D2.2
Secure Computation Infrastructure and Self-Management of VM Security

Project number:	643964
Project acronym:	**SUPERCLOUD**
Project title:	User-centric management of security and dependability in clouds of clouds
Project Start Date:	1st February, 2015
Duration:	36 months
Programme:	H2020-ICT-2014-1
Deliverable Type:	Demonstrator
Reference Number:	ICT-643964-D2.2 / 1.0
Work Package:	WP 2
Due Date:	Oct 2016 - M21
Actual Submission Date:	2nd November, 2016
Responsible Organisation:	TUDA
Editor:	Markus Miettinen
Dissemination Level:	PU
Revision:	1.0
Abstract:	This report describes the prototype implementations of technical components of the compute layer of the SUPERCLOUD architecture. The prototypes are concerned with realizing secure computation environments across several cloud service providers while assuring a high level of isolation of computation from access by the service providers utilizing hardware-based security mechanisms. Ways to use hardware mechanisms like FPGA for accelerating computations are also discussed. In addition, two use case-related prototypes demonstrate how SUPERCLOUD services can be instantiated in practice, whereas prototypes related to security policy modelling and enforcement demonstrate the security self-management capabilities of the SUPERCLOUD architecture.
Keywords:	compute layer, isolation, FPGA, security management

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 643964.

This work was supported (in part) by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number 15.0091.

Editor

Markus Miettinen (TUDA)

Contributors (ordered according to beneficiary numbers)

Mario Münzer, Felix Stornig (TEC)
Marc Lacoste, Alex Palesandro, Denis Bourge, Charles Henrotte, Houssem Kanzari,
Ruan He (ORANGE)
Marko Vucolic, Jagath Weerasinghe (IBM)
Sabir Idrees, Reda Yaich, Nora Cuppens, Frédéric Cuppens (IMT)
Markus Miettinen, Ferdinand Brasser, Raad Bahmani, Tommaso Frassetto, David Gens (TUDA)
Daniel Pletea, Peter van Liesdonk (PEN)

Disclaimer

Executive Summary

This document describes technical solutions realized as prototype implementations of components of the computational sublayer of the SUPERCLOUD architecture. These solutions concern issues related to the orchestration of computational environments across several different cloud service providers as well as the strict isolation of computations from access by the service providers with the help of hardware security features. Also solutions for providing access to hardware-accelerated computations utilizing FPGAs through the SUPERCLOUD API are discussed. Solutions related to the self-management of security settings are demonstrated through prototypes related to security policy modelling and enforcement. In addition, two concrete use case-related prototypes are presented. One addresses the issue of enforcing geo-location requirements of computation (e.g., assurance that computations are performed within given jurisdictional boundaries), whereas the other prototype demonstrates the realization of network function virtualization in SUPERCLOUD, which is necessary for the implementation of the logical communication infrastucture within User Clouds.

Contents

List of Figures

List of Tables

Chapter 1 Overview

This deliverable describes the prototypical implementation of the distributed cloud infrastructure for computation and of the SUPERCLOUD mechanisms for self-management of security of VMs running on this infrastructure. The deliverable also includes a description of a number of components to manage trust in the SUPERCLOUD and underlying infrastructure, relying on hardware-enabled security mechanisms. The implementation prototypes demonstrate three core aspects of the SUPERCLOUD architecture:

- Virtualization and orchestration of computation environments;
- Self-management of VM security;
- Trust management based on hardware security mechanisms.

1.1 Virtualization and orchestration of computation environments

This area covers challenges related to computing virtualization in a multi-cloud setting, on top of which may be built security and availability services such as geolocation-aware data isolation and replication.

1.1.1 Virtualization and orchestration

Horizontal orchestration deals with the orchestration of computation environments in the SUPERCLOUD architecture across multiple providers. Specific orchestration challenges and solutions are explored in the horizontal orchestration prototype, described in detail in Sect. 3.1. In addition, the horizontal orchestration of the SUPERCLOUD computation architecture is studied in the context of an example related to network function virtualization (NFV), with a prototype described in Sect. 4.2.

1.1.2 Geolocation-restricted data replication

Another prototype studied geo-location-aware data isolation, as described in Sect. 4.1. Nowadays connected health systems are able to deliver, via collaborations, enhanced and more efficient care for the patients. In such systems healthcare moves outside hospitals, closer to the analysis tools that extract more and smarter knowledge from this data. These types of analyses make the healthcare tailored to a patient and therefore more accurate.
The empowering/enablement of the healthcare tools and ultimately of the patients requires the data to be replicated in multiple data storages across the world (on different platforms) due to requirements regarding data query performance, load balancing and disaster recovery. Replication can be straightforward when the data may be stored in any storage from any geographical location (e.g. country or continent). Unfortunately, this is not the case in the real world, where the data storage platforms need to adhere to different geolocation directives, regulations or requirements imposed by governmental entities and to other requirements defined by customers or end-users.
We focus on regional restrictions, where data is not free to flow from one region or country to another. Such restrictions might be dictated by the countries policies, from collaboration agreements or even

more fine-grained by the owners of the data (e.g. patients). This situation is further complicated by the increasing reliance on cloud-based virtual machines. Geolocation requirements are present in such connected health platforms, where multi-tenant multi-datacenter data management (depicted in Figure 1.1) needs to enforce geolocation requirements like:

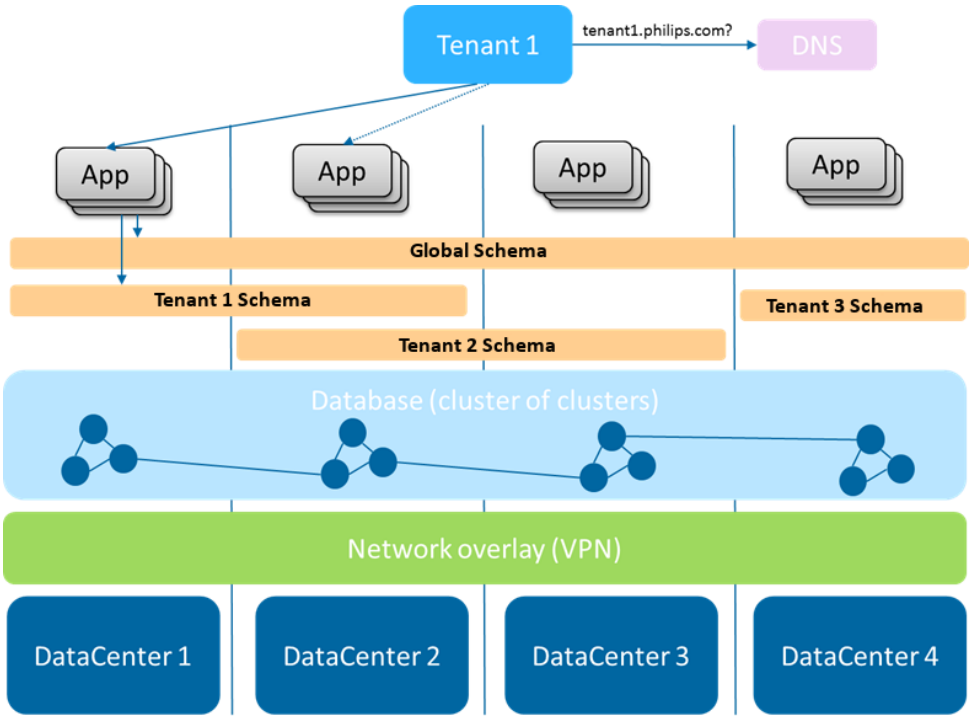

Figure 1.1: Multi-tenant multi-datacenter data management system

- Parts of the data should be managed globally (e.g. master data, service configurations, authentication sources);
- Parts of the data should remain "local":
  - Local could be: on premise or within geographical region
  - Partially-defined by tenant: German customer? Data stays in Germany
- Isolation of data per tenant (multi-tenancy):
  - Keeping all data in one database vs schemas/databases per tenant

Next to these geolocation requirements, we encountered geolocation requirements also in one of the other SUPERCLOUD use cases, namely the Maxdata use case :

- DR8 Location-awareness: Users should be aware of the physical location of Execution Environments (EEs) that process users' data;
- DR9 Location-control: Users should be able to define the set of possible physical locations, at country level, where users' data may be processed.

Storing data only in allowed geographical places can be easy to enforce when the connected health platform is storing and managing the data within a single data storage provider (e.g. a single cloud provider). In such a case, the cloud provider has control over all the virtual machines (VMs) used for deployment of the aforementioned platform and knows where all the VMs and their attached data storages are placed.

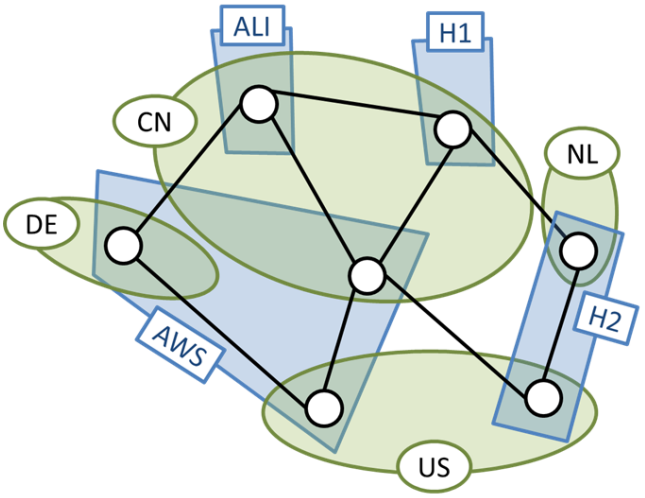

Figure 1.2: Connected health platform. Circles depict servers within the platform, ovals geographical boundaries, and polygons organizations (i.e., cloud providers or hospitals) in control of the servers

This scenario becomes a lot more complex when the platform spans a multitude of cloud providers over multiple locations, and even several on premise hospital systems are part of the aforementioned connected health platform (e.g. Philips HSDP platform). Such a scenario is depicted in Figure 1.2. In this case, multiple VMs belonging to different computing providers are connected; therefore, a centralized solution for replication becomes more difficult to design because of the synchronization across computing providers. The configuration of the deployed VMs is dynamic in such a connected health system, with VMs being deployed in or removed from different computing platforms due to different requirements (e.g. availability, backup procedures, or new services). In this case, a discovery solution, which is able to adapt to the dynamicity of the system, needs to be in place.
One straightforward solution is to encrypt the data and broadcast it all over network with the encryption being done so that only allowed VMs can decrypt it. Such a solution is very inefficient when the size of the data, which needs to be replicated, is big, which is often the case in the connected health systems. Another is to have a central broker that maintains lists of the resources within the connected health system. However, the platforms are dynamic, making such lists difficult to maintain. It would mean that all the computing providers would have to communicate and synchronize their resources/VMs lists and their geographical locations, which ultimately would have also to be trusted. Default solutions for the problem of data replication under geolocation restrictions imply a couple of steps for setting up a secure (e.g. VPN) tunnel between the origin of the data and the place where the data needs to be replicated. The first step is the discovery of a candidate resource where the data can be replicated; the second step implies authentication of the discovered resources; and the third step consists of setting up a session key between the origin and the newly discovered resource where the data can be replicated. In most of the cases, the second and the third step are together part of an authenticated key agreement protocol. Such protocols use a public-key infrastructure for authentication or a pre-shared symmetric key. Here geolocation restrictions would be enforced via out-of-band communication about which servers can be trusted with what data. Furthermore, to the best of our knowledge, there are no authenticated key agreement protocols that provide geolocation.
In Sect. 4.1, we present a prototyped solution that can be used for finding VMs that satisfy needed geolocation requirements. In comparison with state of the art solutions, the proposed solution provides efficient, adaptable and decentralized discovery of replication VMs that satisfy specified constraints. Furthermore, the proposed solution is platform independent, allows easy integration across heterogeneous health systems and lowers the needed amount of trust that the resource/VMs providers (e.g. cloud) are not malicious or dishonest.

1.2 Self-management of VM security

In this section we give a high-level overview of authorization building block of Self-Management of Security as described in D1.2 and describe how the self-management architecture is demonstrated in the Security policy prototypes. The prototypes are described in detail in Sections 3.2 and 3.3.
As described in D1.2, a user-centric control and management of security appears to be a necessity more than a desire for multi-cloud customer: when the dynamic and complex aspect of security management is coupled with the scalability and heterogeneity of muti-cloud environments, it becomes very challenging for a human to manage them. For instance, answers to questions such as how outsourced resources can be manipulated and shared while preserving their confidentiality and integrity? become more difficult to answer when we consider the complexity of nowadays cloud infrastructures, particularly in context of multi-clouds or federation of clouds.
Access control and trust management provide the necessary mechanisms to answer such questions. The security specification of VMs can be achieved with the concept of access control and usage control policy specifications. Based on the different requirements and challenges highlighted in the D1.2 and D2.1, we need to select a security policy model which is capable of modeling various types of security requirements (access control, usage control, and information flow, etc.), provide a way to enforce security requirements dynamically, and most importantly the security policy model should be capable of handling conflicts in security policies.
In this section, we will present our choice of selecting OrBAC model as access control and usage control model which is adequate for handling most of these requirements. Organization Based Access Control (OrBAC) is becoming largely used for modelling access control and usage control policies. It integrates various concepts defined in the previous work such as role, hierarchy, and context. OrBAC also adds extensions to enhance its use in a collaborative system. The main concept of OrBAC is the entity organization. The policy specification is completely parameterized by the organization. This notion encourages researchers to handle simultaneously several security policies associated with different organizations. It is characterized by a high level of abstraction. Instead of modeling the policy by using the concrete and implementation-related concepts of subject, action and object, the OrBAC model suggests reasoning with the roles that subjects, actions or objects are assigned to in the organization. Thus, a subject is abstracted into a role which is a set of subjects to which the same security rules apply. Similarly, an activity and a view are respectively a set of actions and objects to which the same security rules apply.
The OrBAC model introduces two security levels (concrete and abstract). OrBAC defines the concept of context. It is a condition that must be satisfied to activate a security rule. A mixed policy can be offered in OrBAC which defines four types of access: permission, prohibition, obligation and recommendation. Rules conflicts can appear in this policy. This problem may be resolved by affecting a coefficient to each rule. Several types of contexts can be used as temporal, geographical (physical and logical), pre-request, declared, etc. Also, we may have contexts which depend on the application. The hierarchy notion which facilitates the tasks of the administrator is also used in OrBAC. In the same way as RBAC, two types of hierarchy (specialization / generalization and organizational) are defined. Moreover, this hierarchy can be used between different roles, different views, different activities or different contexts. The OrBAC model defines four predicates:

- empower : empower(s, r) means that subject s is empowered in role r.
- consider : consider(b, a) means that action b implements the activity a.
- use: use(o, v) means that object o is used in view v.
- hold: hold(s, b, o, c) means that context c is true between subject s, action b and object o.

Access control rules are specified in OrBAC by quintuples that have the following form:

- permission(org, role, activity, view, context) $\wedge$ empower(org, subject, role) $\wedge$ consider(org, action, activity) $\wedge$ use(org, object, view) $\wedge$ hold(org, subject, action, object, context) $\longrightarrow$ Is_Permitted (subject, action, object).

- prohibition(org, role, activity, view, context) $\wedge$ empower(org, subject, role) $\wedge$ consider(org, action, activity) $\wedge$ use(org, object, view) $\wedge$ hold(org, subject, action, object, context) $\longrightarrow$ Is_Prohibited (subject, action, object).

The obligations differ from permissions/prohibitions by controlling the behavior of the system based on specific events that may occur. For the definition of obligations, we consider two different contexts: the obligation's activation context and the obligation's violation context.

- *Obligation Activation:* An obligation has an activation event after which it becomes effective. This event may be a temporal or an action-based event .

- *Obligation Violation:* An obligation has a violation event which specifies when it is violated. This event may be an action-based, temporal or a relative temporal deadline .

From its specification until its fulfillment, an obligation can have different status, apart from *Obligation Activation* and *Obligation Violation*:

- *Obligation Deactivation:* An active obligation is deactivated when its context ceases to hold. The deactivation of an obligation depends on the type of the obligation. Some obligations may remain required forever after their activation until they are fullfilled.

- *Obligation Fulfillment:* An obligation is fullfilled when its action is taken. Thus, the obligation ceases to be required.

Thus, the specification of the obligations can be expressed as follows:

- obligation(org, role, activity, view, Activation_Context, Violation_Context) $\wedge$ empower(org, subject, role) $\wedge$ consider(org, action, activity) $\wedge$ use(org, object, view) $\wedge$ hold(org, subject, action, object, Activation_context) $\wedge$ $\neg$hold(org, subject, action, object, Violation_context) $\longrightarrow$ Is_Obliged (subject, action, object).

which specifies that the decision (i.e. permission, prohibition or obligation) is applied to a given role when requesting to perform a given activity on a given view in a given context. We call these organizational security rules. Security rules can be hierarchically structured so that they are inherited in the organization, role, activity and view hierarchies. Since a security policy can be inconsistent because of conflicting security rules (for example a permission can be in conflict with a prohibition). The OrBAC model is capable of managing the conflicts at Abstract and Concrete levels. In the scope of current implementation framework we are only considering abstract level conflicts: If no abstract conflicts are detected then no concrete conflicts can exist. However, we have identified concrete conflicts also that are generated due to inference rules.

- **Abstract Conflicts:** An abstract conflict is composed of one abstract permission and one abstract prohibition. An abstract conflict is called a potential conflict because it can create conflicts at the concrete level. For example consider two abstract rules: permission1(org1, role1, activity1, view1, context1) and prohibition2(org2, role2, activity2, view2, context2). Consider a subject subject, an action action and an object object. If subject is empowered simultaneously into role1 and role2, if action is considered simultaneously into activity1 and activity2, if object is used simultaneously into view1 and view2, if context1 and context2 are true for the triple subject, action, object, then the two concrete rules: is permitted(subject, action, object) and is prohibited(subject, action, object) are derived, leading to a concrete conflict. In OrBAC model, these abstract conflicts can be resolved by offering the following possibilities:

  - **Separation Constraints:** When assigning a concrete entity to an abstract one, the separation constraints are verified to prevent the user from violating one and potentially generate concrete conflicts. The separation constraints are not exclusively used to prevent concrete conflicts but can be used to specify separation of duty.
  - **Rule Priorities:** Define a priority order between the two conflicting rules. We associate the authorization rules with priorities in order to evaluate their significance in conflicting situations. Priorities between access control rules may be sometimes derived from the rules syntactical format.

OrBAC has its administration model AdOrBAC. It uses the same logical formalism and the same concepts of ORBAC. As a result, OrBAC is a self-administrated model. In the scope of SUPERCLOUD project, we have extended and adapted the MotOrBAC tool (see section 3.2), which implements OrBAC security policy model.

1.3 Trust management based on hardware security mechanisms

In this section a brief explanation on the topic as well as a brief description of selected approaches will be given. *Intel Software Guard Extensions* (Intel SGX), which has been selected as a means of implementing secure computation within the scope of SUPERCLOUD, will be described in a little more detail. The section should provide an insight about the implications and benefits of Intel SGX compared to other approaches mentioned.

