include

- continuous delivery and deployment (CD)
- deploying applications like services and computation tasks
- orchestration and monitoring of those applications

While these services should support the application owner in developing and deploying applications (**I.a**) it should impose as little limitations as possible on the application owner. Decisions regarding the programming language, libraries, and frameworks should be made by the application owner (**I.b**).

Goal **II** does not imply application security. Application workloads deployed with the help of the service provider should be shielded from the infrastructure, the service provider, and its services. But securing the application is outside the scope of this goal (**II.a**). The confidentiality should also not require manual application modification by the application owner (**II.b**).

2. Terminology

2.1. Terms

2.1.1. Platform

Group of technologies and services that are used as a base upon which applications are developed and deployed.

Expand explanation of term ‘Platform’

Go more into dynamic infrastructure management

2.1.2. Confidentiality

Isolation techniques like containerization and virtualization protect the infrastructure or platform from applications. On the other hand confidentiality is the direct opposite and its goal is to protect the application from the infrastructure or platform it is running on.

2.1.3. Trusted Computing Base (TCB)

The set of all hardware, firmware, and/or software components that are critical to the security of a given computing system. These components are the only components in the computing system that operate at a high level of trust. This does not imply that these components are secure, but that they are crucial to the security of system as a whole.

2.2. Roles

While this section defines roles and their tasks, one should keep in mind that a single entity can take on multiple roles. Section 6.1 will define which roles shouldn’t be aggregated onto one entity.

Remove term “client” from document

Infrastructure Provider

The infrastructure provider controls the hardware and firmware used to provide compute, networking, and storage resources.

Service Provider

An entity providing services on top of infrastructure provided by the infrastructure provider in order to ease the development and deployment of applications – often in the form of a platform.

Application Owner

An entity developing an application for a customer on the platform provided by the service provider.

Data Owner

An entity that is in the possession of possibly sensible data that will be processed or used by an application deployed on the platform provided by the service provider.

Verifier

An entity that verifies that the services provided by the service provider can be trusted.

3. Background

3.1. Virtualization

3.1.1. Virtual Machine Manager (VMM)

3.2. Containerization

For a long time VMs have been the standard virtualization technology used in cloud environments in order to provide virtualized infrastructure with an abstraction level at the operating system level. While containers are also a virtualization technique, VMs and containers address different problems. VMs allow virtual hardware allocation and management, while containers address the issue of interoperable applications. In the cloud environment (see section ??) VMs focus on the infrastructure layer while containers are used in the platform layer [6].

While VMs have been used in order to schedule processes as manageable units they require a file system containing a full operating system. This results in large memory and storage requirements and slow startup times. In contrast, containers share their OS with the host they are running on and only hold application code, runtime, system

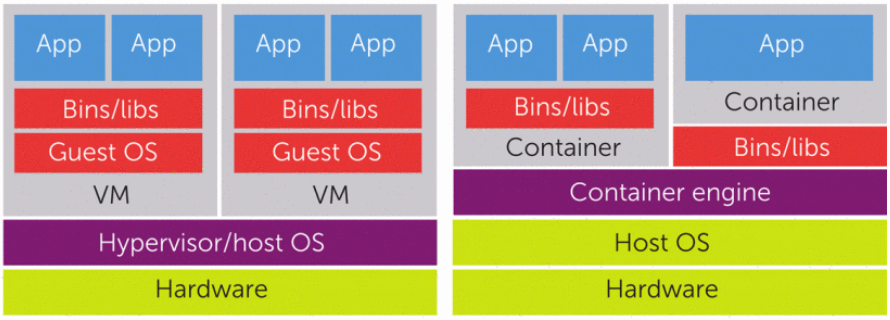


Figure 3.1.: A comparison between VM and container based virtualization architectures Source: “Containerization and the PaaS Cloud” [6]

tools, and system libraries needed for the application (see figure 3.1). This enables interoperability while making containers lightweight.

Today most containerization technologies are based on the Linux Container (LXC) technique. LXC facilitates namespace isolation, which allows groups of processes to be separated. This prevents the (read and write) access to other resources running on the same host and is used to isolate processes, inter-process communication, networking, mount points, and kernel and version identifiers.

A container describes a running instance of a container image. While VMs mount the file root file system in read-write mode (after executing integrity checks), containers mount the root file system as read-only and utilize union mounts in order to add a writable file system on top of the read-only root file system. This allows easier distribution and replacement of images, even for existing containers.

3.2.1. Container Runtime (CR) and Container Manager (CM)

Use OCI specifications to make this section clearer

Explain container image registry

Container runtimes are the pieces of software that manage the lifecycle of containers. Today these runtimes fall mostly fall into two categories, low-level and high-level runtime. While low-level runtimes like runc or kata are only responsible for spawning and running containers on the host, high-level runtimes like containerd or cri-o build on top of low-level runtimes and also manage container images. In order to differentiate and avoid confusion we will refer to high-level runtimes as container managers in this thesis.

4. Technical Research

4.1. Distributed Systems

We will be referring to the definition of distributed systems defined by Steen and S. Tannenbaum [9] which identifies two characteristic features of distributed systems.