The term *trust management*, introduced for the first time in 1996 by Blaze et al. [28], represents one of the three main principles of trust. The definition of trust given by the *Trusted Computing Group* (TCG)[1] is the following: "an entity can be trusted if it always behaves in the expected manner for the intended purpose". This definition does not imply that the behaviour of the trusted entity is good or honest. It only concerns the variability of the behaviour over time, which will remain the same [22]. Trust management systems are used to evaluate service-providing entities based on the measurement of the trustfulness. The trustfulness in turn is, among others, based on: security (information protection); privacy (preserving of sensitive information); data integrity (includes security, privacy and accuracy); credibility (quality of service); efficiency (in respect of turnaround time); availability (of cloud service provider); reliability (ability to perform functions under specified conditions and duration) and adaptability (availability of data storage as well as redundancy to overcome single point of failure times) [66]. In order to reach these attributes of trustfulness, several techniques, such as hardware-based systems, are used, in which the trust is only rooted in particular hardware. In the following, two hardware-based security mechanisms to obtain trust management will be described:

- TPM-based trust management - the trust is rooted in a dedicated *Trusted Platform Module* (TPM)
- Architecture-integrated trust management - trust management features are integrated into the system architecture itself

Trust management, respectively trust, plays an important part in the context of *trusted execution environment*, shortly TEE, which plays a central role in the above-mentioned hardware-based solutions. A TEE is characterized by its tamper-resistant processing environment and is equipped with memory and storage capabilities. The code is isolated and executed on a separate kernel, so called *separation kernel*, of the CPU that guarantees authenticity, integrity and confidentiality of the code and the persistent memory. Furthermore, a TEE constitutes a closed virtual machine, which is detached from rest of the platform. However, the separation kernel is, as stated before, responsible for the isolated

[1] Industry-driven organization for standardization for the deployment of concepts and open standards for trusted computing platforms

execution of code. The general principle of the separation kernel is based on the simulation of a distributed system in order to divide the system into several partitions, such that an isolation between them can be guaranteed. Besides the isolated execution and persistent secure storage, another goal of a trusted execution environment is *remote attestation*. Attestation provides proof that a user is communicating with a particular software that is running on a particular trusted platform and that the software or hardware meets specific integrity requirements respectively. If the user is a remote party, it is called remote attestation [22][34]. The mentioned proof can be a cryptographic signature computed over a hash of the soft- or hardware's *measurement* [34]. According to [75] measurement "is the process of computing a state indicator of hardware and/or software".

Before we start to discuss Intel Software Guard Extensions, it is good to know and understand the principle of other available products/protocols as well, in order to differentiate between Intel SGX and other hardware-mechanism-based systems available as well as to highlight the capabilities.

1.3.1 TPM-based trust management

This subsection describes trust management technologies that are based on *trusted platform modules* in order to obtain TEE-functionality by means of hypervisor protocol extensions.
The trusted platform module (TPM) is build on a hardware component, respectively a dedicated chip, which is specified and standardized by the Trusted Computing Group. A TPM offers technical features such as the generation of asymmetric keys of length up to 2048 bits (usable for RSA en- and decryption) as well as hash algorithms based on SHA-1 and MD5. Further, each trusted platform module is assigned to a unique build-in *Endorsement Key* (EK) in order to approve the validity of the TPM. The EK is considered as a static key that never leaves the TPM. A trusted platform module provides two further keys: *Storage Root Key (SRK)* and *Attestation Identity Key* (AIK). While the SRK is responsible for the application protection, the AIK signs values, which are stored in the volatile storage of the TPM. Besides the slow performance of the TPM caused by a slow communication bus to the CPU, the main disadvantage of the trusted platform module is that the dedicated chip is connected to a single platform, hence a TPM is ineffectual for cloud computing environments. However, solutions on virtualized TPMs (vTPMs) were proposed, whereby a trustworthy *Trusted Virtual Machine Monitor* (TVMM) hypervisor is used to securely bind the physical TPM to the virtual one. The advantage of this solution is the usage of only one single physical trusted platform module as the *Root-of-Trust* (RoT) for all virtualized TPMs in cloud computing environments.
A trusted platform module may be used as the basis of a trusted execution environment. This could be done by means of a *Static Root of Trust Measurement* (SRTM), which hashes the measurement of all software loaded since the BIOS. However, this is impracticable because of a too large *trusted computing base* (TCB). On the other hand a *Dynamic Root of Trust Measurement* (DRTM) may be used to pause the CPU, start the measuring process from that point on, hash the result and resume the paused processes rather than trust everything since the BIOS.
Nevertheless, there are several well-established TEE approaches known that are based on virtualized TPMs, which are in turn backed by a hardware TPM as a Root-of-Trust, such as: Flicker (2008) [63], TrustVisor (2010) [62] and the eXtensible and Modular Hypervisor Framework (XMHF) (2013) [79], among others.

Flicker

Flicker [63] is an infrastructure for executing security-sensitive code, that makes use of the aforementioned Dynamic Root of Trust (DRTM) principle, in which the operating system is paused in order to execute and verify a piece of sensitive code and resumed afterwards. By this means Flicker isolates security-sensitive code from untrusted parts of the system like the operating system and the BIOS.
In this, Flicker relies on only approximately 250 lines of additional code in the application's trusted computing base.

The attestation of the executed code is done by a TPM. To be more accurate, a cryptographic hash function (SHA-1) provided by the TPM is used to reduce each software event to a hash. Furthermore, each hashed value is stored within the *platform configuration register (PCR)* of the TPM by concatenating it with the previously stored result followed by a signature procedure by TPM's AIK.
Through a so-called *late launch* capability of several processing chips, it is possible to pause the current execution environment, execute a small piece of code, respectively *piece of application logic* (PAL), with the help of *SKINIT* (on AMD's architecture) or *SENTER* (on Intel's architecture) and resume the previously paused execution environment operations. This principle is called *Dynamic Root of Trust Measurement* as already briefly introduced and is used by Flicker to achieve the proposed properties. As executions in Flicker pause the whole system, an application should split long work segments into smaller ones in order to avoid long latency times [63].
The designer of Flicker made a few recommendations on today's hardware architecture to improve the experience with their solution. However, there are some known problems in practice, such as perception of hangs on the machine during Flicker sessions. Furthermore, data loss during transmission on block devices may happen.

TrustVisor / eXtensible and Modular Hypervisor Framework (XMHF)

The design principle of TrustVisor resembles the principle of the previously described solution Flicker: TrustVisor provides an isolated and measured execution environment for PALs without any trust to the application nor the operating system. TrustVisor constitutes a hypervisor for special purposes such as code and data integrity as well as preserving of secrecy of portions of applications. Compared to Flicker, *TrustVisor* [62] occupies more storage of the trusted computing base (approximately 6,000 lines of code) for the verification process, but achieves a higher level of security. Due to its frequent usage of the hardware, respectively the TPM, Flicker is inefficient compared to TrustVisor.
TrustVisor uses an own DRTM-like principle called *TrustVisor Root of Trust Measurement* (TRTM) and employs the trusted platform module as well in it's architecture. In this, TrustVisor's solution is in the usage of virtualized TPMs, so-called *micro-TPMs (μTPM)*, for each PAL, which interact with the TRTM and executes on platform's CPU at high speed and provides just a few necessary possibilities, such as: basic randomness, measurement, attestation and data sealing. The μTPM is associated with each PAL and the physical TPM is responsible for the RoT in TrustVisor itself [62].
On the other hand there exists the protocol named *XMHF (eXtensible and Modular Hypervisor Framework)* [79], which represents a special purpose hypervisor as well and its fundamental principle is similar to TrustVisor. *XMHF* encompasses also about 6,000 additional lines of code and offers modular extensibility and automated verification of hypervisor memory integrity while performing at high speed. On the basis of the modular core design of XMHF, extensibility is provided to so called *hypapps (Hypervisor Applications)*, which are allowed to use custom features and desired properties. In detail XMHF is designed to provide the core functionality of TrustVisor with the ability to support custom hypervisor applications [79].

Other solutions

There are further solutions known such as the work done by Vavala et al. [80], which constitutes an enhanced version of the TrustVisor/XMHF principle. The protocol includes three additional *hypercalls*. A hypercall is the hypervisor equivalent to a system-call. Virtual machines will use hypercalls to request actions or information from the hypervisor [20]. The first hypercall is responsible for the availability of memory to a PAL, the second and third hypercalls are made for retrieving shared keys. While one key is made to secure data sent to known receivers, the second key is made to validate data that were secured by a known sender (PAL). This protocol is not only provided with three additional hypercalls, but rather implements two desirable key features. First, the architecture only loads, identifies and runs modules of the TCB that are actually needed. Second, a robust and verifiable execution chain is responsible for the correct execution sequence of the code modules. The verification of the correctly executed code is done on the client side. The client only has to verify the chain's endpoint

in order to trust the whole chain [80].
Another solution in the domain of TPM-enabled trust management is called *CloudVisor* [84]. The focus of CloudVisor lies on the protection level, which in detail provides protection and guarantees privacy and integrity to the whole level of the hosted virtual machines even if the virtual machine monitor (VMM) is compromised. To achieve these features, CloudVisor is using currently available hardware to achieve nested virtualization (for machine partitioning) and trusted computing (by means of TPMs) [84].

1.3.2 Architecture-integrated trust management

This subsection describes trust management technologies that are integrated into the system architecture and thus do not confine TEE-functionality to a dedicated module, but span it across the whole system architecture.

ARM TrustZone

TrustZone is a technology introduced by *ARM*, which provides hardware mechanisms to separate a *System-on-a-Chip* (SoC) system environment in a non-secure *normal world* and a *secure world*, whereas the latter can be used to hide security critical operations from the normal world. The separation encompasses all hardware and software resources, such as CPU cores, memory and peripherals (e.g. I/O devices). This is achieved by providing two separate virtual CPUs for each physical core and two separate 32-bit memory address ranges, for the normal and secure world respectively. The switch between the two worlds is supervised by a mechanism called the *monitor mode*. When a world switch is done in the monitor mode, the current world state is saved and the state of the world being switched to is restored. The monitor mode always runs in the secure world, and access from the normal world is controlled tightly [23].
A distinct feature of TrustZone is that it only implements the basic hardware mechanisms to partition the system into two worlds alongside with the necessary CPU instructions. The design of the software model is left completely open. This includes the code, which is executed inside monitor mode and the way how access from the normal to the secure world is authorized, the support for single- and multi-threading, process scheduling etc. Likewise there are no predefined methods for authentication, cryptographic primitives, key management, software measurement or software attestation. A public specification of a standardized TrustZone API [43], released by the *GlobalPlatform* association is available. Software that implements the TrustZone API includes, amongst others, *OP-TEE* [15] and *SierraTEE* [16].

Intel SGX

This subsection describes the trust management and its advantages through the use of Intel Software Guard Extensions.

Intel Software Guard Extensions (Intel SGX) comprises a new set of instructions and mechanisms for the Intel architecture that provide integrity and confidentiality to particular regions of code executed on untrusted machines. In this, the security model of Intel SGX assumes the operating system and hypervisor to be potentially compromised [34][49]. Intel SGX works by establishing a protected container, a so-called *enclave* in a protected address space, which cannot be accessed by any non-enclave code [34].
Intel SGX relies on a complex key derivation procedure and key hierarchy in order to derive keys that are needed to perform different tasks. For sake of simplicity neither key derivation procedure nor the particular SGX instructions will be discussed in detail here. Instead the following paragraph gives a high-level overview of the life cycle of an Intel SGX enclave as well as a brief description of the most important functionality [34][49].

1. **Enclave creation:**
   On creation of an enclave the maximum amount of memory, which can be allocated within the enclave, must be specified. Allocation of the protected memory is done in the next step. This practice makes memory usage within the enclave flexible and adjustable to the current needs. The enclave is marked to be in an uninitialized state after creation.

2. **Loading of initial code and data:**
   After the creation of the enclave, the memory can be allocated to it and initial code and data can be loaded into the enclave. It is important to note that the initial code and data comes from unprotected memory and the initial state of the enclave will thus be known to the untrusted OS or hypervisor respectively and therefore should not contain unencrypted secrets. All code and data, which form the initial state of the enclave must be loaded in this stage, as after the subsequent initialization of the enclave, this possibility will be disabled. In this stage the enclave will also update its measurement that is used in software attestation. The software attestation process is going to be described later on. The enclave is still marked as uninitialized at this point.

3. **Enclave initialization:**
   In order to fully initialize the enclave and to be permitted to launch, a special data structure, the *EINIT token*, must be obtained, which has to be issued by a *Launch Enclave* (LE). The LE is a privileged enclave that is provided by Intel itself. It must still be created, loaded and initialized like any other enclave, but the initialization does not require a valid EINIT token. This is because the LE is already cryptographically signed by a Intel key hard-coded into Intel SGX. It is very important to point out that the initialization of any SGX enclave not authored by Intel itself, requires an LE to work. The consequences of this will be discussed towards the end of this subsection. After successfully obtaining and verifying the EINIT token, the enclave is marked as initialized and code within the enclave can be executed. As indicated above, it is not possible anymore to load further code and data into the enclave.

4. **Use and destruction of the enclave:**
   After initialization the enclave is able to authenticate to a remote party using software attestation. This will allow secrets to be disclosed to the enclave over a secure channel. Untrusted applications outside the enclave can now call trusted code, which resides within the enclave. Furthermore, the trusted code has an exclusive ability to directly access data held within the enclave. The untrusted application code receives output from the enclave when the trusted function returns, whereas the enclave data remains in protected memory.
   After the enclave is ready with the execution, it can be destroyed and used memory will be de-allocated.

The identities of Intel SGX enclaves are managed by keeping a so-called *Enclave Signature Structure* (SIGSTRUCT) for each individual enclave. The SIGSTRUCT holds enclave metadata, a measurement of the enclave, a unique *product id* (ISVPRODID), a *security version number* (ISVSVN) as well as an RSA signature over the previous fields, which is computed using the enclave author's private RSA key. The initialization of an enclave as described above can only take place if there is a valid SIGSTRUCT present for the enclave. Intel SGX refuses to initialize the enclave otherwise [49][34].
An enclave author can issue certificates created from the same RSA key to enclaves that are running different versions of the same code. These enclaves would have the same ISVPRODID, but may differ in their ISVSVN. SGX allows for the transfer of secrets between two enclaves, which run different versions of the same software, but only if the source enclave has lower or matching ISVSVN than the target enclave. This helps enclave authors in handling security updates to software running in enclaves. Upon detection of a security issue in an enclave, the author would create an enclave with the fixed version of the code, and migrate the secrets from the vulnerable enclave to the new one [34].

The aforementioned Launch Enclave that has to partake in an enclave's initialization process is also Intel SGX's biggest drawback, as it must be provided by Intel before any enclave deployment can take place. This effectively puts Intel in full control over which people will be able to deploy SGX enclaves and which will not. Enclave authors would therefore be forced to maintain a business relationship with Intel, should Intel plan to effectively make use of this licensing mechanism [34].
As indicated before, Intel SGX utilizes software attestation. During the software attestation process, the contents of an Intel SGX enclave are measured by computing a secure hash over the parameters, which have been used to create the enclave as well as over the initial contents of the enclave. For doing this, a 256-bit SHA-2 hash function is used. The initialization of the enclave also finalizes the measurement hash [34].
Intel SGX supports two different types of software attestation: *local attestation* and *remote attestation*. Local attestation allows an enclave to prove its identity to another enclave on the same CPU by creating an *attestation report* (REPORT), using a single CPU instruction. This REPORT is generated by computing a *Message Authentication Code* (MAC) over the enclave's measurement and other parameters with a symmetric attestation key shared between the two enclaves, which participate in the process. It should be clarified, that the REPORT and the aforementioned SIGSTRUCT are two different data structures, which serve different purposes, although both contain a measurement of the enclave.
In turn, remote attestation allows for attesting to an enclave, which is not on the same platform. The remote attestation process verifies and signs the local attestation report, which is then sent to a remote party. The signature is not implemented in hardware, but in a privileged *Quoting Enclave*, due to its complexity. The Quoting Enclave verifies and signs the secure communication between the attested enclave themselves. Therefore the local attestation mechanism, which is implemented in Intel SGX, is used. It is important to note that the attestation key is not shipped with the chip, but generated in a special key generation facility and then supplied later using Intel's key provisioning service [34][50].

1.3.3 Summary

In the following we will summarize and compare the properties of Intel SGX with the previously discussed TPM-based solutions as well as with ARM TrustZone and give an insight about the suitability of Intel SGX for trust management in the context of SUPERCLOUD.
The most important feature, which puts Intel SGX alongside with TrustZone, ahead of TPM-based approaches like TrustVisor or XMHF, is the fact that the respective Root-of-Trust is realized on chip-level and integrated into the CPU. Therefore security critical operations can be executed atomically using special CPU instructions. In contrast, solutions, which rely on a TPM as Root-of-Trust, suffer from the fact that the TPM is not indivisibly bound to the CPU. This enables Intel SGX to have a particular security model: Unlike TrustVisor and XMHF respectively, which rely on a trusted hypervisor and BIOS, Intel SGX distrusts all system software including BIOS, drivers, operating system and hypervisor. Additionally, Intel SGX does not suffer from significant performance loss compared solutions like Flicker [63], which directly utilize the TPM. Intel SGX also utilizes software attestation, but other than the previously described SRTM / DRTM solutions, only the contents of each respective enclave need to be measured and attested [34].

Table 1.1: Comparison of hardware-based trust management solutions

	TPM-based solutions	**ARM TrustZone**	**Intel SGX**
Principle	Special-purpose hypervisor based on DRTM principle and TPM-based Root-of-Trust	System divided in secure/non-secure partition on chip-level	Creation of secure partitions in memory isolated from the rest of the system on CPU-level
Economics	Standardized and open specification	Standardized and open specification	Closed specification
Performance	Flicker: slow TPM performance caused by slow communication bus to CPU	Runs on main CPU, hence high speed communication	
	TrustVisor/XMHF: executes at high speed through usage of virtualized TPM on CPU	RAM limited	RAM unlimited
Functionality	Operates with provided systems only	Allows only one TEE	Allows multiple TEEs (enclaves); Tightly integrated into CPU
Flexibility	Flicker: runs arbitrary code without access to OS or drivers XMHF: based on modular construction, hence extensibility is provided to hypapps	Runs arbitrary code	
Security	Somehow physical protection of keys, TPM not well-bound to CPU	Key-storage not encrypted	Key-storage on CPU
	Integrated as hypervisor extension	Integrated on CPU, hence infeasible to modify; CPU sharing between TEE and untrustworthy code	
Drawbacks	Perception of hangs and data loss concerning block-device transmission during Flicker sessions	Only one possible TEE; confined to the SoC market	Intel has full control over licensing and key distribution

When comparing ARM TrustZone to Intel SGX, the most obvious difference is that TrustZone is confined to the SoC market. Notwithstanding this, TrustZone bears the most conceptual similarity to the aforementioned TPM-based solutions. However, there are some differences in how the technology is realized:

- ARM TrustZone only allows the presence of a single TEE on the same system, whereas Intel SGX supports having a number of enclaves on the same machine.
- TrustZone provides no predefined method of software measurement or attestation, software design is completely in the hands of the developers.
- An API is publicly specified, but its implementation is left to third parties. In contrast, Intel provided a publicly available SDK [9] for the SGX.

For sake of clarity, Table 1.1 summarizes and compares the respective properties of the trust management solutions discussed prior in this section.

Its innovative security properties make Intel SGX the approach that most accurately tackles the situation in today's cloud computing scenarios. The potential licensing-related drawbacks mentioned in the previous subsection notwithstanding, Intel SGX emerges as a sophisticated and interesting candidate to approach the implementation of a secure computation infrastructure for SUPERCLOUD, bearing a lot of potential in research.

1.4 Outline of the document

The rest of this document is organized as follows. In Chapter 2, we give an overview of the different prototypes that have been implemented, classifying them as enabler-, use case-, or technology demonstrator-oriented. We then present enabler-oriented prototypes in Chapter 3, use case-oriented prototypes in Chapter 4, and technology demonstrator-oriented protypes in Chapter 5. Finally, we conclude in Chapter 6.

Chapter 2 Implementation Prototypes

The prototypes discussed in this deliverable can be broadly divided into three distinct categories: *Platform Enabler Prototypes*, *Use Case Prototypes* and *Technology Demonstrator Prototypes*.

- The purpose of the Platform Enabler Prototypes is to demonstrate basic implementation components required for realizing the SUPERCLOUD abstraction layer.
- The Use Case Prototypes demonstrate the use of SUPERCLOUD capabilities in realizing actual SUPERCLOUD services.
- The focus of the Technology Demonstration Prototypes is narrower, exploring dedicated novel technology solutions which can be integrated into the overall SUPERCLOUD architecture to provide new or improved computational capabilities for realizing SUPERCLOUD services or platform features.

At this point, the individual prototypes have been developed as stand-alone technical components. In the forthcoming work of SUPERCLOUD, the prototyped components will be further harmonized and integrated with the overall SUPERCLOUD architecture implementation to yield a functional integrated prototype demonstrating the capabilities of the SUPERCLOUD abstraction layer in practice. The relation of the individual implementation prototypes to the overall SUPERCLOUD architecture is shown in Fig. 2.1.

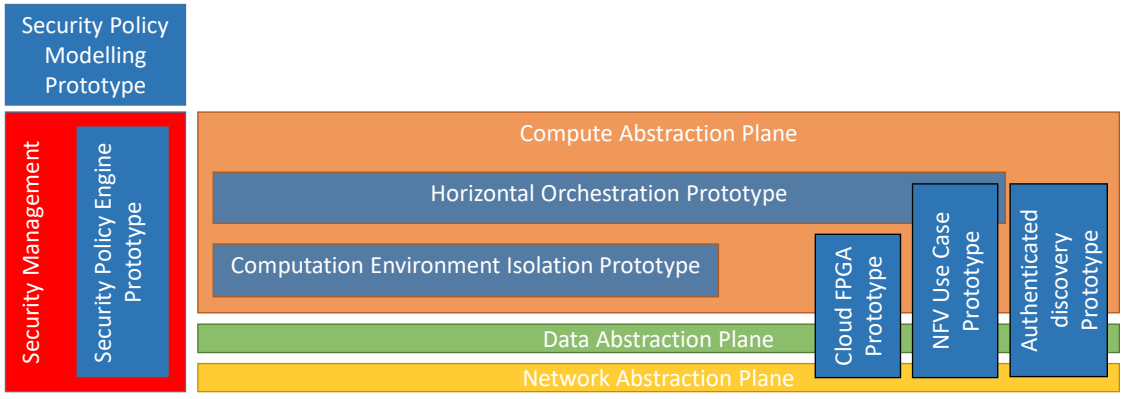

Figure 2.1: Overview of implementation prototypes

2.1 Platform enabler prototypes

These prototypes are described in Chapter 3. The *Horizontal Orchestration Prototype* is discussed in detail in Sect. 3.1. It addresses questions regarding the orchestration of computation environments in the compute plane of SUPERCLOUD across different cloud providers as well as the underlying abstraction layers of the SUPERCLOUD implementation.

Security self-management and policy modelling aspects of the Security Management component of the SUPERCLOUD architecture are demonstrated by the Security Policy Modelling Prototype (Sect. 3.2) and the Security Policy Engine Prototype (Sect. 3.3).

2.2 Use case prototypes

These prototypes are described in Chapter 4. The *NFV Use Case Prototype* and the *Authenticated Discovery Prototype* touch all sub-architecture layers of the SUPERCLOUD architecture. The Authenticated Discovery Prototype (Sect. 4.1) demonstrates approaches for enforcing geolocation requirements for SUPERCLOUD services, while the NFV Use Case Prototype (Sect. 4.2) demonstrates the realization of Network Function Virtualization (NFV) services on the SUPERCLOUD architecture.

2.3 Technology demonstrator prototypes

These prototypes are described in Chapter 5. The *Computation Environment Isolation Prototype* described in Sect. 5.1 addresses issues related to the low-level isolation of computation environments from access by cloud providers, utilizing novel hardware-security technologies like *Software Guard Extensions (SGX)* from Intel.
A similar hardware-based approach is explored in the *Cloud FPGA Prototype* in Sect. 5.2, where the aim is to investigate mechanisms through which optimized FPGA-based computation operations could be offered as part of the SUPERCLOUD service APIs to virtual machines running in user clouds.

Chapter 3 Platform Enabler Prototypes

This chapter describes the Platform Enabler Prototypes that demonstrate basic implementation components required for realizing the SUPERCLOUD abstraction layer. Three prototypes are presented, covering the orchestration of computation environments across different cloud providers – Horizontal Orchestration Prototype discussed in Sect. 3.1 – and security self-management – Security Policy Modelling Prototype described in Sect. 3.2 and Security Policy Engine Prototype described in Sect. 3.3.

3.1 Virtualization and horizontal orchestration

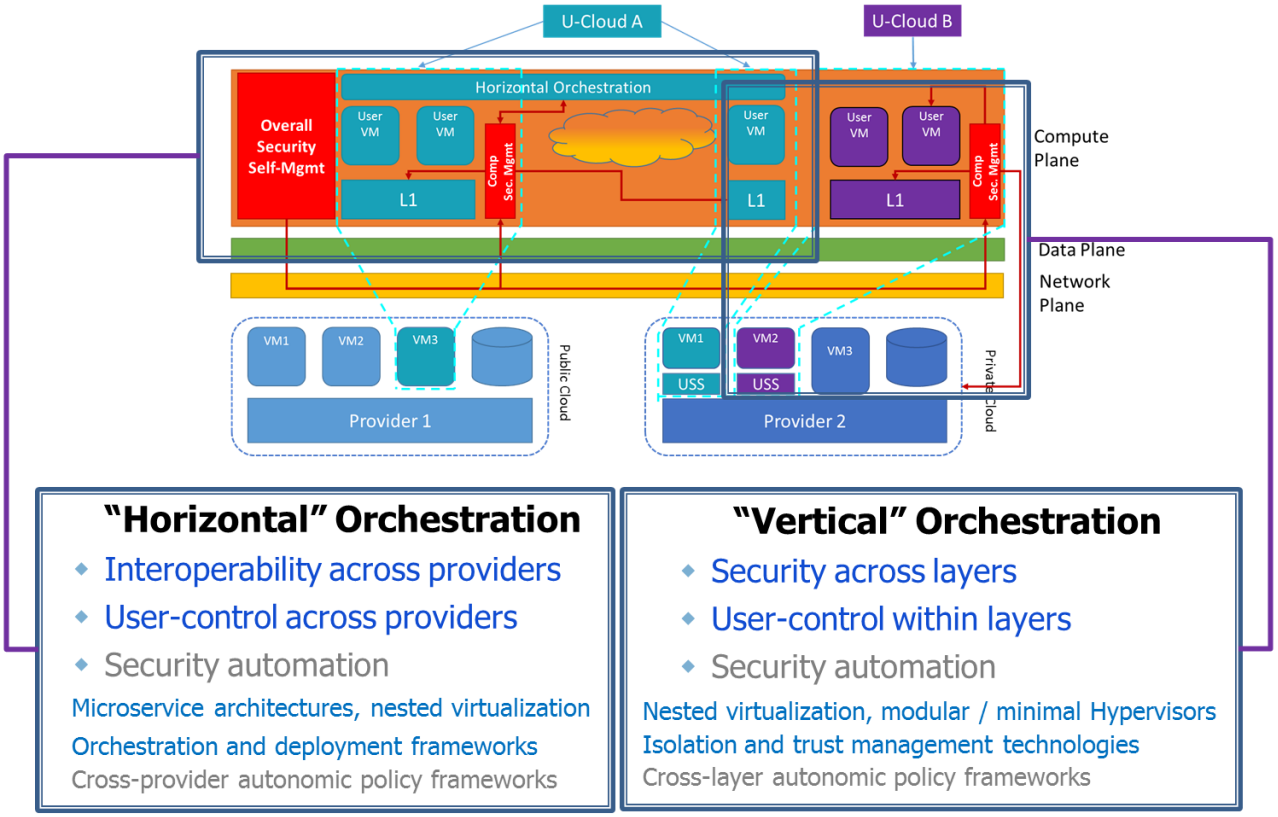

Figure 3.1: SUPERCLOUD computing architecture: horizontal and vertical orchestration

The core of this prototype illustrates a first implementation of the *horizontal computing virtualization architecture*: instantiation and deployment of a distributed multi-cloud, in which security services can be selectively weaved in different areas of the architecture realizing the U-Cloud horizontally (multi-provider aspects), to overcome the interoperability barrier. Applications can then be deployed based on the U-Cloud resource aware-constraints, automatically being in a secure environment. This first aspect, shown in Figure 3.1 will be described in more detail in Section 3.1.1.

A second dimension of the prototype is an implementation of the *vertical computing virtualization architecture*: the modular hypervisor enables users to achieve multi-layer U-Cloud control. This second aspect, also shown in Figure 3.1 will be described in more detail in Section 3.1.2.
A last dimension of the prototype deals with *trust management*: guarantees are provided regarding cross-layer and multi-provider trust management, also in relation with Intel SGX-related isolation technology. Extensions are also ongoing regarding self-management: to allow cross-layer and multi-provider autonomic security monitoring (VESPA framework extension). This third aspect will be described in more detail in Section 3.1.3.

3.1.1 Horizontal orchestration

We present ORBITS, a comprehensive multi-cloud orchestration architecture, that conciliates: (1) flexible provisioning requirements of microservices-based applications, handling placement, elasticity and availability; and (2) infrastructure homogeneity to let the user completely control its security appliances. The architecture also tends to compatibility with the legacy application orchestration logic.

3.1.1.1 Design requirements

This prototype aims to conciliate two main design properties:

- **Flexible provisioning** In the multi-cloud, the application logic should influence resource allocation. The prominent importance of data transfer costs pushes the placement logic nearer to the application, where the role of the deployed microservice and its interaction with other services is easily predictable.

  The application orchestration logic should consider different classes of parameters (e.g. resource costs, replication rate, anti-affinity) to enforce on-demand provisioning of resources, guarantee placement constraints required by MPC use-cases or fault-tolerance for provider outages of EHR.

  Broker-based approaches [18, 10, 37] provide a unified view of multi-cloud resources, reconciling provider specificities and semantics with a "least common denominator" philosophy. This approach enables optimized provisioning of application resources, but without giving extensive control over the infrastructure to users.

- **Infrastructure homogeneity** The multi-cloud should provide an homogeneous infrastructure from security and resource abstraction standpoints. Homogeneity on an infrastructure across multiple sites may be obtained through an overlay layer, implementing user-desired security policies and services, and decoupling provider API from customer usage. Infrastructure homogeneity enables the possibility to have the same security services on each provider to protect execution of applications. Overlay-based approaches [40, 74] provide the user with an important level of control (e.g., virtualization layer, security appliances), but lack effective multi-provider orchestration tools.

A further design property is required to ease transition to multi-clouds:

- **Undisruptive compatibility** Application life-cycles in the multi-cloud should reuse existing tools to manage deployments/releases and should not disrupt development cycles.

3.1.1.2 The ORBITS architecture

In what follows, we consider a simple use-case where users have (1) a microservice-based application with (2) related orchestration logic, (3) a number of N distinct providers and M regions that they require a priori, (4) a set of security services and configurations they want to deploy for protection of applications, and (5) a list of static provider constraints (e.g. legal country, minimum availability).

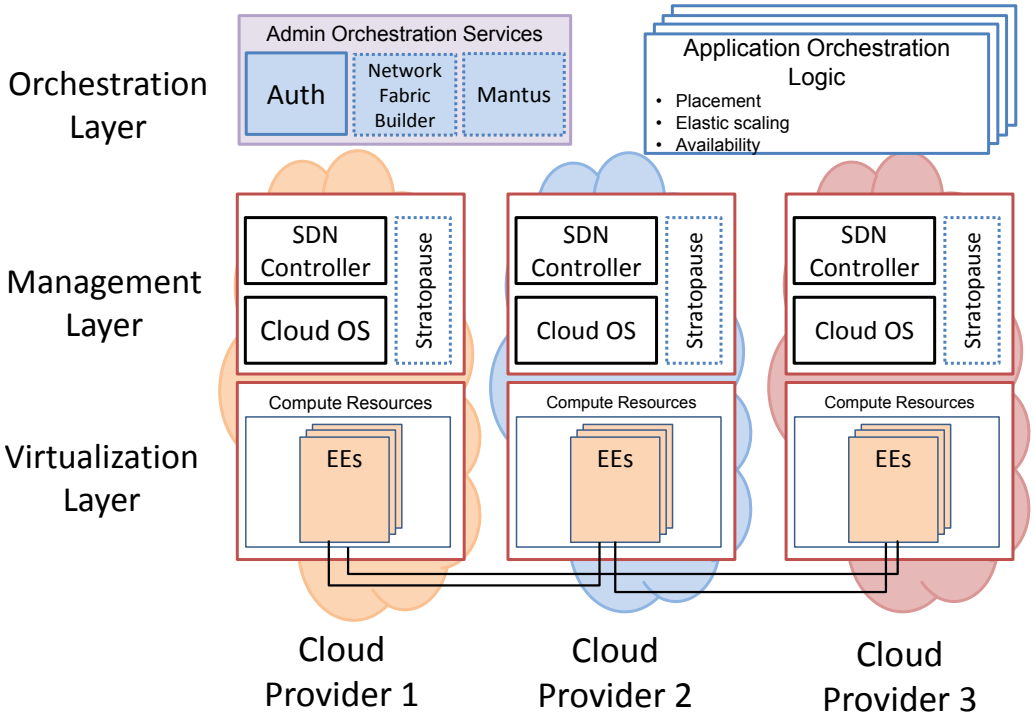

Figure 3.2: The ORBITS architecture. Components with dashed borders represent new services introduced compared to legacy multi-cloud architectures.

Figure 3.2 gives an overview of our multi-cloud architecture. We adopt a three layered-design, following a bottom-up description.

- The *virtualization layer* executes scheduled jobs, with trade-offs between performance and isolation. Virtualization provides isolation between different workloads, leveraging security services specified by the user at build-time. This meets the requirement of homogeneous sets of system security services.
- The *management layer* is in charge of resource provisioning on each overlay provider. This layer would meet the same requirement of homogeneous security services, focusing on not only the execution of applications but also on access to resources.
- The *orchestration layer* coordinates different provider instances and application orchestration. This layer, leveraging its global view of available providers, ensures flexible provisioning across multiple providers required by the use-cases.

Management and virtualization layer services are deployed on each provider selected to be inside the multi-cloud. We refer to those instances as "overclouds", as they are overlay instances that provide the homogeneity layer to the orchestration layer.

3.1.1.3 The virtualization layer

The virtualization layer is in charge of execution of microservices with a provider-agnostic approach. Virtualization is a widely-adopted approach to obtain isolated and transparent hardware resource sharing, between competing software or systems. Several technologies may be adopted to put in place execution environments (EEs), that are generally not interoperable.
In ORBITS, the overlay virtualization layer should realize interoperability among isolated EEs across different providers. This is not obtainable at underlay level because of technological heterogeneity.

The compute virtualization layer should: (1) provide interoperability across the entire multi-cloud, hiding provider heterogeneity; (2) be customizable, allowing the user to deploy its chosen security services; (3) impose minimal performance overhead. Although providers already propose their own isolated execution environments, typically VMs, this remains insufficient, as customers have not enough control to add security components inside [40], or improve performance by enforcing co-residency [74]. Two main technological alternatives are available to realize the virtualization layer:

- **Nested Virtualization (NV)** is a system architecture with two layers of virtualization: the guest OS virtualizes a nested guest [26]. The concept may be generalized to an arbitrary number of nested guest layers, leading to recursive virtualization [39, 41]. This extra level of virtualization may be executed through nested hardware-assisted full virtualization [72] or paravirtualization [24] over hardware-assisted virtualization [40]. Performance of such techniques have always represented an impeding factor for their massive adoption. However some recent works showed more acceptable overheads [40, 26]. The full virtualization alternative requires NV-extension support in the cloud provider hypervisor. This assumption is usually not verified. Thus, we focused on a paravirtualization-oriented design, that does not require any explicit support from the provider, even if not compatible with all existing OSes [40].
- **Containers** are user-space environments on an OS providing isolation between them and host resources [77]. Resource isolation is achieved using new kernel functionalities (e.g., cgroups, namespaces for Linux). Containers still suffer from major isolation concerns due to Linux kernel sharing and achieve weaker isolation than VMs. Initially considered as simple lighter VMs [77], container technologies recently evolved in per-application portable environment [6] introducing a flexible mechanism to package applications. However, this application-oriented life-cycle is slightly incompatible with the IaaS model that deploys general-purpose environments with an application-independent life-cycle. Recent work has also shown that overlay containers do not degrade significantly performance [74].

In both cases, user microservices composing complex application will be run inside EEs provided by the virtualization layer. NV and containers offer different trade-offs in terms of isolation and performance. Therefore, the user can adapt the virtualization technique to microservice workloads, isolating homogeneously across providers components of an application selectively. However, the virtualization layer only introduces EE homogeneity across multiple providers. For complete infrastructure homogeneity, it is necessary to rely on another common layer for resource provisioning. This is the Management Layer, a distributed layer that includes control services of overclouds.

3.1.1.4 The management layer

For infrastructure homogeneity, ORBITS aims not only at virtualization interoperability but at homogeneous resource management across multiple clouds. This implies uniform APIs across providers. Two classes of management services of ORBITS overclouds are introduced, for *local resource provisioning* (Cloud OS and SDN controller) and *relation with the orchestration logic* (Stratopause component).

- **Local Cloud OS and SDN controller** components are normally in charge of compute, storage, and networking management (including tenant accountability). They are the natural choice to implement local resource provisioning.
- **Stratopause** is the link between local resource provisioning and application dispatching. It informs regularly the application orchestration framework about available overclouds, e.g., resources, cloud attributes (provider, region, virtualization technologies). When the application orchestration logic schedules a job, Stratopause ensures it is run in the most adequate user-controlled EE, leveraging eventual co-residency bootstrap. The orchestration logic collects updates from Stratopause instances to reach placement decisions. Stratopause also collects

Table 3.1: Compute and network security services.