Collection of autonomous computing elements
: Distributed systems consist of a number of computing elements which are behaving independently to each other. These computing elements can either be hardware devices or software processes.

Offer a single coherent system
: While consisting of a group of autonomous computing elements, a distributed system should appear to its users as a single system. This implies that the autonomous computing elements have to collaborate in some form.

In order to achieve this single coherent view, distributed systems are often organized to have separate layers of software that operate on top of resources that may offer different ways of interaction and abstract their differences into unified interfaces. These so called "middlewares" are often also responsible for providing inter-resource communication, security and accounting services, and fault tolerance by masking and recovering from failures.

Steen and S. Tannenbaum define four design goals of distributed systems:

Resources sharing
: Distributed systems should make it easier for users and applications to be able to access and share resources in order to enable and support collaboration and exchange of information.

Transparency
: Different types of transparencies are used in order to hide the physical location, differences of interaction, replication, concurrency, and failures of a resource or process inside a distributed system.

Openness

The components that make up a distributed system should be open, that means that these components should be interoperable, composable, and extensible, in order to make the usage and integration with other systems easier.

Scalability

A distributed system should be able to scale in various dimensions. The three main dimensions identified by Neuman [5] are size, geographical, and administrative scalability.

4.1.1. Distributed Computing

Distributed computing systems are inherently distributed systems, which means they share the fundamental characteristics. The goal of a distributed computing system is to facilitate distributed system principles in order to group a set of systems at a geographical distance and providing problem-solving environments for its users.

In general distributed computing systems allow users to share, manage the access to, and use computing resources.

Cloud Computing systems usually have a centralized control, facilitate open and proprietary protocols and interfaces, in order to provide on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured services ([7]).

The NIST Definition of Cloud Computing [7] defines three distinct service models of cloud computing, which can be applied to the general distributed resource management system definition.

Infrastructure as a Service (IaaS)

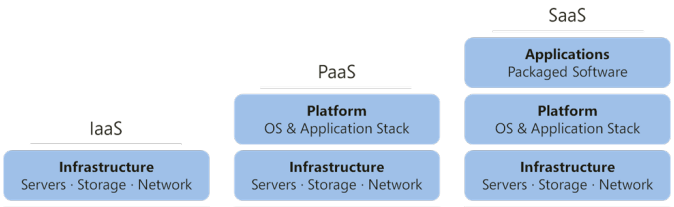
In an IaaS service model the managed resources are fundamental computing resources. This includes – physical or virtual – machines, storage, and networks. The consumer does not manage or control the underlying infrastructure but can deploy and run arbitrary software, including operating systems, onto the provided infrastructure.

Platform as a Service (PaaS)

The PaaS service model adds another layer of abstraction on top of the IaaS service model. Consumers can deploy and run applications on a provided platform that supports a set of programming languages, libraries, services, and tools. So the managed resources in a PaaS service model are applications. The consumer does not manage or control the underlying infrastructure, and additionally has no control over the operating system.

Software as a Service (SaaS)

In this model the managed resources are applications. But in comparison to the PaaS model the consumer is not able to run arbitrary applications, but only a provider specific selection of applications. The customer does not manage or control the underlying infrastructure, operating system, or even the application.



Comparison in a table

4.2. Confidential Computing (CC)

Data can be in three distinct states: “at rest”, “in transit”, and “in use”. These three states describe data that lies in persistent storage, traversing the network, and data that is currently being processed. While technologies in order to protect data “at rest” and “in transit” are commonly used today, there are not many methods to protect data “in use”.

By executing computations in hardware-based trusted execution environments (see section 4.2.1) confidential computing (CC) protects data “in use”. In order to assure a client that the requested environment can be trusted, remote attestation protocols (see section 4.3) are used. The combination of these two practices enable service providers to offer trusted services that can be verified remotely by clients or third-parties.

4.2.1. Trusted Execution Environments (TEEs)

Properties

There are different definitions of a trusted execution environment (TEE) with varying properties. We will focus on the properties needed to build a platform where securing client data and building trust is the main concern. These properties are:

Data confidentiality

Prevent unauthorized entities to view data that is in use within a TEE

Data integrity
: Prevent unauthorized entities to add, remove, or change data while it is in use within a TEE

Code integrity
: Prevent unauthorized entities to add, remove, or change code executing in the TEE

Explain why these three properties are important in order to build trust

Why hardware-based TEE

The security of a software layer can only be as strong as the layers below it. This is why an ideal security solution acts from the lowest layer possible. By providing security through the lowest layer – the hardware – it is possible to remove almost all layers below the TEE from its TCB. This includes the infrastructure provider, host operating system, hypervisor, service provider, and platform the TEE is running on. The only component remaining in the TCB of the application is the hardware providing the TEE properties.

Utilizing software-based TEEs would mean giving control of enforcing TEE properties in the hands of the service provider, conflicting with goal **II.a**. For this reason this paper will focus solely on hardware-based TEE technologies.