	Compute	Network
Management Layer	Security Management Module [13] Hardening [8, 19]	DDos Protection [5]
Virtualization Layer	Introspection [7] MAC security profiles [4]	Middleboxes [76, 31]

microservices dispatching commands to local overlays, which are transmitted to the local Cloud OS to provision resources according to expressed requirements.

The management layer enables using equivalent security services on different providers, e.g., to fulfill EHR systems security requirements. However, this layer does not have the overall vision of all deployed overclouds. An orchestration layer is thus needed to coordinate different overclouds instances for flexible provisioning of tasks across providers.

3.1.1.5 The orchestration layer

The orchestration layer is composed of two classes of components performing orchestration at *infrastructure* and *application* levels.

Infrastructure orchestration

Infrastructure orchestration is realized by the components of the Admin Orchestration Services shown in Fig. 3.2. The MANTUS component deploys management and virtualization layers on selected providers, whereas the Network Fabric Builder provides on-demand interconnection between such providers and the Authentication & Authorization Service manages identity and access across overlay instances. In the following, we will provide an overview of each of these components.

3.1.1.5.1 Mantus: multi-cloud overlay construction

MANTUS defines and deploys the overlay cloud infrastructure on a selected group of providers. It leverages a cloud template text-description for the overlay-infrastructure following the "Infrastructure as Code" paradigm, where deployed services are defined [46]. MANTUS objectives are to: (1) customize the cloud template according to tenant requested security services; and (2) select a subset of cloud providers, compatible with policies expressed by the tenant needs. Such services may concern network and system control, management services, and virtualization and data plane.
To instantiate overlay clouds on multiple providers, the MANTUS orchestration workflow is as follows:

1. **Service definition** MANTUS uses a code description to automate infrastructure resource provisioning and configuration, providing benefits in terms of reproducibility and maintenance. Such description concerns services from management and virtualization layers (Cloud OS services, SDN controller, and "virtualization" nodes).

   In a private cloud, the customer has the highest level of control over security: he may place the desired protection mechanisms on all available services. The overcloud instantiated by MANTUS offers the same level of control as in a private cloud, formalized in a provider-independent text description. Table 3.1 shows some examples of different classes of security services that may be deployed on each overlay cloud.

2. **Service enrichment** As shown in Figure 3.3, MANTUS extends the abstract service description with the list of security services the user provided as input (as shown in Table 3.1). The initial description is then enriched by the addition of selected services from providers. *Access control*

and *hardening services* may be introduced as new services in the provider-agnostic description, that normally should have network connectivity with control services. Network applications may be described as configuration files that have to be deployed inside the *SDN controller* description, similarly for *hypervisor appliances* that have to be added to compute nodes. Finally, *network middleboxes* may be described as extra services, chained together by traffic steering flows. Such service chaining-oriented description requires a clear definition to express steering rules and transmit such model to local virtual network fabric builder. Several works [76, 31] addressed similar problems and may be taken as reference.

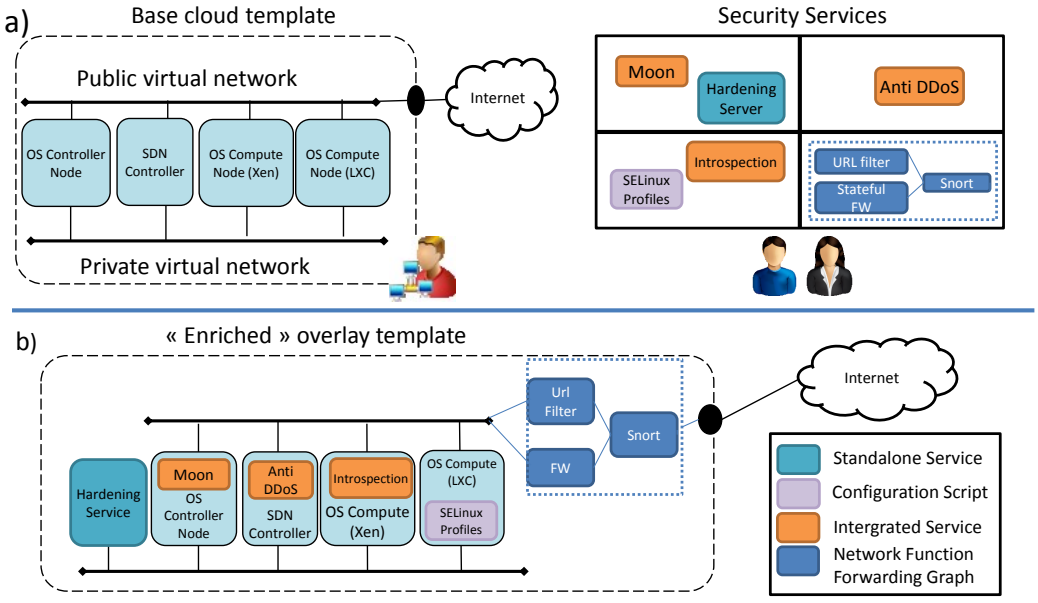

Figure 3.3: (a) Initial sample overlay template; (b) services after the enrichment process.

3. **Filtering** In parallel to steps 1. and 2., MANTUS retrieves a list of available providers, and applies on it a simple "filter & weight" algorithm. We make the assumption that MANTUS may retrieve a list of provider regions with pre-defined and comparable SLAs. Initially, the filtering logic will drop providers that do not satisfy static requirements, expressed by the tenant. Such requirements may include technical constraints (e.g., minimal availability) or non-technical constraints (e.g, legal residence of the provider, location of specific regions).

4. **Weighting** The tenant specifies the minimum amount of M distinct providers and a total number N of regions it requires. According to this specification, MANTUS will grade remaining provider regions with a utility function, giving more importance to cost savings (e.g. virtual compute instances prices, incoming/outcoming data transfers costs) or to location for roughly optimizing latency issues for customers. N regions and M distinct providers with the highest weights will be selected for instantiation. We assume that the amount of requested providers is fixed at instantiation time and cannot change during execution of applications.

5. **Instantiation** When providers are selected, the provider-agnostic description of services is converted into the one of the selected cloud providers.

3.1.1.5.2 Network fabric builder

Virtual networks are created inside each overlay cloud by hosting cloud providers. To create multi-provider connections, a network fabric builder should extend local virtual networks across provider barriers. Such virtual network connectivity should be transparent to microservice-based applications.

3.1.1.5.3 Authentication & authorization service

This component manages transparently identity and access for tenants and administrators across deployed overclouds. The design of some typical authorization services are described in Deliverable D2.1. Typical authorization policies may for instance be compared using the approach explored in [58].

Application-level orchestration

While the role of infrastructure services is building and maintaining of the ORBITS multi-cloud, flexible provisioning across clouds is the role of the application orchestration logic – typically for placing application microservices across providers.
Orchestration frameworks are usually composed of *application frameworks* and of a *resource multiplexer* (e.g., Mesos [2]). Application frameworks are responsible for application deployment on available resources, following a user specification. The resource multiplexer guarantees fair sharing between frameworks on a pool of resources.
Several classes of applications may rely on well-known and widely adopted frameworks, e.g., for data-oriented jobs [3, 1]), general-purpose applications [11, 12]. Newer frameworks also target further classes, e.g., network services . For generic applications, frameworks use declarative languages to automate deployment, scaling, and operations. In our scenario, we enhance the placement logic of application frameworks, introducing the awareness of multi-provider overclouds, deployed by MANTUS. The overcloud-aware placement leverages STRATOPAUSE instances to receive updates about (1) overcloud instance availability and (2) dispatch on a certain provider a selected job.

3.1.1.6 Architecture realization

We are implementing a proof-of-concept prototype of this architecture using Mesos, OpenStack Kilo, introducing a first implementation of MANTUS and STRATOPAUSE. We use Xen and LXC as virtualization technologies. For the management layer, we adopted OpenStack ([17]) as Cloud OS deployed on overlay clouds. OpenStack services are deployed at instantiation time, while compute node will be added/removed on the fly.
We started implementing MANTUS and the Network Fabric Builder with Keystone federation. TOSCA [14] service modeling provides a declarative language to describe application deployments, providing interoperable description for cloud services. When cloud providers are identified, the TOSCA description of service is mapped to a per-provider description, like OpenStack Heat HOT or AWS CloudFormation. Finally, we are working on a Mesos orchestration framework to validate the flexible provisioning capabilities of STRATOPAUSE. STRATOPAUSE acts as a fake Mesos slave, communicating to Mesos master, the resource multiplexer, available resources and overlay cloud attributes. When an application framework places a task on overcloud ,the respective STRATOPAUSE instance will instruct OpenStack services to provision resources to execute such tasks, enforcing the most appropriate virtualization technique (LXC or paravirtualized XEN VM), and if necessary inform the OpenStack scheduler about affinity/anti-affinity with previously deployed tasks.

3.1.2 Vertical orchestration

The goal of this prototype is to validate the feasibility of the "vertical" system architecture" (see D2.1) by proposing a first "node" which can instantiate U-Clouds in a non-distributed context. We also aim to evaluate performance degradations induced by the usage of multiple layers of virtualization, and the security potential of such an architecture facing concrete workloads.

3.1.2.1 The NOVA microhypervisor and Genode framework

NOVA OS Virtualization Architecture (NOVA) is a research microhypervisor that combines high-performance hardware-assisted virtualization for the x86 architecture with a minimal TCB (around

10 KLOC) and the capability-based security design of L4 micro-kernels. Such features make it a good candidate for the L0 hypervisor of our architecture.
Like other micro-kernels, NOVA suffers from limited hardware support and the complexity of developing or porting user-level applications to its specific environment. This drawback may be alleviated by using Genode, an open-source modular OS framework to build special-purpose OSes running on various platforms, such as NOVA. Genode provides ready-to-use drivers for the most common devices, and several programming primitives making the link between the kernel API and user-level applications.
NOVA and Genode have a similar approach to system architecture, enabling Genode framework concepts to map directly onto the micro-kernel structure, e.g., both rely on capabilities for security and isolation. Their architectures differ from traditional monolithic systems in several ways.

3.1.2.1.1 Memory management

Strong isolation is required between different user-level applications. To guarantee that components may only access their assigned memory, each of them is placed in a *Protection Domain* (PD). This kernel object has its own virtual memory and I/O mappings, similarly to a VM. Each PD can have associated threads, called *Execution Contexts* (EC). A PD maintains a table of *capabilities*, referencing other kernel objects such as PDs. This is the only way to access them. Capabilities are added to this table whenever an object is created inside the PD, and can then be delegated to other PDs at will through inter-domain communication.
Genode adds another layer of abstraction for memory management: only the root user-level component interacts directly with memory pages. The other components can then obtain memory from it through remote procedure calls: the root component creates a kernel object called a *dataspace* that represents a contiguous address-space region, and sends the requester a capability to the dataspace. Using this capability, the requester can map the dataspace into its local address space and access it.

3.1.2.1.2 CPU scheduling

In NOVA, an EC cannot be scheduled for execution on the CPU unless associated with a *Scheduling Context* (SC). This kernel object defines a priority and a time quantum. The microhypervisor implements a priority-based preemptive round-robin scheduling between the SC. Multiple SC can be assigned to a single EC. An EC without SC can still be run if one is lent by another EC.

3.1.2.1.3 Inter-domain communication

NOVA only supports synchronous communications between protection domains using RPCs. To allow others to communicate with it, a PD can create any number of kernel objects called *portals*, and delegate capabilities to them. The communication is then initiated by an hypercall from a caller EC specifying a capability to the targeted portal and various message-specific parameters. On the target side, the portal is bound to a handler EC without SC. The kernel activates this handler by giving the caller's SC to him. Once the handling operation is finished, the handler returns the SC to its original owner through another hypercall, along with the return values of the handling operation. Genode uses this mechanism to implement remote procedure calls (RPC) between components, but also asynchronous notifications and both synchronous and asynchronous bulk transfer.

3.1.2.1.4 User-level organization

In Genode, the organization of user-level components clearly benefits from the microkernel-based architecture. When started, a component only has capabilities to its parent. A hierarchical structure built recursively has the following benefits: (1) the behavior of components is position-agnostic; (2)

only the policies regarding the transmission (or not) of capability delegation from other components allows the parent to influence the activity of its children. Resource management follows the same recursive pattern: each component is given a limited quota of physical resources by its parent when created, taken directly from the quota of this parent. This mechanism enables strong isolation between the different branches of the recursive hierarchy in terms of resource consumption. Moreover, because a parent always retains the possibility to take back his resources by killing the child, the hierarchical organization offers a useful abstraction for the management of isolated subsystems.
Interactions between components are handled through *sessions*, a user-level abstraction that extends the kernel-level RPC mechanism and provides a secure client-server relationship implementation. Genode also permits safe resource delegation between components: resource exchange is allowed only between a parent and his direct child, but it can be extended by recursively relaying a change of quota upstream from the sender until a common parent is found, and then downstream to the receiving component. Such resource usage accountability ensures the robustness and the security of the system.

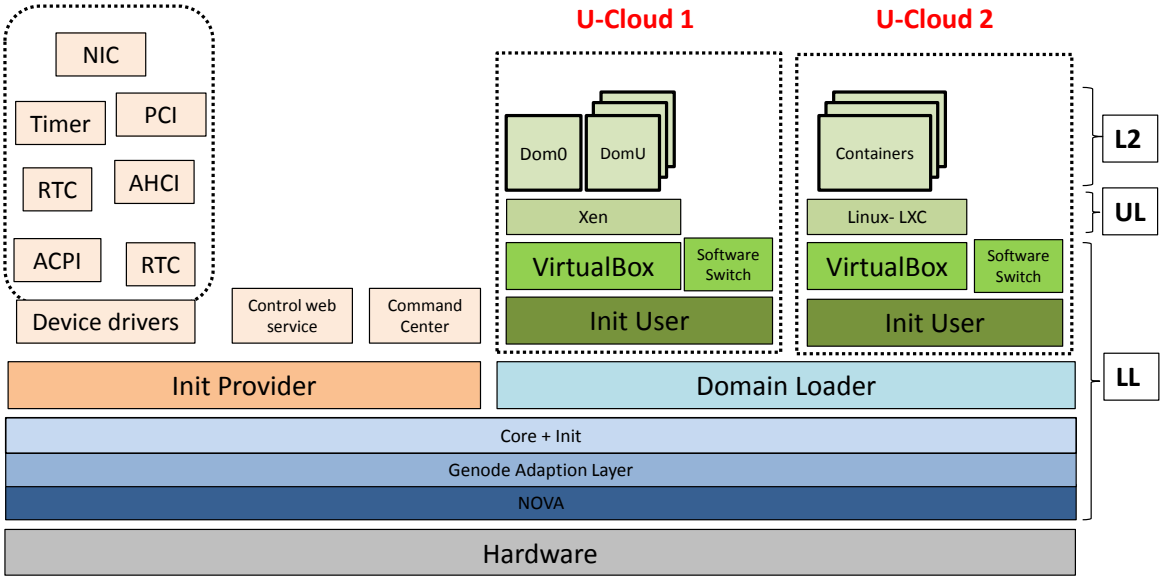

Figure 3.4: Vertical Architecture Overview

3.1.2.2 Core components

The prototype architecture consists of several Genode components, interacting together via the previously described mechanisms. These components fall into three main categories: (A) *launchers and roots*, (B) *drivers and resource multiplexers*, and (C) *virtualization components* (see Figure 3.1.2.1.4).

3.1.2.2.1 Launchers and roots

These components are responsible for instantiating hierarchically the different subsystems of the architecture. They require the configuration of the subsystem to be provided (generally in the form of a ROM module, whose content is formatted in XML), along with an appropriate amount of resources, either by their parent or through a specific session.
Core is the root component of the Genode Framework. This is always the first to be built when a system is started. It seats directly above the micro-kernel, and handles every interaction between it and the user-level. As such, it is the only component to directly access the raw resources without depending on Genode abstractions (services and sessions). It is however subject to the same standards as any other user-level component in terms of resource usage accountability and isolation.

Init is another base component of the Genode Framework. Second component to be created, directly by Core, it is then responsible for instantiating the entire system. To that end, Init obtains every available resources from its parent (Core) via sessions, and parses an XML file describing every components to be created, and the resources that need to be allocated to them. Init is not necessarily the child of a Core component, and would behave similarly elsewhere in a component hierarchy.

3.1.2.2.2 Drivers and multiplexers

The *drivers* are the interface between the physical devices and the system. They allow Genode components to interact with the devices through dedicated sessions. Because a device driver only allows one simultaneous session to be opened, in systems where several components must share a physical resource, *multiplexers* must be added to seamlessly handle it.
In our prototype, the drivers include a platform driver for the PCI controller, an ACPI driver, a timer driver (using NOVA semaphore mechanisms as back-end), an RTC driver, a NIC driver, and an AHCI driver. Hard drive access multiplexing is provided by a block device partition server, and the file systems are handled by a port from the rump kernels.

3.1.2.2.3 Virtualization components

The microhypervisor approach of NOVA removes the Virtual Machine Monitoring functions from the privileged hypervisor. Running a virtual machine on NOVA thus requires a VMM to be executed as a user application. A specific VMM named Vancouver was developed alongside NOVA, providing high performance and a reduced code-base (around 20 KLOC). However, Vancouver is somewhat lacking in features, and does not offer the same flexibility and wide support as more common technologies. Genode includes a port of VirtualBox that runs entirely in user-level and supports hardware-assisted virtualization.

3.1.2.3 Additional components

From this initial subsystem, were developed some more system services to address: (1) complete functionality of virtualization system; (2) isolation of user and administrator domains; and (3) secure remote control over deployed resources. Such new components (represented with dashed borders in Figure 3.1.2.1.4) are briefly described next.

3.1.2.3.1 Launchers and roots

InitU and InitP are modified versions of the standard Init that we developed to mitigate some shortcomings consequent to its minimality. They are used as root of every user domain and of the provider domain, thus representing the single interface for their interactions with the rest of the system. While Init was already able to play that role, it lacks dynamicity. For example, the configuration of the subsystem cannot be modified at run time without destroying it entirely and rebuilding it with the modification. Our modifications help reach the high flexibility desired for our user-centric approach. Because the requirements differ slightly between user and provider domains (for example the InitU of the user domain is built and destroyed on the fly, whereas the life cycle of the provider's InitP is the same as the one of the system, we introduced two separate components.
The *Domain Loader* deals with the delegated instantiation of the user domains. To allow the launching of a user domain from a provider-space service while protecting the launched domain from interference by the provider, we needed to delegate the actual instantiation to a trusted component outside of the provider domain. The Domain Loader plays that role, obtaining the configuration of the subsystem and the resources through a specific session. Because it is placed under all the InitU (and thus is

parent to every user domain), the Domain Loader can control their interactions with the rest of the system by choosing how to route their session requests.

3.1.2.3.2 Drivers and multiplexers

The *Network Switch* is a pure L2 software switch, which implements the backward learning algorithm over multiple VLANs. A different instance is created in each U-Cloud to enable connectivity for user nested VMs and hypervisors. A privileged instance of the switch has to be executed among the administration services to multiplex NIC resources among different users.

3.1.2.3.3 Control logic and interfaces

For our architecture to be usable as a node in a multi-provider cloud, we needed to implement several components that would allow a remote control of the system, while still maintaining the decentralized, hierarchical structure characteristic of Genode. Our prototype includes two services to address this :

- *Epimetheus* is a minimal REST lwip-based web service that provides a simple way of remotely controlling the system. For example, when an adequate PUT request is received, the web-service opens a ephemeral session on Prometheus and use it to transfer the configuration options of the requested subsystem. Epimetheus could be easily expanded to handle a variety of management operations.
- *Prometheus* is the central component of the control logic in our system. It is the link between the different control interfaces (in our prototype, Epimetheus instances) and the other components. As such, the Prometheus instance implements several checks to ensure the validity of the requests and the availability of the needed resources. Prometheus represents the single-point of control for the administrator: Prometheus conserves an opaque handler to allocated resources leaving the possibility to the administrator to kill a "user" subsystem if detecting nasty activities.

3.1.3 Trust management

This part of the prototype provides a first implementation of the following components of the SUPERCLOUD Virtualization Architecture: *Trust Management* (management of cross-layer, and cross-provider Chains of Trust) and *Isolation* (hardware-enforced using Intel SGX technology). Integration with *Self-Management* components (using a framework like VESPA) will come in a second step.

3.1.3.1 Design goals

Currently, two elements seem to be missing for a comprehensive trust management and isolation framework for multi-clouds:

- A first requirement is to establish and verify the integrity of the link between a virtual machine (VM) and hardware resources. The Trusted Computing Group introduced the *Chain of Trust (CoT)* abstraction: integrity of a component may be verified by following the CoT to a root of trust (RoT), usually a tamperproof hardware element such as a TPM [78]. Abbadi et al. defined a model to describe CoTs in a multi-cloud infrastructure, both vertically (across layers) and horizontally (across domains) [21]. However, it remains unclear how to map this model to concrete cloud isolation technologies.
- A second requirement is to guarantee secure execution of VMs with hardware protection even if some intermediate infrastructure layers are compromised. Intel's Software Guard Extensions (SGX) [64] through the *enclave* abstraction for a secure computation unit provides significant enhancements compared to previous isolation solutions (e.g., [61, 78]): it guarantees VM security

even if the hypervisor is completely compromised, reducing the TCB only to the CPU chip. This isolation technology could provide a starting point towards a comprehensive security framework handling both types of CoT, and supporting multiple isolation technologies. However, it remains unclear how enclaves can be chained together practically.

This prototype proposes a protocol for establishing trust between chains of Intel SGX enclaves. The protocol formalizes horizontal (single layer) CoT establishment for multi-cloud infrastructures according to the model of [21], both when enclaves are located on the same and on remote Intel SGX platforms. Attestation protocols have been implemented on the OpenSGX [51] Intel SGX emulator. Preliminary evaluation results tend to show our protocols could scale to large CoTs, thus being applicable to realistic multi-cloud infrastructures.

3.1.3.2 Chains of trust in a multi-cloud

Abbadi et al. proposed a simple architectural model to manage trust in a distributed cloud [21] (see Figure 3.5).

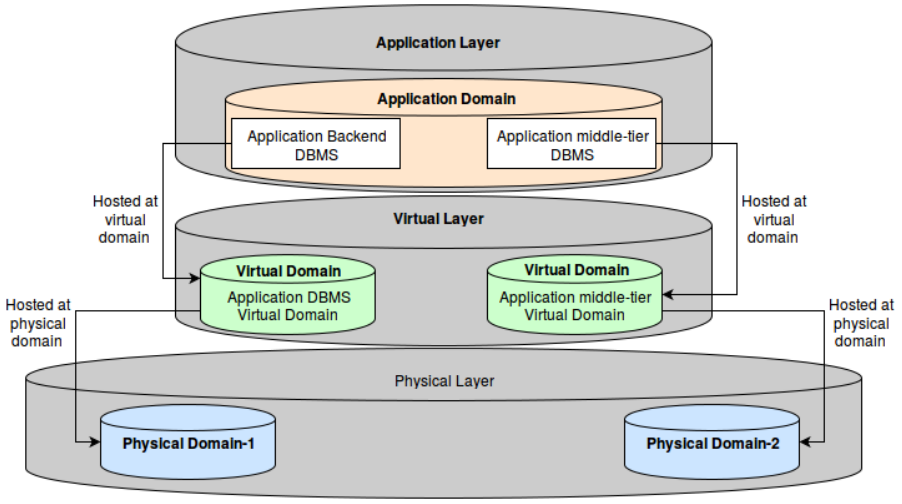

Figure 3.5: Multi-cloud infrastructure model

- Vertically, the infrastructure is modeled as several layers, software and hardware containing resources. The *physical layer* consists of computing (CPU, memory) and storage hardware resources. The *virtual layer* contains the VMs (virtual CPU and memory) and virtual storage. The *application layer* leverages virtualized resources to run applications. One or several *virtualization layers* manage allocation of host resources among VM instances.
- Horizontally, the infrastructure is seen as a federation of *provider domains* that manage resources within a given perimeter, and according to a common policy. *Domain federations* group together domains in each layer.

In a *Chain of Trust (CoT)*, the links between elements of the chain element represents confidence between two entities: a *Trustor* and a *Trustee*. To establish this confidence, an assertion is needed in order to demonstrate that a piece of software has been properly instantiated on the platform. This process is known as *attestation*: it provides a Trustor with an authentic and fresh copy of the properties of a Trustee. Thus, the Trustor can make timely decision of the ability of the Trustee to operate in certain state.
A *Root of Trust* (RoT) is a component that must always behave in the expected manner: its misbehavior cannot be detected. The RoT set includes at least the minimal set of functions enabling a description of the platform characteristics that affect its trustworthiness.

A CoT provides an iterative means to extend the trust boundary from the RoT set to extend the collection of trustworthy functions. Typically, a CoT could be built as follows. The first element of the CoT (RoT) should be established from a trusted entity or an entity that is assumed to be trusted, e.g., a tamper-evident hardware chip. The RoT then measures the trust status of the CoT second element. As the verifier trusts the RoT, the verifier also trusts the RoT measurement of the second element that is now fully part of the CoT. This process is continued to other elements of the CoT (see Figure 3.6). This approach may be extended to manage trust relationships in the cloud using CoTs that may cross layers (vertically) or domains (horizontally).

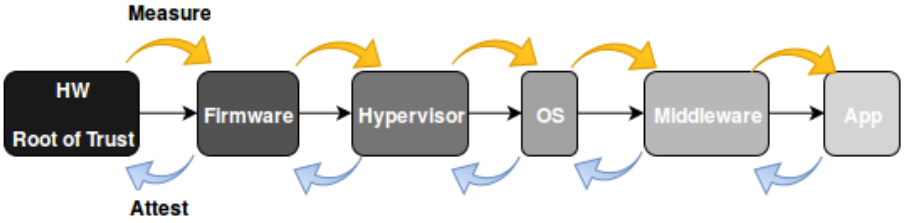

Figure 3.6: CoT concept

3.1.3.3 Intel SGX and OpenSGX

As discussed in Sect. 1.3, Intel's Software Guard Extensions (SGX) is an extension to Intel architecture for generating protected software containers, referred to as *enclaves*. Inside an enclave, software code, data, and stack are protected by hardware-enforced access control policies that prevent attacks against the enclave content. SGX allows part of an application code to run isolated inside an enclave. The enclave region of the main memory is encrypted. The content is only decrypted inside the CPU using processor-specific keys. The TCB (Trusted Computing Base) is restricted to the CPU and the application running inside the enclave. Even an adversary with extensive control over the hardware cannot access or modify the enclave. The enclave is protected from other software running in the host, including the OS and the hypervisor.
SGX features allows building a CoT based on three elements:

- A RoT for storage materialized by the sealing keys for the enclave software to encrypt and integrity-protect data.
- A RoT for measurement captured by two measurement registers, MRENCLAVE and MRSIGNER: MRENCLAVE returns an identity for enclave code and data; MRSIGNER returns an identity of an authority over the enclave. These values are recorded when the enclave is built, and are finalized before enclave execution begins. Only the TCB has write access to these registers to reflect accurately the identities available when attesting and sealing.
- A RoT for reporting, equivalent to the report mechanism provided by EREPORT and EGETKEY instructions: an evidence structure called *REPORT* is returned, cryptographically bound to the hardware for consumption by attestation verifiers.

With Intel SGX, a trustor can gain confidence that the correct software is securely running within an enclave on the trustee. In order to do this, SGX architecture produces an attestation assertion that conveys: the identities of the software environment being attested, details of any non-measurable state (e.g. the mode the software environment may be running in), data which the software environment wishes to associated with itself and a cryptographic binding to the platform TCB making the assertion. To build a CoT, enclaves will need to authenticate one another. The EREPORT instruction provided by the SGX architecture will be particularly useful for this purpose. When invoked by an enclave, EREPORT creates a signed structure, known as a REPORT. The REPORT structure contains the

identities of the two enclaves, the attributes associated with the source enclave, the trustworthiness of its hardware TCB, additional information to pass on to the target enclave (e.g., USERDATA), and a message authentication code (MAC) tag.
OpenSGX [51] is a fully functional, instruction-compatible Intel SGX emulator to explore the software/hardware design space. It is also a platform to develop enclave programs, providing additional OS components, such as an enclave program loader/packager, and debug/performance monitoring tools.

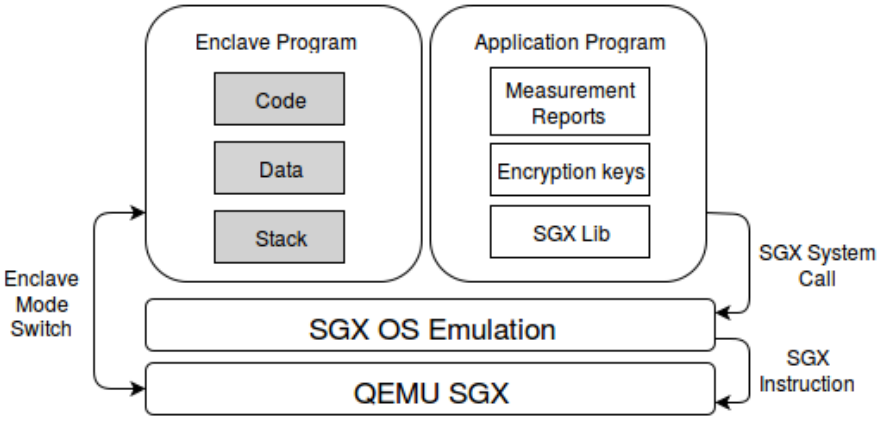

Figure 3.7: OpenSGX design overview

A packaged program (Wrapper) runs together with the Enclave Program together as a single process in the same virtual address space. Figure 3.7 shows the memory state of an active Enclave Program (grayed boxes are isolated enclave pages). As Intel SGX uses privileged instructions to initialize and set up enclaves, OpenSGX introduces a set of system calls to service requests from the Wrapper program.

3.1.3.4 CoT attestation protocols

Building a CoT implies a series of mutual attestations between neighboring elements of the chain to build persistent links of confidence. Depending whether elements of the chain belong to enclaves within the same SGX platform or to remote platforms, two attestation sub-protocols have been specified: intra-platform attestation and remote attestation respectively – a complete CoT establishment being a composition of the two protocols.
Intra-platform attestation protocol: This protocol establishes trust between two enclaves (A and B) on the same Intel SGX platform. Each enclave authenticates the other. The protocol confirms they both run on the same platform according to the SGX security model. The protocol steps are the following, where A is the Trustee and B is the Trustor (see Figure 3.8):

1. Once communication paths between enclaves are established, enclave A obtains the identity of enclave B (MRENCLAVE value).
2. Enclave A invokes the EREPORT instruction to create a signed REPORT sent to enclave B (using the previous MRENCLAVE value) over the untrusted communication channel.
3. When it receives this REPORT, Enclave B calls the EGETKEY to retrieve its Report Key, re-computes the MAC over the REPORT structure, and compares the result with the MAC attached to the REPORT. A match in MAC values means that enclave A runs on the same platform as enclave B.
4. Mutual authentication is achieved by Enclave B creating a REPORT for enclave A, using the MRENCLAVE value from the REPORT it just received.

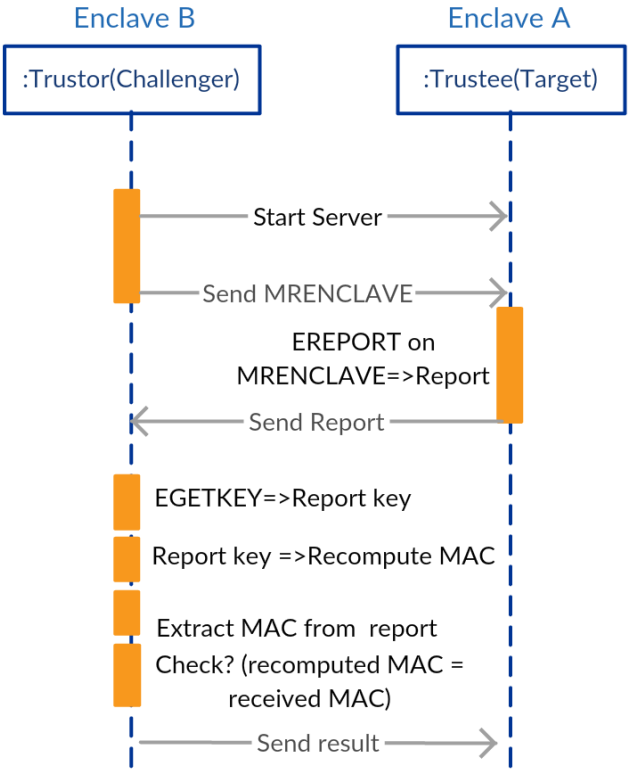

Figure 3.8: Intra-platform attestation

5. This REPORT is sent to enclave A that can then verify it in a similar manner to confirm that enclave B runs on the same platform as enclave A.

Remote attestation protocol: In case of remote platforms, SGX enables a special enclave called the *Quoting Enclave* to be remotely created. This enclave verifies REPORTs from other enclaves on the remote platform using the intra-platform attestation protocol described above. It then replaces the MAC over these REPORTs. The output of this process is called a QUOTE. The protocol steps are the following (see Figure 3.9):

1. Initially, the Trustor enclave establishes communication with the remote target (Trustee).
2. The Trustor issues a challenge to the target to obtain proof the Trustee can join the CoT. The challenge contains a nonce for liveness purposes. It also contains the Quoting Enclave identity.
3. The target enclave generates an ephemeral public key for the Trustor to use in further communications with the target.
4. The target enclave generates a manifest that includes a response to the previous challenge. A hash of this manifest is included as USERDATA for the EREPORT instruction that will generate a REPORT binding the manifest to the enclave.
5. The target enclave then sends the REPORT to the Quoting Enclave for verification and signing.
6. Playing the role of challenger for the target enclave, the Quoting Enclave retrieves its Report Key using the EGETKEY instruction and verifies the REPORT. The Quoting Enclave creates the QUOTE structure, signs it, and returns it to the target enclave.
7. The target enclave forwards the QUOTE structure and any associated manifest of supporting data to the Trustor enclave.

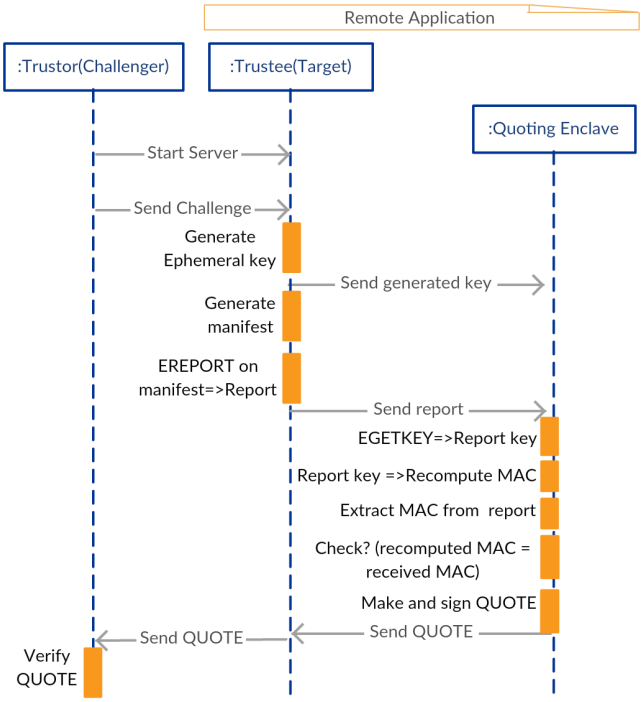

Figure 3.9: Remote attestation

8. The Trustor uses certificate and revocation information or an attestation verification service to validate the signature over the QUOTE. After checking manifest integrity, it checks the validity of the response the initial challenge.

3.1.3.5 Prototype realization

The OpenSGX project provides mainly customized libc and lightweight cryptographic libraries, and basic wrapper functions for SGX instructions. Our implementation is based on those libraries. It provides a user API to build, verify, and study CoTs.
To guarantee attestation, SGX allows creating cryptographic keys (EGETKEY) and cryptographic reports (EREPORT) to check the integrity of an enclave during exchanges with other enclaves. We thus specify an interface to create keys, get reports from SGX, and check integrity of reports by comparing the computed report with a received one. An additional interface manages communication channels between elements of a CoT such as network connections between enclaves, and reading/writing to/from enclaves. Finally, a user API is defined based on previously established APIs to perform the main attestation protocol operations. Several procedures are distinguished depending on the type of SGX platform (local or remote) and to the attestation role (challenger or target).
More details, as well as preliminary encouraging scalability results may be found here [54]. Those CoT-building protocols are a first step towards distributed trust management for a plurality of enclaves. Future work includes: (1) extending our framework to manage CoT vertically across infrastructure layers; and (2) integration with a security self-management framework for distributed clouds such as VESPA.

3.2 Security Policy Modelling

The Security Policy Modelling prototype is built around the capabilities of the MotOrBAC editor, which is used in SUPERCLOUD for setting up and updating security policies related to U-Clouds.

3.2.1 Security Modelling Tool: MotOrBAC Editor

The OrBAC model has an associated tool called MotOrBAC that was developed by Institut Telecom (IT) to help to design and implement security policies using the OrBAC model. The current versions of this tool can design, upload and store security policies and simulate them. The policy simulation can be used to verify the consistency of a security policy. The tool can also detect potential conflicts and help the designer eliminate them. MotOrBAC exists in two versions. The first version is completely free and is distributed under GPL license. The second version, newer and more functional and actively developed, is partially open source and is distributed under Mozilla license. Unlike the first version, MotOrBAC V2 is implemented entirely in Java. It uses an API that has been specially developed to incorporate features of the implementation of the OrBAC model in existing or under development applications. This API, the OrBAC API, is not open source but can be requested on the official Web site of OrBAC.

3.2.1.1 MotOrBAC Architecture

MotOrBAC uses the OrBAC application programming interface (API) to manage the policies displayed in the graphical user interface (GUI). The OrBAC API can be used to programmatically create OrBAC policies. The concrete security policy inferred by MotOrBAC can be translated to configure a security component such as a firewall for example. Another solution to enforce an OrBAC security policy is to use the OrBAC Java API, on top of which MotOrBAC is built. This API uses the Jena java library3 to represent an OrBAC policy as a RDF graph. It can be used to load MotOrBAC RDF policies and interpret them, i.e. access control requests can be made on a loaded policy. Jena features an inference engine which is used by the OrBAC API to infer the concrete policies and the conflicts.
The MotOrBAC architecture is defined in Figure 3.10.