4.2.2. TEE Models

There are two distinct models of TEEs, process-based and VM-based.

Process-based TEEs

Process-based TEEs introduce a new programming model. A program needs to be split into two components, trusted and untrusted. These are often referred to as the "enclave" and "host". The enclave is executed in an environment where all TEE properties are provided. The enclave component should contain all code that interacts with sensitive data, whereas the host component is responsible for handling non-sensitive tasks like networking and file I/O.

While the host is not shielded the enclave is protected from the rest of the system, this includes

- the enclave's own host
- other processes

- the operating system
- the bootloader and firmware such as the BIOS
- the hypervisor and host operating system (in virtualized environments)
- hardware other than the processor

Go into detail about how process-based TEEs work?

Splitting a program into enclave and host is challenging. It requires a deep understanding of security and how these process-based TEE solutions work. To ease the development of such applications SDKs and frameworks often hide the split between host and enclave from the developer[8].

Library OSes like Gramine and Occlum go even further and provide a POSIX-like runtime environment with support for network, file I/O, and multithreading. These library OSes contain the whole application in the enclave. But because the enclave does not have access to the OS running the program, the library OS provides libraries which implements OS system calls as library functions. This library then interfaces with a boilerplate host for I/O[10].

Even though these SKDs, frameworks, and library OSes ease the development and make porting of existing applications easier, using process-based TEEs still requires more development effort and modifications of existing applications.

Example: Intel SGX

VM-based TEEs

The main concept of VM-based TEEs is to apply the TEE properties to a full virtual machine running on top of a VMM. This enables VM-based TEEs to run basically any application without modifications. While VM-based TEEs have a larger TCB than process-based TEEs they are explicitly designed to protect workloads from underlying software layers.

Example: AMD SEV

Example: Intel TDX

Comparison

Figure 4.1 shows a simplified comparison between the TCBs of an application running without CC, inside a process-based TEE, and inside a VM-based TEE. The main difference is the size of the TCB. While process-based TEEs should be used for security critical

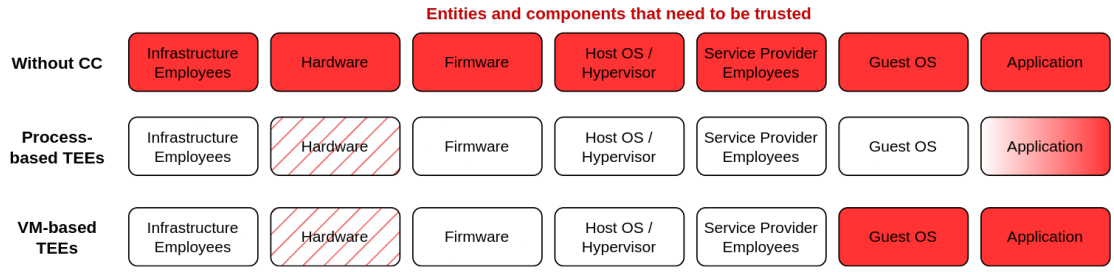


Figure 4.1.: Comparison of TEE models. Boxes marked in red highlight entities and components that have to be trusted. In both TEE models the CPU still enforces the TEE properties and thus still has to be trusted. In the process-based TEE the application is marked with a gradient because the application has to be split into two parts.

applications, the need to modify existing applications in order to run them conflicts with goal **II.b**.

But VM-based TEEs align with goal **II.a** by shielding the virtual machine and thereby applications running inside it from the outside, while also allowing the execution of unmodified applications which does not conflict with goal **II.b**.

4.2.3. Measurements

Often, TEEs also provide the capability of measuring the initial or current state of the TEE which can be used as evidence in a remote attestation system (see chapter 4.3). In order to prohibit the modification of the evidence the measurements are signed by the hardware.

4.2.4. Problems

Performance impact

Find performance comparisons

Support for accelerators

Reference the heavy use of accelerators in ML

Explain ongoing effort without going too much into detail (Reference CRONUS)

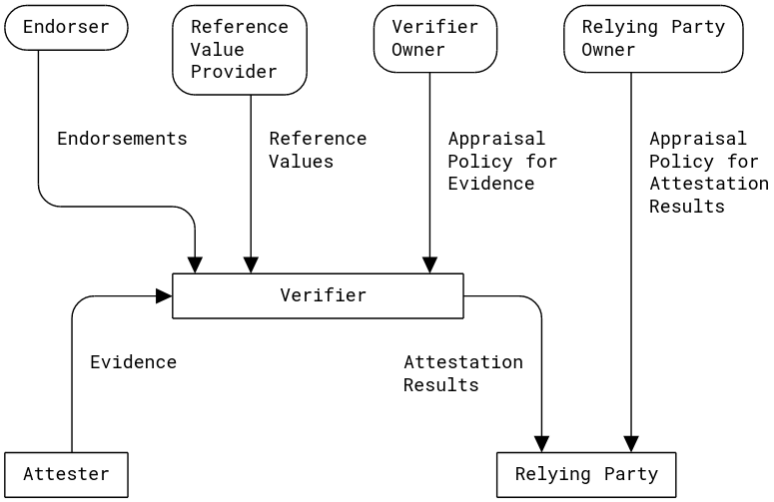


Figure 4.2.: RATS architectural overview.
Source: *Remote ATtestation procedureS (RATS) Architecture* [3]

4.3. Remote Attestation

Running applications inside a CC enabled environment is not enough in order to deepen the trust with clients or third-parties. In order to prove that the application is running inside a CC enabled environment and that neither the application nor the system it is running on has been tampered with, the service provider can facilitate remote attestation.