When an OrBAC RDF policy is loaded by the API, the concrete policy can be inferred and stored in memory. An instance of the OrbacPolicy java class which encapsulates an OrBAC policy uses a cache of concrete security rules to enhance the performances when the policy is queried. Contexts are evaluated upon a query; this feature is actually used in the MotOrBAC simulation tool to show the contexts state. The contexts implementation can be easily extended in order to interface the API with other applications and add new types of contexts. Integrating the OrBAC Java API into a Java application can be done without modifying the application source code. Aspect Oriented Programming (AOP) can be used to separate security concerns from other concerns relative to the application.

3.2.1.2 MotOrBAC Functionalities

MotOrBAC can be used to perform several tasks on OrBAC security policies:

- **Edit Policies:** the administrator can create the abstract entities he/she needs (organizations, sub-organizations, roles, activities, views, contexts) and the abstract security policies. Hierarchies defined for these concepts are also defined. These different concepts and policies can be expressed and defined through a graphical interface. Figure 3.11 shows the definition of hierarchies as a graphical tree. Many types of security rules can be specified using MotOrBAC. We can define permissions, prohibitions and obligations. Figure 3.11 shows these different types. Many types of contexts are available to express conditions of rules activation. Each type of context define a specification language like Beanshell language (set of Java language), Prova (set of Prolog).

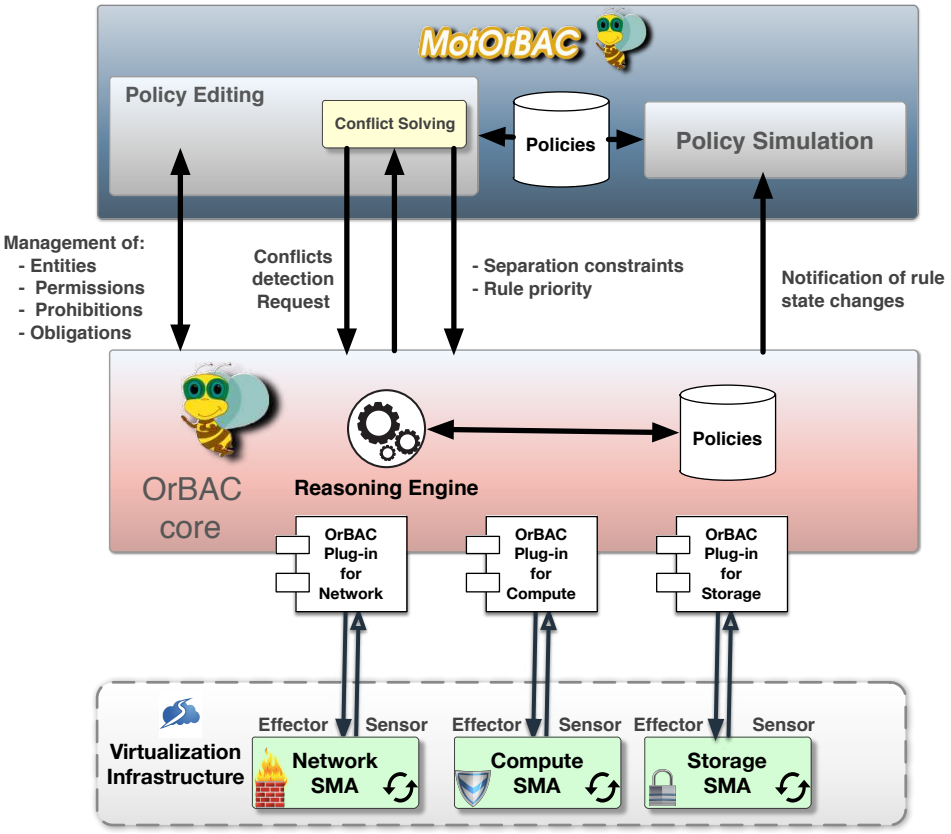

Figure 3.10: MotOrBAC tool architecture

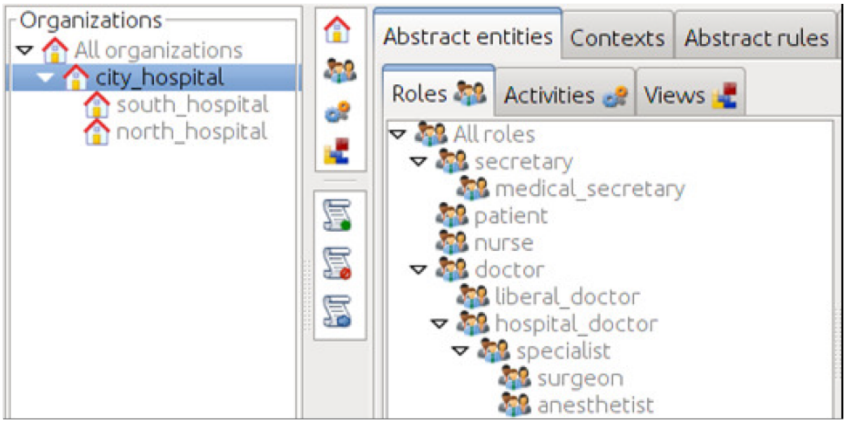

Figure 3.11: Graphical hierarchy representation in MotOrBAC

- **Policy simulation:** after having specified concrete entities (subjects, actions and objects), the concrete policy can be inferred as shown in Figure 3.12. Subjects, actions and objects can have attributes. The simulation interface is presented in Figure 3.13.

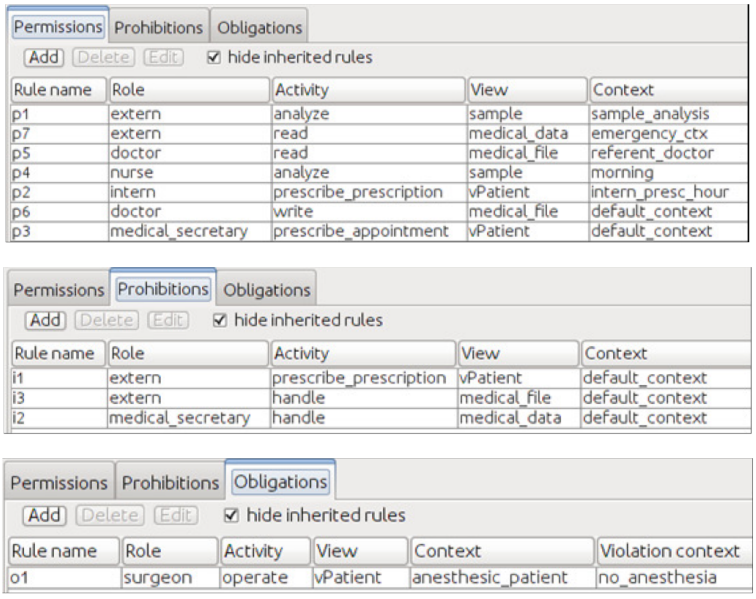

Figure 3.12: Example of abstract rules

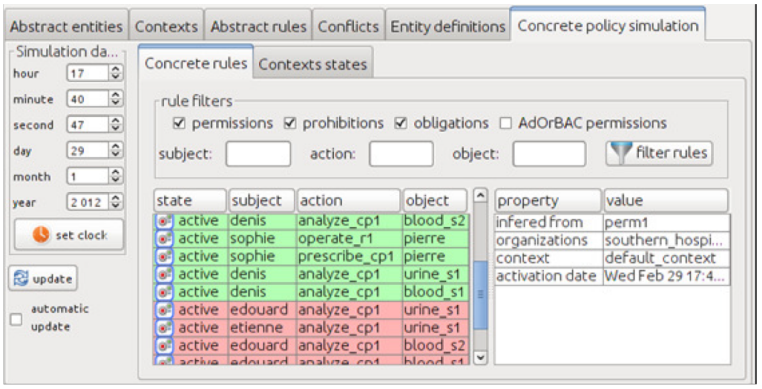

Figure 3.13: Simulation of concrete OrBAC policy

- **Policy consistency verification:** abstract conflicts between abstract rules can be detected. Once abstract conflicts have been detected, MotOrBAC is able to suggest the administrator some solutions to solve them. Figure 3.14 presents an interface of possible managements of these conflicts.

- **Administrative rights management:** the administrative rights of a subject or a role can be specified in order to decentralize the policy administration. MotOrBAC is able to express the administration policy using the same formalism. It can be used to specify the administration policy, each AdOrBAC policy associated to each policy. AdOrBAC model implemented in API OrBAC allows a decentralized administration of the policy. We can for example control creating abstract entities, creating rules, etc. Figure 3.15 represents the definition of a view by specifying the insertion of objects in the *use_teaching_resource_assignment_view* sub-view of the view *activity_assignment_view*.

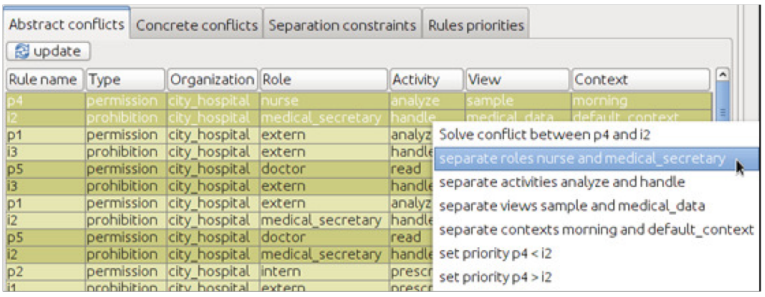

Figure 3.14: Conflict management interface

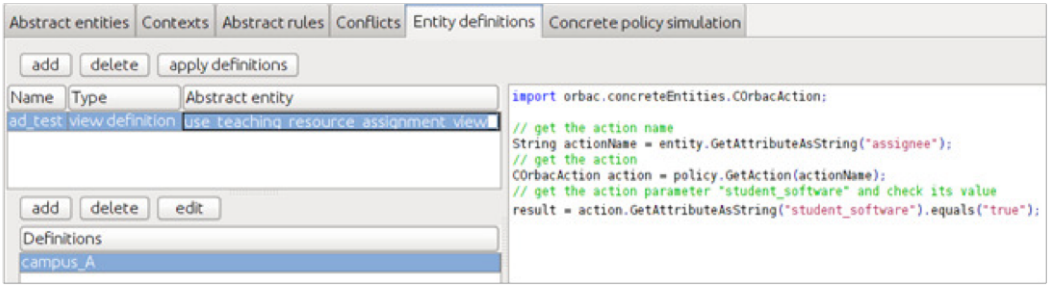

Figure 3.15: View definition

- **Delegation Mechanism:** with MotOrBAC model two types of delegation are possible. It is possible either to define delegation of some rule to someone without giving him a specific role or it can be delegated the role itself, which implies that the whole set of rules belonging to that role will be delegated. For example, considering the e-voting use case, it is supposed that the client can delegate cryptographic operations to a trusted server during voting process.
  - **Rule delegation:** is implemented in AdOrBAC by creating sub-views of the license delegation view. The policy subjects can delegate their rights to other users, by putting a constraint on the new sub-view and giving a role/subject the right to insert licenses into it as shown in Figure 3.16.
  - **Role delegation:** is implemented in AdOrBAC by creating sub-views of the role delegation view. The policy subjects can delegate their roles to other users, by putting a constraint on the new sub-view and giving a role/subject the right to insert role assignment objects into it. The approach of role delegation is very similar to rule delegation as shown in Figure 3.17.

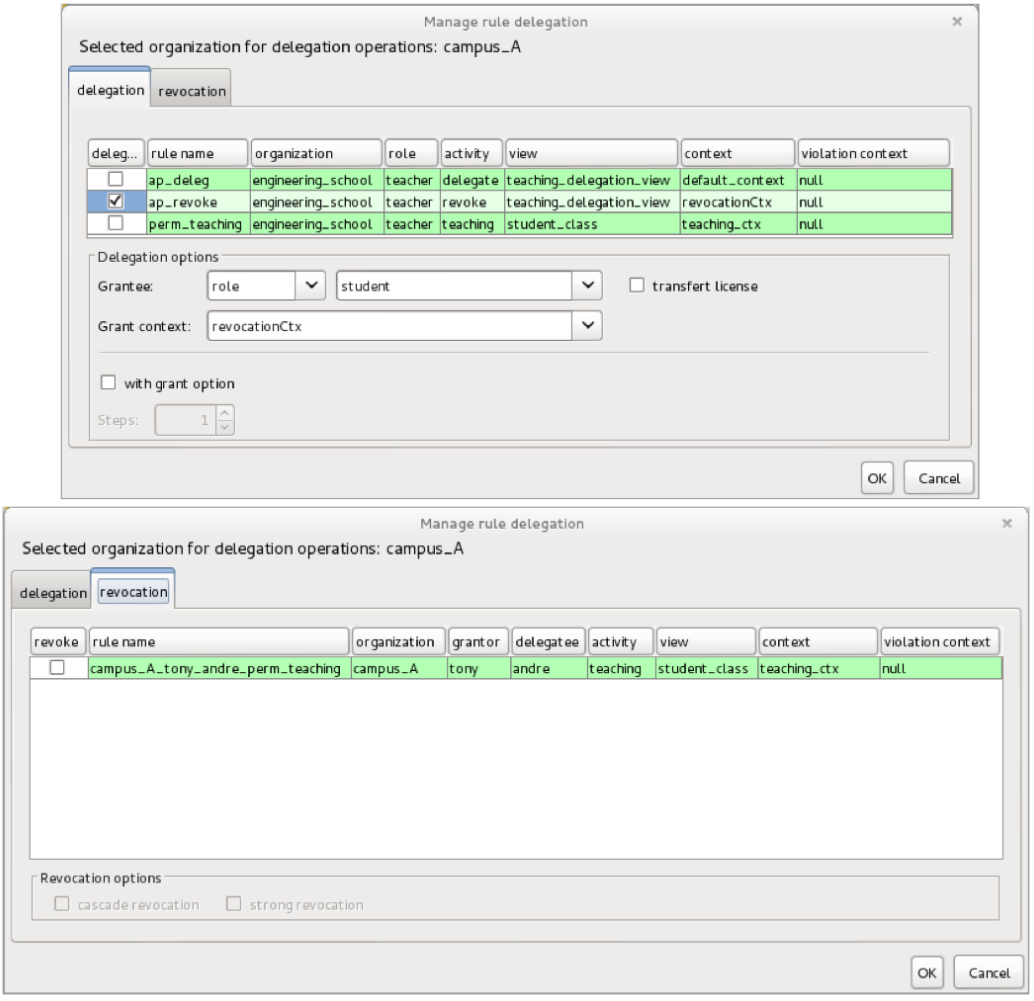

Figure 3.16: Rule delegation

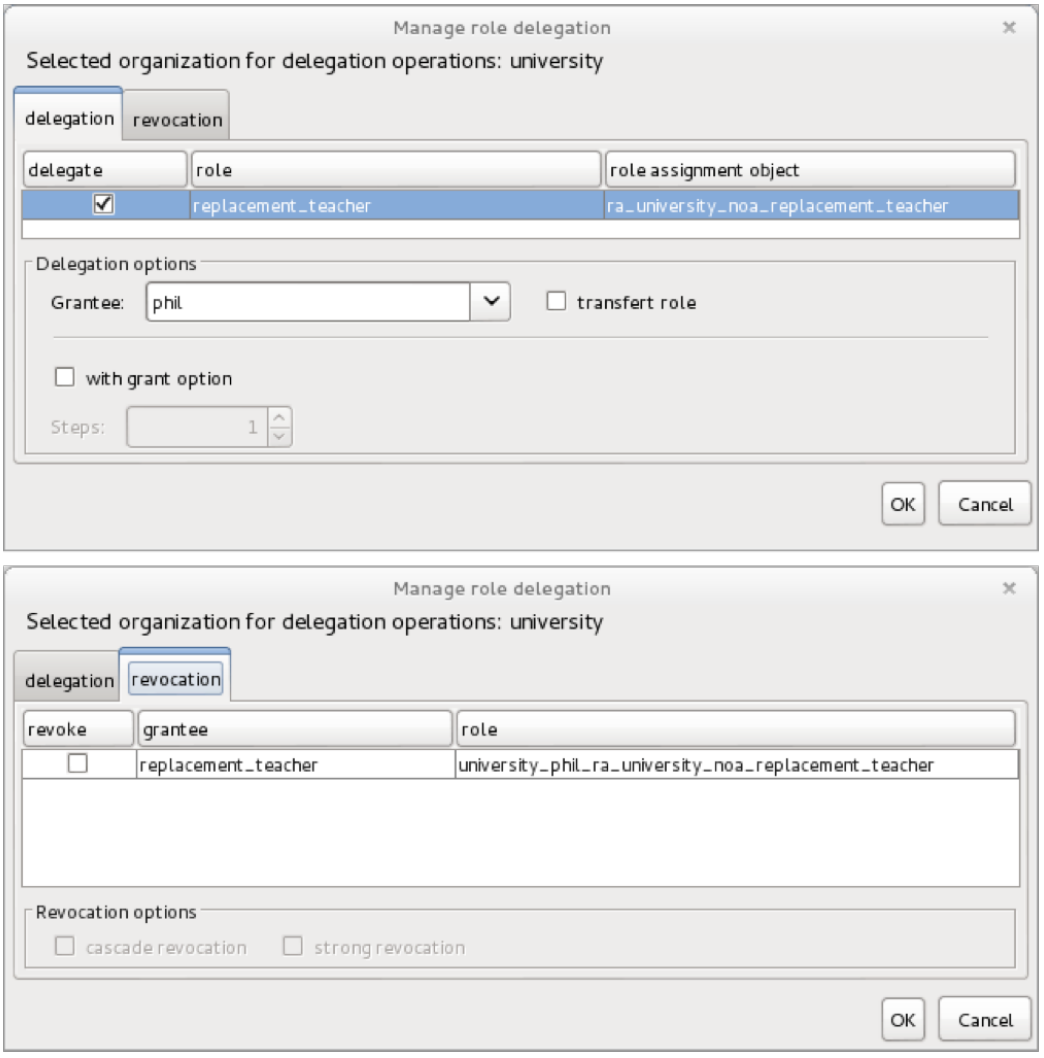

Figure 3.17: Role Delegation

3.3 Security Policy Engine

3.3.1 Policy Decision Point

The security policy engine module represents the Policy Decision Point, we call it as OrBAC-PDP, of the SUPERCLOUD architecture. It is the module that decides about the security rules to be enforced. This module is invoked by the various Policy Enforcement Points (PEP)s when we are considering the access control and it notifies PEPs when some usage control rules (obligations) are activated. It allows to take a decision (allow/deny) according to specific actions to be applied within the SUPERCLOUD framework and also trigger obligations for example in the case of usage control rule enforcement or in the case of any security flaw detection. In SUPERCLOUD, the PDP might be located at different locations than the PEP such as at Network later or at Application level. The PDP makes the decisions about which entities are permitted to access the resources, based on the policy that is set for accessing that resource. More specifically, once security policies are loaded in the OrBAC-PDP, the next step is to dynamically control the enforcement of these security policies based on the changed occurring in the environment. Whenever the PDP receives a request from the PEP, it checks it against the corresponding policies defined in the PAP and comes up to a decision. After making the decision it sends the reply back to the PEP. In order to provide access control and usage control for different levels of granularity, we are considering multiple decision points in our architecture. Thus, we categories OrBAC-PDP in two categories:

3.3.1.1 Autonomous PDP

The OrBAC-PDP usually acts autonomously in the domain where it is assigned. It makes decisions on all authorization requests that it understands, which means, on all queries for which it has respective policy available.

3.3.1.2 Distributed PDP

The OrBAC-PDP may additionally delegate decisions to further PDPs that are more specialized. However, for interacting with other PDPs, we need to establish and verify their trust relationship and communicate with each other at least using an authentic channel.

3.3.2 Policy Enforcement Point

The policy enforcement point (PEP) is the point that receives the request from the requester for accessing that resource and enforces the access control and usage control decision.

3.3.2.1 Forward PEP

If we talk about a Forward PEP, the PEP simply acts as decision forwarder and hence forwards any decision requests to OrBAC-PDP and returns PDPs answer to the local caller. This is the PEP default implementation, which may have some temporal and/or contextual limitations.

3.3.2.2 Autonomous PEP

If we talk about Autonomous PEP, the PEP acts as ancillary PDP based on local or distributed security policy decision database and security context information (general and/or promptly only) that are available at the respective module or architecture layer implementing a PEP. The authorization for local autonomous decisions is configured by PDP.
By default, PEP actively queries the PDM for every decision. However, in case the local PEP (i) has some local security context information PDP does not have available or does not understand and/or (ii) has very many time-critical decision requests than PEP could also come to decisions of its own based on a policy database, which of course is pre-configured by PDP.

3.3.3 Security Policy Enforcement Modes

In the SUPERCLOUD architecture, due to the dynamic changes in the contextual information the set of active/inactive policies needs to be synchronized with the set of active/inactive rules which are already enforced by the PEPs. For this purpose, we have defined an enforcement approach based on the concept of push and pull modes as shown in Figure 3.18, depending on the type of the security rules that we are considering.

3.3.3.1 Pull Enforcement

is a type of mode in which SDN controller invokes OrBAC-PDP to retrieve all the active rules related to permissions and prohibitions. The PEP calls, using the variables <subject, action, object>, the API method Is_permitted in order to evaluate the access.

3.3.3.2 Push Enforcement

is a type of mode in which, instead of PEP, the OrBAC-PDP pushes the changes, that occur in the security policy, to the PEP in the form of notifications to activate or deactivate different rules.

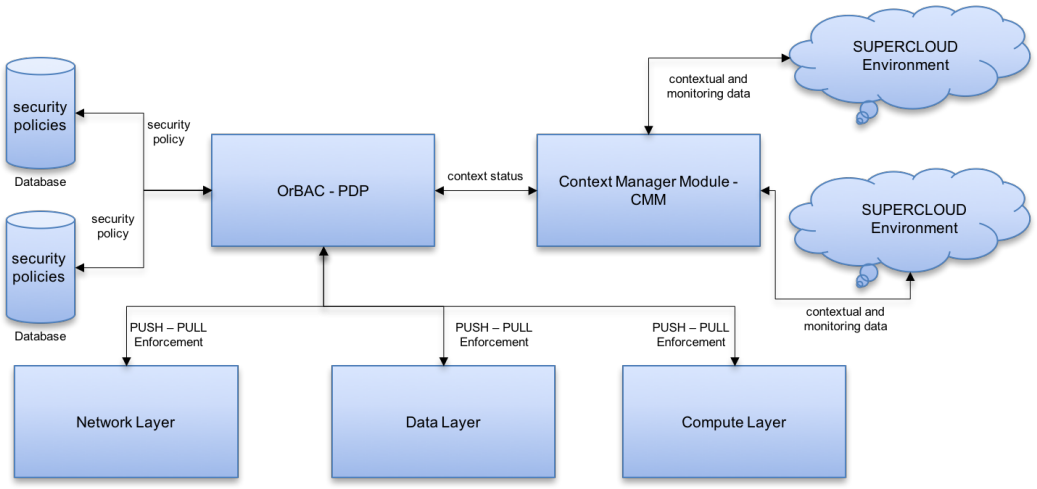

Figure 3.18: Security policy enforcement modes

Our core purpose to define push mode is to control obligation enforcement during the application execution. As discussed in the previous section 3.2, these obligations can have different states (Activation, Deactivation, Violation, Fulfillment), during the whole lifecycle of the application execution. In our approach, we enforce these different changes of obligation through considering and evaluating different states of the context. For this purpose, the context and monitoring modules are actively monitoring the changes happening in the environment and regularly notify the status of context to the OrBAC-PDP. Once the OrBAC-PDP has been notified that a context is in a new state, an analysis phase takes place in the OrBAC-PDP to check if there is any obligation that is activate, deactivate, fulfilled or obligation is violated.

Chapter 4 Use Case Prototypes

This chapter describes the Use Case Prototypes that demonstrate the use of SUPERCLOUD capabilities in realizing actual SUPERCLOUD services. Two prototypes are presented, demonstrating approaches for enforcing geolocation requirements for SUPERCLOUD services – Authenticated Discovery Prototype described in Sect. 4.1, and covering the realization of Network Function Virtualization (NFV) services on the SUPERCLOUD architecture – NFV Use Case Prototype discussed in Sect. 4.2.

4.1 Authenticated discovery for geolocation-restricted data replication

4.1.1 Related work

In this section, we analyze the state of the art related with the topic of data replication in VMs that are placed in allowed geographical places. This state of the art consists of the topics of geolocation, authenticated key agreement and grid resource discovery.

4.1.1.1 Geolocation

Geolocation of hosts on the Internet is currently achieved through a variety of evidence-gathering practices, including mining data from "who is" databases and DNS records, using Internet topology data, network metrics. The IP-based solutions alone provide geolocation with the accuracy of country reasonably reliably, but up to cities unreliably [44]. Example of IP geolocation services include IP2Location[1] and IPInfoDB[2]. Many more similar services are available paid or free. Other geolocation solutions use metrics of the network connections like round-trip times, route (made of successive hosts, which are named hops) and transit delays of packets across the Internet Protocol (IP) network. Landa et. al [56] proposed a model for the analysis of Internet round trip time (RTT) and its relationship to geolocation distance. However, this solution relies on a database and it does not take in consideration the live nature of the global networks, possibly making this solution unable to cope with network changes. In particular, Internet delays are known to violate the triangle inequality [44]. Katz-Bassett et. al [53] present Topology-based Geolocation (TBG), an approach which estimates the geographic location of arbitrary Internet hosts. The network round-trip delay is used as part of this solution. The proposed approach is leveraging network topology, along with measurements of network delay, to constrain host position. Furthermore, Huffaker et al. [45] make a comparison between the three source of geolocation knowledge (database, delay, and topology) and the conclusion is that although inconsistencies exist, aggregated results could prove useful for geolocation.
One other solution for addressing geolocation requirements is the use of service level agreements (SLAs) that specify the geographic region of a service. Reliance on a contractual obligation, however, may fail to detect misbehavior (malicious or accidental) on the part of the service provider. For example, a careless service provider may move client data to an overseas data center, for cheaper IT costs. Benson et al. [27] and Peterson et al. [70] independently proposed using proofs of data possession and host geolocation to bind cloud data to a specific geographic location. Extending this work, Gondree et

[1] http://www.ip2location.com/
[2] http://www.ipinfodb.com/

al. [44] proposed constraint-based data geolocation (CBDG), a data geolocation solution that builds on constraint based techniques for host geolocation. This methodology is generic enough to use any distance-latency model, including topology-aware models and semi-trusted landmarks at known locations. Bartock et al. [25] proposes to tackle the problem of geolocation using hardware root of trust, which is a combination of hardware and firmware that maintains the integrity of the geolocation information. The solution uses a hypervisor configuration. The main disadvantage of such solutions is that they require specific hardware or that require heavy integration efforts in multi-cloud systems.

4.1.1.2 Grid resource discovery

Resource discovery is a major challenge in multi-cloud environments. Most of the resource discovery existing techniques utilize a centralized mechanism, where each cloud interacts with a central entity or a meta-broker. However, a centralized approach to resource management and discovery does suffer from shortcomings like performance vs. scalability, security vulnerabilities and single-point-of-failure [52]. Some instances of decentralized resource discovery are available in grid computing (e.g. InterGrid [38]). Kocak and Lacks [55] propose to place the load of managing the network resource discovery inside of the routers. In the proposed protocol, the routers contain tables for resources similar to routing tables. These resource tables map IP addresses to the available computing resource values, which are provided through a scoring mechanism. This solution needs to be integrated in routers and is requires all the routers to adhere to the solution, which could prove difficult for integration in a multi-cloud scenario. Wright et al. [82] propose a software abstraction layer for resource discovery by applying a two-phase constraints-based approach to a multi-provider cloud environment. All these solutions insert latency in the system, which can hinder the performance.

4.1.1.3 Authenticated key agreement

A wide variety of cryptographic authentication schemes and protocols has been developed to provide authenticated key agreement to prevent man-in-the-middle attacks. These methods generally mathematically bind the agreed key to other agreed-upon data, such as public / private key pairs, shared secret keys or passwords. Such protocols are: STS (station-to-station[3]), SRP (secure remote password [83]), MQV (Menezes-Qu-Vanstone[4]) and AKEP2 (authenticated key exchange protocol 2). The STS, SRP and MQV protocols are based or similar with the Diffie-Hellman protocol and the STS and AKEP2 provide mutual entity authentication. These protocols are at least a three-pass protocol, which inserts latency in the data replication process. Furthermore, if one decides to use one of these protocols within a multi-cloud system, close integration needs to be done between sometimes maybe different implementations of the same protocol.

4.1.2 Proposed solution

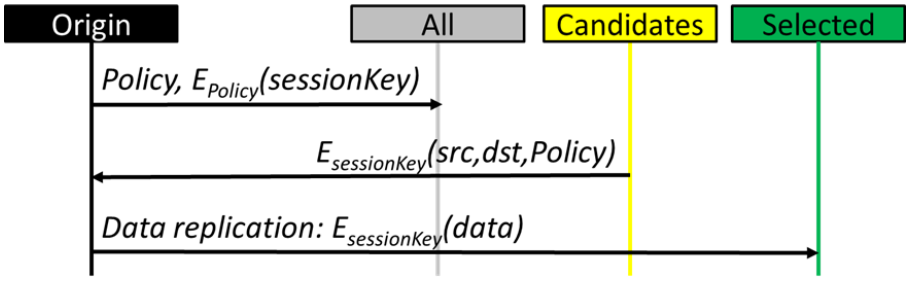

Figure 4.1: Authenticated discovery protocol

[3] https://en.wikipedia.org/wiki/Station-to-Station_protocol
[4] http://csrc.nist.gov/publications/nistpubs/800-56A/SP800-56A_Revision1_Mar08-2007.pdf

The proposed solution protocol, depicted in Figure 4.1, contains the following three steps:

- The origin broadcasts the message: $Policy, E_{Policy}(sessionKey)$ to "All":
  - The origin is the node that decides to replicate data because of the reasons previously specified in this document due to requirements regarding data query performance, load balancing and disaster recovery. In Figure 2 the origin node is depicted as a full-black circle.
- Targets, the rest of the nodes, receive the message broadcasted by the origin. They try to decrypt the broadcast message using their respective private keys and their reply is one of the two values:
  - *Do not understand*
    - They do not have the required keys to decrypt the message. This implicitly means that they do not satisfy the requested policy.
  - *OK, replication possible*, $E_{sessionKey}(src, dst, Policy)$. This answer is sent by the "Candidates".
    - They do have the required keys to decrypt the message. This implicitly means that they do satisfy the requested policy as asserted by the various authorities.
    - The target sends to the origin an ok message and a message encrypted with the session key to show that it is placed in an allowed geographical position (according to the policy).
- Origin selects one or more VMs where the data will be replicated, these being the "Selected" in the above figure. Next the origin sends the sensitive data, encrypted with the session key proposed in the beginning. Therefore the message sent to the replication VMs is: $E_{sessionKey}(DATA)$.

4.1.3 Architectural integration

This section describes the integration of the authenticated discovery protocol depicted in Figure 4.1 within a multi-cloud infrastructure (e.g. SUPERCLOUD, depicted in Figure 4.2). The integration consists of deploying in every VM of the U-cloud (connected healthcare platform) a component that is separately handling the data replication: "Data replication service". Such a component is lightweight and relies on being provisioned with a private key (when necessary). This is present in the compute plane of the U-cloud. The "Data replication service" triggers the authenticated discovery protocol and relies on the network plane of the U-cloud for broadcasting the discovery message (step 1). Next the same "Data replication service" component from candidates VMs is answering to the discover message (step 2). In step 3, the "Data replication service" sends the encrypted data to the selected VM for replication. This arrow (3) goes through the Data abstraction plane signifies that this plane needs to be aware of the replicated version of the data.
The previously described process is triggered when additional users (red "App"s) are using the App and therefore load balancing is needed for a better performance of the system.

4.1.3.1 Types of data

The solution presented in this chapter can cope with any type of data. The data that needs to be replicated should go through the following steps: packaging, encrypt package, transport package, decrypt package and finally un-packaging. For example for a database, the proposed solution would require a dump of the portion of the database that needs to be replicated (e.g. records and columns). Next this dump file is encrypted, sent via the proposed authenticated discovery protocol to the place where the data will be replicated. Then the received encrypted package is decrypted by the receiving replication VM.

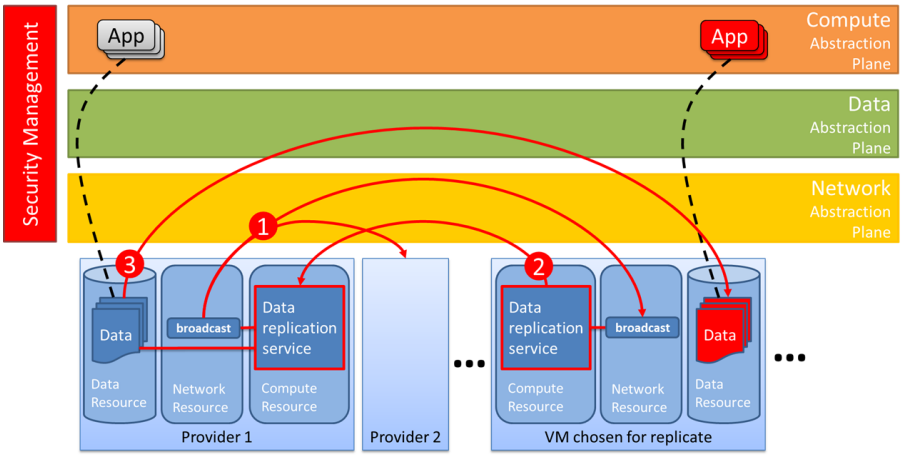

Figure 4.2: SUPERCLOUD architectural integration

Within the SUPERCLOUD architecture, a connected health system would be deployed as a U-cloud and therefore use the abstraction planes (compute, data, and network) as depicted in Figure 4.2. A U-cloud, as defined in the SUPERCLOUD deliverable D1.1 "User-centric management of security and dependability in clouds of clouds", is a user-specific ensemble of computational, data and networking services.[65] In Figure 4.2 the steps: 1, 2, 3 are map-able to the protocols steps described in Figure 4.1.

4.1.4 Validation

In this section we present the main features of the proposed multi-cloud authenticated discovery prototype, together with the problems, featured by the state of the art solutions, which this solutions is overcoming and a qualitative analysis about how this prototype affects the trust relationships between the components.

4.1.4.1 Performance

Our solution performs a fast replication of data by relating on a shorter and therefore faster protocol. The existent solutions which would comprise of three different protocol exchanges (discovery, authentication, key agreement) contains more steps and therefore are less efficient than the solution proposed in this paper. This can be observed easily in Figure 4.3. A default solution protocol would comprise of 9 steps.

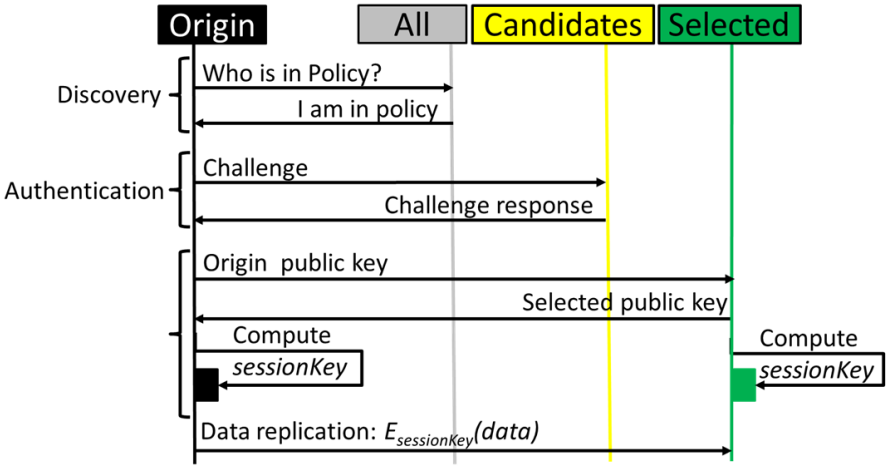

Figure 4.3: Default solution protocol

In comparison with the default solution protocol, our solution can achieve a better performance because it embeds discovery, authentication and key agreement in a single exchange, two-step protocol, which is more efficient. Our solution needs only one protocol exchange, and this one exchange only has to be finished by servers that satisfy the discovery broadcast. In our solution, the replication-VM is authenticating itself to the origin by being able to decrypt the discovery message that is sent by the origin. The decryption is based on getting a secret key from the geolocation attribute-based CAs. The attributes of a replication-VM do not necessarily need to match perfect the geolocation, since it is known that precise geolocation is difficult. A fuzzy approach can be leveraged and is explained in an extension.

4.1.4.2 Multi-cloud integration

Usually replication in multi-cloud system, appended with other local hospital servers, means integration between these solutions for enabling existent solutions like for example: TLS protocols. This problem is overcome by our proposed solution. The newly inserted platform-independent protocol integrates easily with heterogeneous (e.g. multi-cloud scenarios) systems because it does not rely on close integration between the computing providers. Our solution is based only on exchange of encrypted content and peer-to-peer connected VMs without relying on possible communications and specifics deployed for the multi cloud system. For example, the discovery phase relies on just sending an encrypted message, which is not platform dependent.

4.1.4.3 Flexibility

Our proposed solution comes up with a discovery process that is relevant to the current topology at the time of running the protocol, therefore automatically taking into account dynamic changes that occurred in the past. Furthermore, the proposed solution allows decentralized discovery of places (where data replication is allowed) without the need of having a central entity that orchestrates this discovery.