In a remote attestation environment there are at least two roles: the attester and the relying party. The attester produces information about itself (evidence), on which the relying party makes a decision whether to trust the attester or not. In our context the attester would be the system running a client application and the relying party would be the client accessing or sending data to the application.

4.3.1. Remote Attestation Procedures (RATS)

Remote Attestation Procedures (RATS) defines a general architecture, roles, and messages in order to establish trust between the relying party and the attester. Figure 4.2 shows the general architecture. RATS introduces few new roles where the most prominent one is the verifier. It produces attestation results by using evidence, endorsements, reference values, and applying an appraisal policy to assess the trustworthiness of the

attester. This procedure is called “appraisal of evidence”. These attestation results then support the decision process of the relying party on whether to trust the attester or not.

RATS Architecture - indepth

4.4. Existing Solutions

4.4.1. Commercial Solutions by well-known Service Providers

AWS Nitro

Description: AWS Nitro

- More of a TEE technology
- does not solve trust problem with the service provider

Confidential VMs (Azure, GCP)

Description: Azure Confidential Containers

- AMD SEV enabled VMs or VMs providing Intel SGX capabilities
- Verification has to be implemented by the customer

GCP Confidential Dataproc

Big data processing through fully managed data processing frameworks and tools like Spark and Hadoop. Uses confidential VMs to provide confidentiality.

Dataproc: Verification of nodes?

GCP Confidential Space

Let multiple parties share confidential data with a workload while retaining the confidentiality and ownership of the data.

Confidential Space: More research needed

Even though these solutions all employ CC technology to shield running application workloads from the software layers below (except AWS Nitro), they lack verification of those TEE environments. But this isn’t a problem with the service provider, as putting the verification of their own TEE environments makes the verification pointless. Either

the client does the verification on its own or they employ an external verifier in order to appraise the evidence produced by the TEE (as mentioned in section 4.3.1 we will dive deeper into solving this problem in chapter 6.2).

The more significant problem with these solutions is the lack of raw attestation evidence (produced by the hardware) and/or reviewable virtual firmware used for virtualization in VM-based TEEs. Both are essential in removing the service provider as a trusted entity.

4.4.2. Kubernetes as platform base

Kubernetes is an extensible, open source platform for managing containerized workloads and services. Due to its open source nature and popularity there has been a rapidly growing ecosystem surrounding Kubernetes. It provides general platform features such as deployment, orchestration, scaling, load-balancing, integration with logging, monitoring, and alerting solutions.

A simplified overview over core components inside a Kubernetes cluster (see figure 4.3):

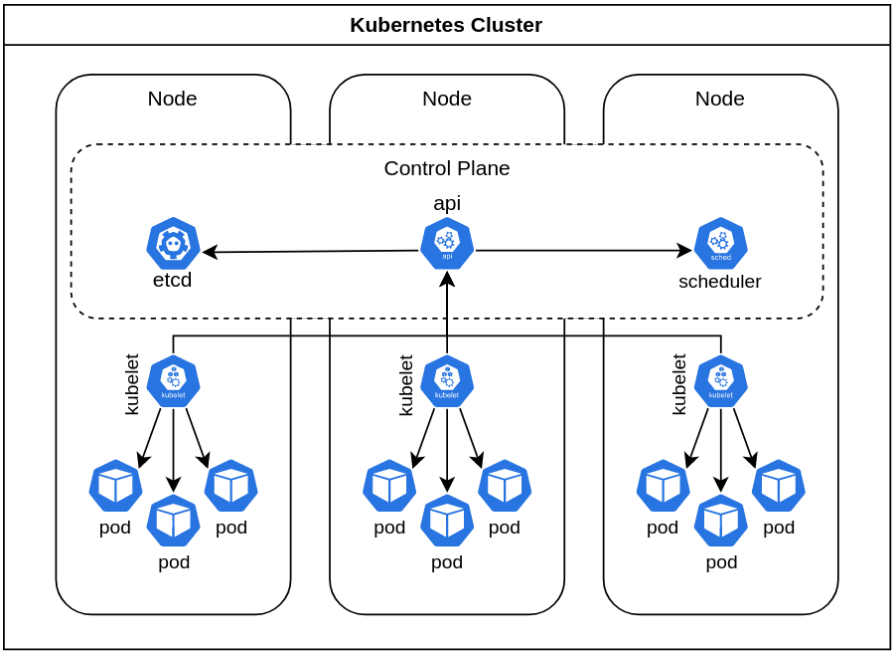


Figure 4.3.: A simplified overview over the Kubernetes components.

Nodes
(Possibly virtual) machines running containerized applications. On each node runs a component called kubelet that manages the pods and the corresponding containers on the node.

Pods A group of containers that share storage and network resources that models an application-specific logical host. The containers that make up a pod are always located on the same node and are scheduled in unison.

Control Plane
A collection of pods that manage the nodes and workload pods inside the cluster. This includes an API, a backing store (etcd) for all cluster data, and a scheduler that assigns pods to nodes.

Container Runtime
Responsible for executing containers on a node (see section 3.2.1).

Kubernetes: Expand as needed in the next chapters

Kubernetes' non-monolithic design allows the replacement of almost every aspect and component of a cluster. This allows a Kubernetes cluster to be run on basically any underlying infrastructure, regardless of the hardware choices or networking design.

There are three levels in a Kubernetes cluster on which confidentiality can be applied to: The node, pod, or container. There are distinct benefits and problems for applying confidentiality on each layer. We will discuss them in the following sections.

Confidential Nodes

The most outer layer where confidentiality can be applied to in the Kubernetes architecture are the nodes. By using confidential computing enabled virtual machines – for example by facilitating AMD SEV or Intel TDX – a Kubernetes cluster operator is able to shield workloads running inside the cluster. Prominent service providers like Azure and GCP already offer the option of deploying their managed Kubernetes clusters with confidential worker nodes. However, these solutions do not include verification of the nodes which means that one would have to build a custom remote attestation system – for example by implementing the RATS (section 4.3.1).

The biggest problem with confidential nodes is the restriction of cluster admin privileges and verifying these as a client. While the workloads are shielded from the infrastructure cluster administrators would still have full control over the workloads running inside the cluster which breaks goal **II.a**.

Confidential Containers

The other extreme would be to apply confidentiality to containers. A process running inside a TEE would only receive confidential data like decryption keys or personalized information after verifying that the process is running inside a TEE via remote attestation. Since the TEE shields the process from the cluster this approach remove the cluster administrator from the TCB of the application.

Even though this approach does not conflict with the goals of this paper, it does collide with the design of Kubernetes. In the Kubernetes architecture the smallest deployable unit of computation is a pod, a collection of application containers. These containers share storage and network resources, but it becomes very hard to share these resources when the containers are shielded from each other. The next approach addresses this architectural issue.

Confidential Pods

Instead of applying confidentiality to containers we can also shield pods from the outside world. This improves the last approach as per definition a pod in a Kubernetes cluster defines an application-specific logical host. As opposed to shielding a single container, shielding a pod would allow sharing storage and networking resources between containers composing the pod.

5. Traditional Distributed Computing Model

5.1. Generalized Distributed Resource Management Architecture

Traditional distributed computing platforms can be split into two layers, the control plane and the data plane.

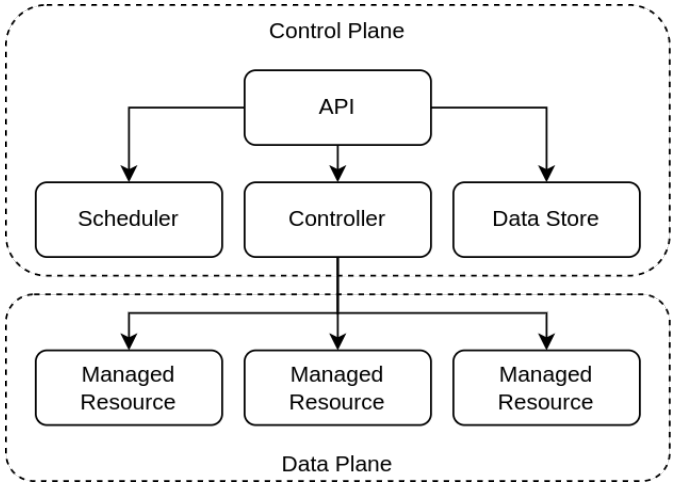


Figure 5.1.: Overview over general components in a Dynamic Distributed Resource Management System

The control plane is a set of components that make global decisions about the distributed system as a whole and acts as a single source of truth regarding the configuration and state of all components and resources in the distributed system.

The control plane generally consists of the following components:

Data Store
: Responsible for consistently storing the current configuration and state of all components and resources.

API An interface for users, administrators, and other components in the distributed system to interact with.

Controller(s)
A component or set of components responsible for managing all or a specific type of computing resources.

Scheduler
Component responsible with determining the physical location of a newly requested computing resource.

The data plane aggregates the computing resources managed by the control plane, for instance virtual or physical machines, storage, networks, or more abstract resources like services, tasks, or secrets.

Customers usually interact with the system by using for example graphical or terminal applications. There are no additional services that are expected to be managed by the customer.

5.2. Thread Vectors

Infrastructure

- Memory (in-scope)
- Physical (out-of-scope)

Platform

- Arbitrary code execution (in-scope)
- Code modification (in-scope)
- Supply-chain (in-scope)

Software threads out-of-scope

5.2.1. IaaS Service Model

In the IaaS Service Model the infrastructure provider would offer the capability of allocating infrastructure to the application owner and the application owner would be responsible for deploying and orchestrating application workloads onto said infrastructure.

Even though the application owner manages and controls the software stack running on the executed infrastructure, the infrastructure provider is still able to read and manipulate the main memory of the provided machines.

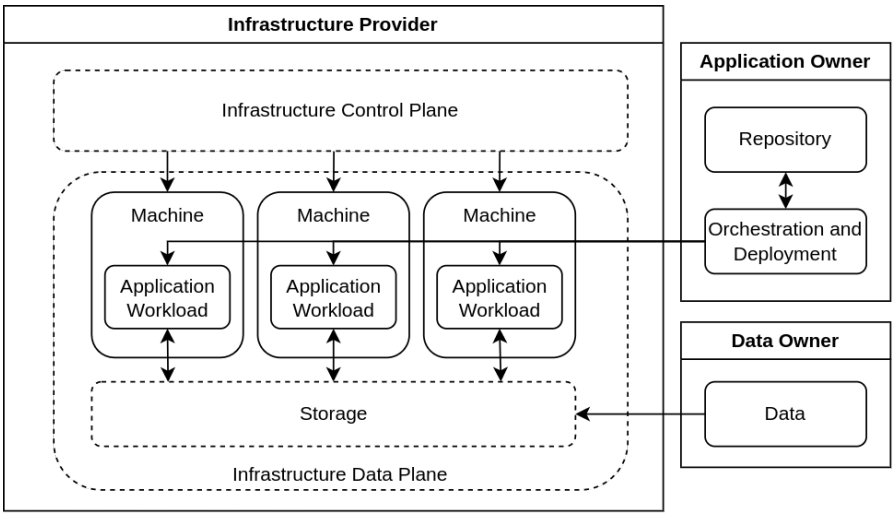


Figure 5.2.: IaaS service model architecture overview.

5.2.2. PaaS Service Model

As outlined in the confidential computing section techniques for shielding data at rest and in transit are commonly used. For this reason we will simplify the architecture view by leaving out the storage and network infrastructure. Applying the generalized distributed resource management architecture to a PaaS service model, we now get the following architecture:

The illustrated control plane in figure 5.3 is a combination of the infrastructure and platform control plane. While the infrastructure control plane allocates infrastructure, the platform control plane deploys and orchestrates application workloads on said infrastructure. In order to do so the platform control plane create execution environments for application workloads and executes application code provided by the application owner through a repository. This implies that the platform control plane is able to execute arbitrary code inside the execution environments it created. And because the service provider manages the control plane, the data owner has to trust the service provider to run only unmodified application code on his/her data.

The data owner could try to use encryption techniques in order to shield the data from the service provider and only provide the decryption key to running application workloads after verifying their integrity, but because the service provider is able to execute arbitrary code in the provided execution environments, it would still be to modify the running workload after the decryption key has been supplied.

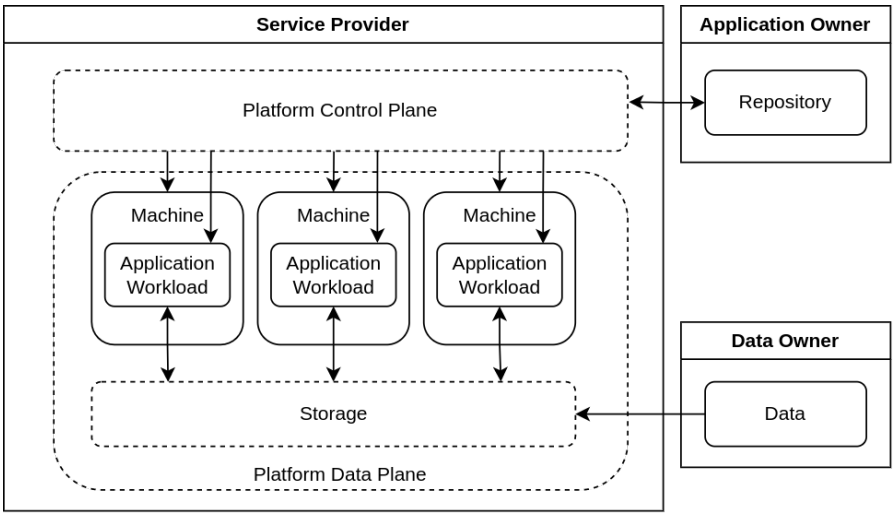


Figure 5.3.: Distributed computing platform architecture overview.

5.2.3. SaaS Service Model

Service Provider would become application owner, defeats purpose

6. Secure Distributed Computing Architecture Proposal

6.1. Trust Model

In the following chapter we will often refer to entities and components as “trusted” and “untrusted”. This trust relation is defined from the perspective of the data owner as this role is in possession of the sensitive data.

6.1.1. Roles

Infrastructure and Service Provider

The infrastructure provider is responsible for the availability of the infrastructure used to provide compute, networking, and storage resources. As such the infrastructure controls and has access to the physical hardware and firmware.

On the other hand the service provider is responsible for managing services that utilize the infrastructure provided by the infrastructure provider in order to ease the development and deployment of workloads. As such the service provider is not only responsible for deploying the applications, but also to provide services that allow the retrieval of attestation evidence and initialize TEE environments.

Application Owner

The application owner designs and implements application workloads that will be deployed and orchestrated by the service provider. This role needs to prove to the customer aspects of compliance to the defined requirements. It also defines computing resource requirements for the application workloads in order to run and maintain compliance.

Data Owner

As the owner of the confidential data, this role is concerned with the visibility and integrity of their data and the compliance of the application with the requested requirements.

Verifier

In order to remove the service provider from the application's trust model, an external party has to verify the service provider's confidentiality claims. The verifier provides an attestation service that appraises evidence and returns attestation results.

While the application owner and verifier have to be trusted by the data owner, the goal is to be able to outsource the management infrastructure and orchestration of applications, while not having to trust the infrastructure or service provider. This implies that the entities that take on the role of infrastructure or service provider should not also be assigned the role of application owner or verifier.

Roles that have the same trust status can be aggregated into the same entity. For example the infrastructure provider can at the same time be the service provider and the roles application owner, data owner, and verifier can be incorporated into the same entity.

6.2. Architecture

6.2.1. Overview

For convenience, from now on we will refer to confidential data as "secrets". Figure 6.1 shows an overview over the components used in this architecture proposal, aiding the reader with understanding the relation between the components introduced in the next section.

6.2.2. Components

Control Planes

In a dynamically provisioned platform managing the infrastructure and orchestrating applications is the responsibility of the control plane. We are grossly simplifying the control plane here by considering it as a single component. In traditional platforms the control plane is a high-privileged component in order to perform management and orchestration tasks.

We want to apply the least privileges principle to the control plane and only allow privileges necessary for basic management and orchestration. This includes managing the lifecycle of applications on a node, migration of applications between nodes, and monitor computing resource usage. But privileges that have the potential to compromise the integrity of application code and data should only be assigned to trusted entities

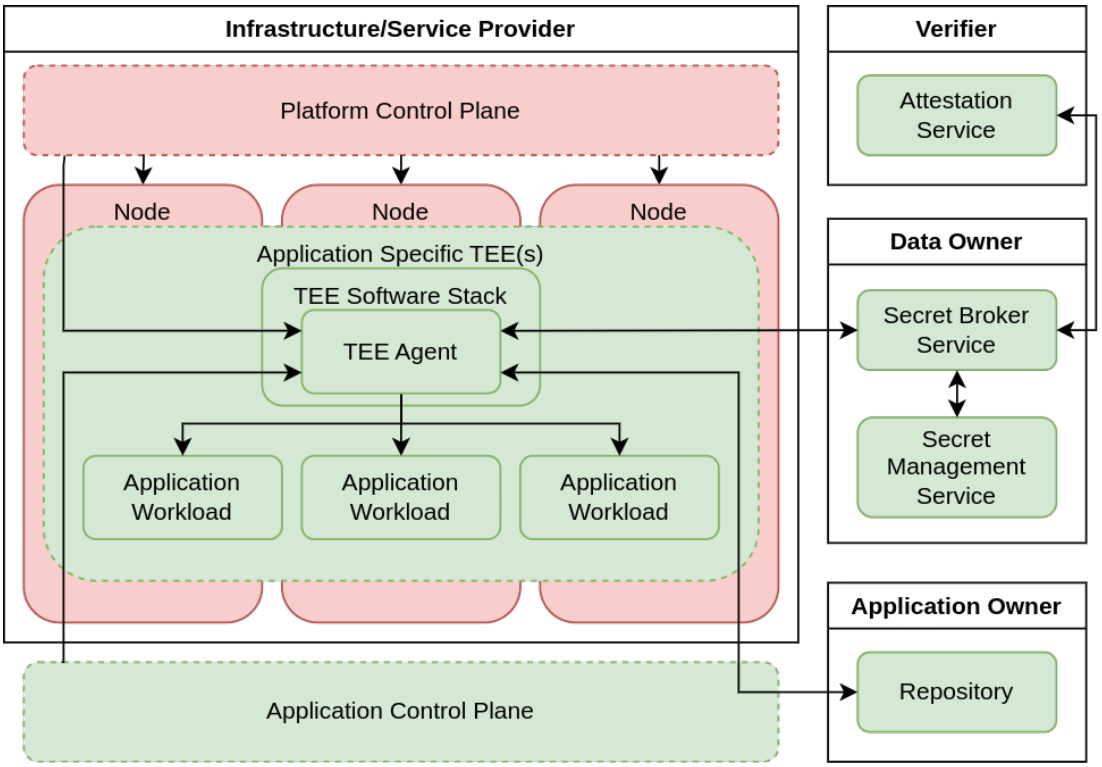


Figure 6.1.: An overview over components used in this architecture proposal. Defines which components are managed by which role and whether it is trusted. Entities and Components marked in red are untrusted, while those marked in green are trusted.

and components. This implies that entities and components controlled by the service provider should not have these privileges.

To achieve this the traditional control plane has to be split into a platform and an application control plane. As stated above the platform control plane manages and orchestrates the platform with only the set of permissions that is needed to achieve this. On the other hand the application control plane is responsible for everything that directly interacts with the application. To make the split of the control plane effective the service provider can not control or have access to the application control plane. This implies that an external entity like the application or data owner has to manage this control plane instance.

TEE Agent

TEEs are designed to shield everything within the environment from the outside. Because the control plane is not part of a specific application TEE, the control plane is not able to interact with applications inside a TEE. To allow and control interactions with between the control plane and applications a new component is needed. The TEE agent itself is executed inside a TEE – not necessarily the same TEE as the application – and is responsible for managing the application specific workloads. It controls the access into the application TEE in order to perform management and orchestration actions and is responsible for enforcing the split control planes' permissions.

In a more traditional platform each node inside a platform cluster runs an agent which manages the applications running on the particular node. But because the node is under the control of the infrastructure or service provider the node and everything running on it can not be trusted. This is why an application specific agent is needed.

The TEE agent is the trusted component that is responsible for the whole attestation workflow. It is responsible for pulling application code, verifying the integrity of the code, executing application code inside a TEE, and injecting secrets into the application TEE. As such the enclave agent itself has to be verified before sending secrets to it. In section 6.3 we will go more into detail how this verification of the TEE agent and application code is performed.

Repository

In order to mitigate supply chain attacks the application owner has to store application code in a trusted signed repository or sign the application code itself. This signature then has to be validated by the TEE agent before executing the code on confidential data.

Secret Management Service

The secret management service is a component that stores confidential data and controls the access this data. The management of this service has to be done by the data owner or delegated to a trusted entity.

Secret Broker Service

The secrets broker service receives secrets requests from the TEE agent with evidence about its own integrity. This evidence is then forwarded to the attestation service which returns attestation results on which the secrets broker service then decides whether to

trust the TEE agent. If the TEE agent is trusted the secret broker service then relays the secrets from the secret management service to the TEE agent.

Attestation Service

In the RATS architecture the attestation service takes on the role of the verifier. It appraises evidence from the TEE agent and returns attestation results to the secret broker service. See section 4.3.1 for more information about how the appraisal of evidence is performed.

6.3. Attestation Workflow

Following is the workflow when the deployment of an application is requested:

1. The platform control plane spawns a new TEE on an arbitrary node.
2. After initialization of the TEE the whole TEE software stack is measured.
3. The measurements are then sent as evidence to the secret broker service.
4. The secret broker service lets the attestation appraise the evidence and then decides whether to trust the TEE agent.
5. If the agent is trusted the secret broker service queries needed secrets from the secret management service and sends them to the TEE agent. These secrets have to include repository or application code signatures.
6. After receiving the secrets the TEE downloads application code from the repository and verifies it with the signatures it received.
7. The TEE agent then starts the application workloads in a TEE. This could be the same TEE where the agent is running or a new workload specific TEE.

6.4. Comparison to traditional architecture

7. Conclusion

…

Bibliography

[1] *A Technical Analysis of Confidential Computing*. URL: https://confidentialcomputing.io/ccc-a-technical-analysis-of-confidential-computing-v1-3_updated_november_2022/.

[2] Raad Bahmani et al. "{CURE}: A Security Architecture with {CUstomizable} and Resilient Enclaves". In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 1073–1090.

[3] Henk Birkholz et al. *Remote ATtestation procedureS (RATS) Architecture*. RFC 9334. Jan. 2023. DOI: 10.17487/RFC9334. URL: https://www.rfc-editor.org/info/rfc9334.

[4] Jianyu Jiang et al. "CRONUS: Fault-isolated, Secure and High-performance Heterogeneous Computing for Trusted Execution Environment". In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2022, pp. 124–143. DOI: 10.1109/MICRO56248.2022.00019.

[5] C Neuman. *Scale in Distributed Systems. In Readings in Dist. Comp. Syst.* 1994.

[6] Claus Pahl. "Containerization and the PaaS Cloud". In: *IEEE Cloud Computing* 2.3 (2015), pp. 24–31. DOI: 10.1109/MCC.2015.51.

[7] Tim Grance Peter Mell. *The NIST Definition of Cloud Computing*. URL: https://doi.org/10.6028/NIST.SP.800-145.

[8] Felix Schuster. *Confidential Computing - How to process data securely on third-party infrastructure*. URL: https://content.edgeless.systems/hubfs/Confidential%20Computing%20Whitepaper.pdf.

[9] Maarten van Steen and Andrew S. Tannenbaum. *Distributed Systems*. 3rd ed. 2017.

[10] Chia-Che Tsai et al. "Cooperation and security isolation of library OSes for multi-process applications". In: *Proceedings of the Ninth European Conference on Computer Systems*. 2014, pp. 1–14.

A. Appendix

A.1. First Appendix Section

Figure A.1.: A figure

…