4.1.4.4 Trust management

The solution proposed in this document does not need to trust the cloud or even a possible multi-cloud deployment that the clouds/multi-cloud system will trustily enforce all the SLAs and use only the allowed geographical regions for replication. Furthermore, for the geolocation ABE authentication, the fuzzy authentication approach can split the trust between different CAs or semi-trusted landmarks that release secret keys for different geolocation measurement (e.g. ping, hops, etc.). The trust is dissipated even more by the usage of setting expiration dates on the secure tunnels created between an origin and a replication node. When a secure tunnel expires, a new key is negotiated. This way the security of the discovery process is enhanced. Furthermore, this expiration triggers a discovery process that might reveal better (e.g. closer) replication VMs.
We are moving the trust from a mesh of clouds, their security solutions, and integration of the security solutions, SLAs and collaboration SLAs to trusting a clear protocol. The simplicity and clarity of the protocol minimizes the attack surface. This moves the trust to the certification authorities and the key generation authorities.
sing a single CA that is fully trusted by all users and that reliably monitors user attributes is reasonable in small systems. However, for large and distributed systems, such as a connected health system, this is usually not the case. In the literature, Multi-Authority Key Generation Systems (MA-KGS) have been proposed to tackle this problem [32, 67, 57]. In these systems, the task of generating parts of a users secret keys relating to particular attributes is performed by so-called Key Generation Authorities, or KGAs. Apart from a system-wide public key generated by a Certification Authority (CA), each Key Generation Authority generates attribute public keys for each of its attributes. The user requests secret key parts from each KGA for (a subset of) the attributes it is responsible for. Therefore, a malicious KGA can issue secret keys for limited number of attributes. However, if the key material of

a KGA is compromised, then this still poses a risk because this material can then be combined with other user secret keys to obtain access to material that otherwise would not be accessible. [71]
For reducing the risk that the key material of the KGA is compromised, thereby reducing the level of trust in KGAs that is needed, we propose to use the technique of Multi-Authority Key Generation System. In such a system (exampled depicted in Figure 4) the user must receive secret keys from multiple KGAs in order to be able to decrypt the discovery message. These pieces of the secret key are associated with different subsets of geolocation attributes (e.g. ping time, hops number, IP address, DNS). This fits also with the fuzzy approach mentioned above. A secure distributed key generation solution [71] could also be leveraged for provisioning of the replication machines with the secret keys (depicted in red in Figure 4.4).

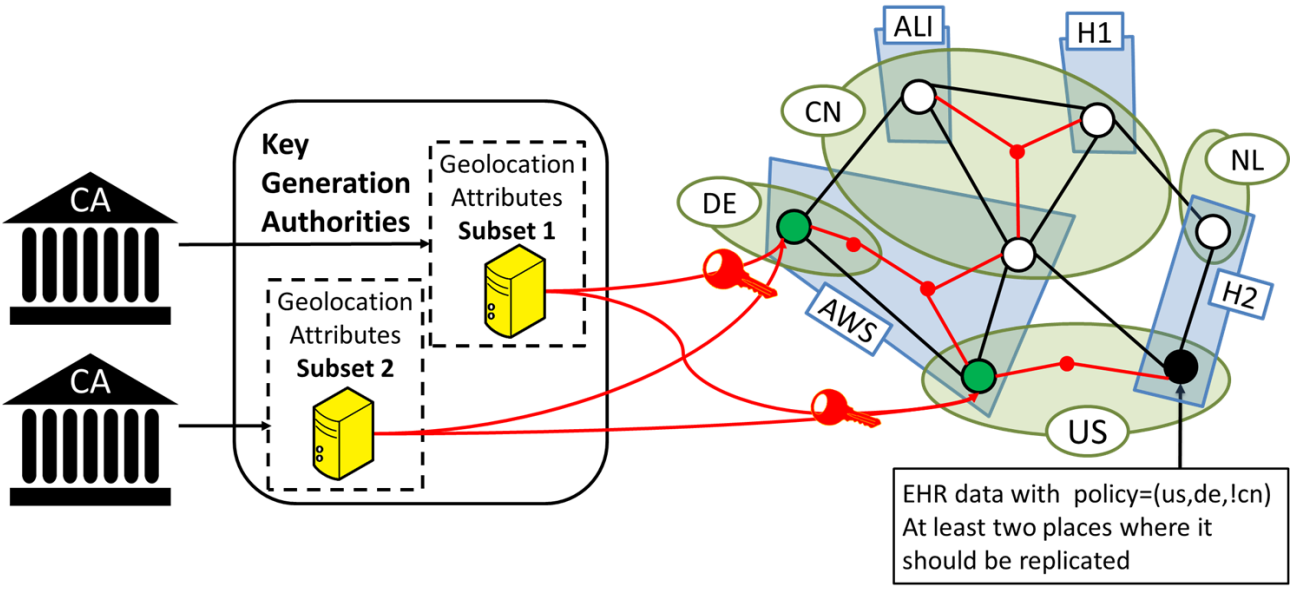

Figure 4.4: Authenticated discovery protocol using a Multi-Authority Key Generation System

4.1.5 Conclusions

This prototype can be used when replication of data is needed and when this data should be replicated only in allowed locations. The proposed solution can be generalized for just authenticated discovery, since it allows agreeing on a key by sending the encrypted key to the place where the data needs to be replicated. The encryption can be done using attributes that are different from geolocation and fuzzy authentication can be leveraged. The proposed solution is well suited for systems as the one that the SUPERCLOUD project is building because it does not rely on developing new integrated solution between the computing providers (e.g. cloud providers, hospital server), but only on a cryptographic protocols which are platform independent.
From a trust point of view the solution presented in this section represents an alternative to trusting the clouds with enforcing the SLAs and use only the allowed geographical regions for replication. Furthermore the geolocation ABE authentication can split the trust between different CAs or semi-trusted landmarks that release secret keys for different geolocation measurements (e.g. ping, hops, etc.). The proposed solution moving the trust from a mesh of clouds, their security solutions, and integration of the security solutions, SLAs and collaboration SLAs to trusting a clear and easy to audit protocol. The simplicity and clarity of the protocol minimizes the attack surface therefore enhancing the trust in the fact that the data will be stored only in allowed geographical location.

4.2 NFV use case prototype

This prototype, part of OPNFV [68], illustrates an NFV-oriented use case based on the SUPER-CLOUD architecture. It demonstrates security architecture and services for NFV applications such as federation of identity and authorization, and cross-provider security policy management (using the Moon framework [69], see Figure 4.5), network isolation, etc. Possible extensions include attribute-based encryption for data management. The underlying infrastructure should be OpenStack and OpenDayLight-based.

4.2.1 Use case

The considered use case features a Cloud Service Provider (CSP) proposing a cloud services solution to its customer: it provides networks, storage and computing resources to enable them to create their own cloud virtual machines and software containers. The CSP provides several distributed infrastructures to enable its clients to create their own cloud resources according to their geographic preferences (e.g., for improving the availability of a service to one area or for mirroring). In this context, *multi-tenancy* refers to the use of the same infrastructure by multiple but independent customers, and *multi-cloud* refers to the fact that various cloud infrastructures collaborate together to meet customer requirements. Since different types of resource should be protected and these resources can be dynamically created, removed or scaled up/down, the corresponding protection should also adapt to this dynamic aspect.
For our use case, users or applications create VMs, data stores and virtualized networks across the distributed infrastructure. For the protection of VMs, an access control mechanism enables to restrict manipulations on VMs. For storage, an attributed-based encryption mechanism is used. For virtualized networks, a firewalling mechanism protects each created sub-network. Finally, diverse resources are pooled into tenants which will be coordinated by security policies in order to avoid misconfiguration among security mechanisms. A user defines one security policy for each tenant, and this policy is enforced through different security mechanisms as discussed previously.
During execution, the security policy may be modified. Corresponding enforcement mechanisms are then able to adapt to this modification dynamically. New security mechanisms may also be inserted, and be self-configured through the security policy.

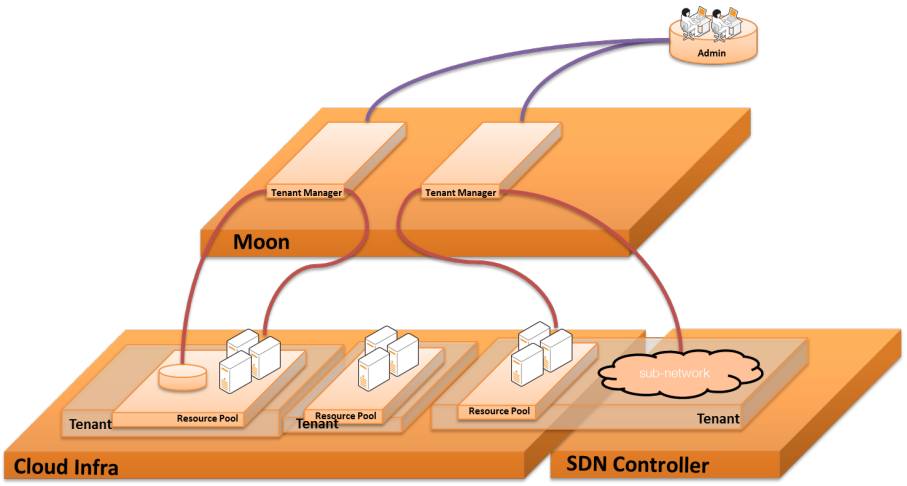

Figure 4.5: Moon authorization system

4.2.2 Prototype architecture

From implementation point of view, our prototype will apply a micro-service architecture. All security mechanisms will be implemented in containers (managers in Figure 4.6). This approach enables easy activation, configuration and update. Different security protection mechanisms can then be integrated through a well-defined northbound security management interface. Elementary enforcement agents (PEP in the Figure) will be deployed through the whole infrastructure, and controlled through a southbound security management interface.

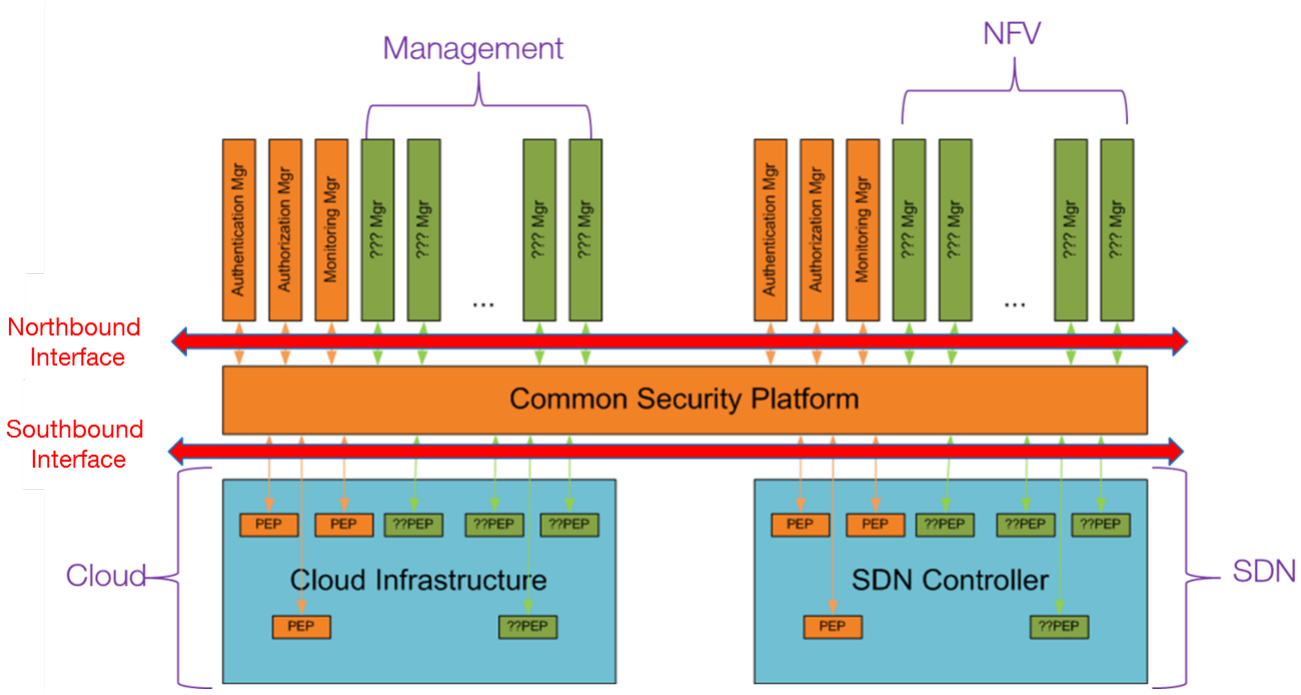

Figure 4.6: Prototype preliminary architecture

Chapter 5 Technology Demonstrator Prototypes

This chapter describes the Technology Demonstration Prototypes that explore dedicated novel technology solutions which can be integrated into the overall SUPERCLOUD architecture to provide new or improved computational capabilities for realizing SUPERCLOUD services or platform features. Two prototypes are presented, covering low-level isolation enabled by new hardware-security technologies like Intel SGX – Computation Environment Isolation Prototype described in Sect. 5.1 – and FPGA-based hardware mechanisms for optimized computations of virtual machines in user clouds – Cloud FPGA Prototype presented in Sect. 5.2.

5.1 Computation Environment Isolation Prototype

SUPERCLOUD is intended to be a self-managed system which requires a minimal amount of administrative efforts. One of the requirements to achieve this goal is the property of self-managed security. As described under [D1.1 4.3.1.2], in the concept of SUPERCLOUD the self-e of security is based on particular security services including *isolation*.
Intel Software Guard Extensions (SGX), introduced with Intel's latest Skylake processor generation, is a promising technology which can be used to isolate the processes of different users from each other. SGX refers to extensions to the processor's instruction set architecture. The aim of these extensions is to enable developers to integrate a protected execution environment, called enclave, in their software. Protecting an enclave is performed by the processor due to a memory protection system. Further, the integrity of an enclave can be proven upon an attestation scheme. Figure 5.1 shows two possibilities to implement the isolation service using the SGX technology.

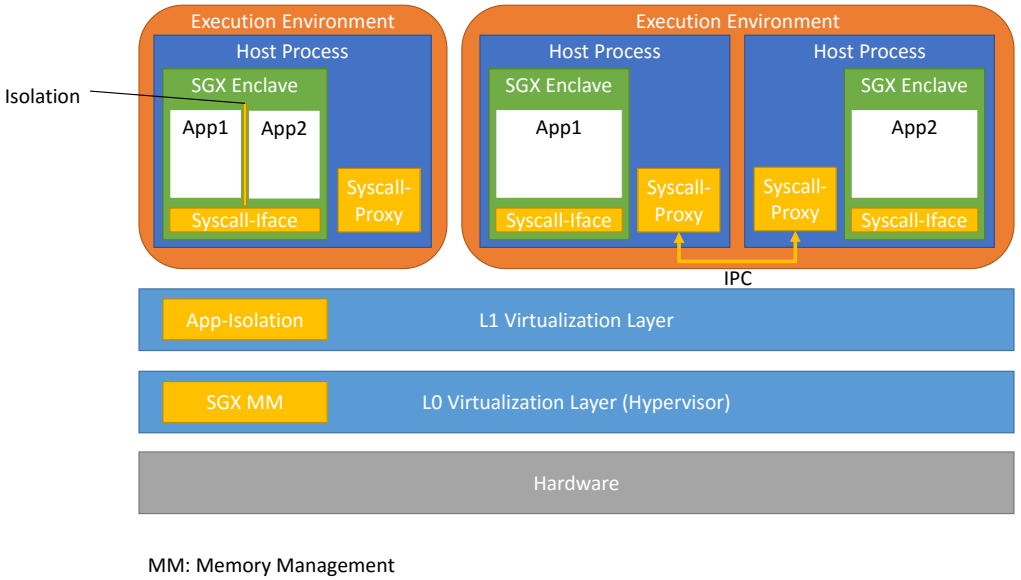

Figure 5.1: SUPERCLOUD Computation Environment Isolation Concept

Both variants are build on top of the *virtualization layer*. In the design depicted by the upper left part of the diagram, more than one application can be executed in an enclave whereas in the other design each application runs in a separate enclave. Since Intel SGX introduces a new execution and protection model, applications which are not developed for this technology can not be executed in an enclave without further ado.

5.1.1 Technical Challenges

There are three main aspects that are important to support the execution of legacy applications[1] within an enclave: *Operating system services*, *virtual memory management*, and *inter-process communication*.

5.1.1.1 Operating System Services

With regards to operating system services, the challenge arises from the fact that privileged execution, i.e., supervisor mode instructions, cannot be executed when the processor is in enclave mode. Therefore, the standard system-process interaction via system calls is not possible from within an application running inside an enclave. This interaction includes input, output, memory management, file system, and networking functionality. Usually, an application would issue the SYSENTER or SYSCALL instructions along with the necessary parameters, which would cause the CPU to transition to supervisor mode and trap into the kernel. Upon finishing the requested service the system would then return to user mode and continue executing the application.

5.1.1.2 Virtual Memory Management

Virtual memory isolation poses another challenges when dealing with code executing inside an enclave. In particular, individual processes on a regular system are isolated from each other through different virtual address spaces. This means that the same virtual address might be mapped to a different physical address from within two separate processes. This indirection ensures that memory that is not shared between two processes is exclusive to the respective process and cannot be accessed by the other one. Usually, enclave memory is assigned to a process on a per-page basis. The enclave page becomes part of the address space of the host process. This is impractical in the scenario of executing real world applications within exclusively in enclave mode, because there is no application code running in the host process.

5.1.1.3 Inter-Process Communication (IPC)

The third technical challenge is represented by inter-process communication (IPC). It often is required that processes communicate with each other and with that share some sort of data. However, the established mechanisms for performing IPC are either based on operating system services which are not available in an enclave or violate the security assumptions of SGX.

5.1.2 Solution Alternatives

To address the above-mentioned challenges related to the isolation of computation environments within enclaves, there are various technical options for realizing a solution. In the following, we discuss these potential solution alternatives.

5.1.2.1 System Call Interface

While the issue with the system call interface could in theory be addressed through binary rewriting techniques similar to para-virtualization, this interface is standardized and implemented in practice not by the application itself, but by common system libraries, such as libc under Linux. Therefore,

[1] Applications designed without the Intel SGX development and execution model in mind.

implementing the system call interface via a proxy library represents an alternative approach towards addressing this issue.
In particular, we propose replacing the standard system call functionality within the C library that is usually linked to applications dynamically at runtime, with an SGX-aware proxy implementation, that forwards privileged execution to a non-SGX implementation within the host process. The non-SGX end of this system call proxy would need to retrieve the arguments, service the actual system service request, and return its results to the enclave.

5.1.2.2 Virtual Memory Management

There are two different possibilities for implementing the virtual memory management for enclave memory. The first possibility is to have maintain one single host process for all the running enclave applications. The benefit of this solution is that the memory management functionality can be implemented within this host process by claiming all possible SGX memory, and assigning it to the applications through standard memory allocation implementations.
The drawback of this possible implementation scenario is that the applications running in enclave mode would not be separated through virtual memory anymore, but become part of the same address space. With regards to the threat model, this would not represent a major problem though, because individual applications running in enclave mode have to be considered trusted already.
The second possible implementation strategy is to assign every application its own host process. The problem with this is that the host process would only be created to have enclave memory assigned to its virtual address space, but would not be executing any application code. The performance overhead introduced by this solution would therefore be expected to be higher than in the first solution.

5.1.2.3 Inter-Process Communication

The most straightforward way to realize IPC in the context of SGX is to execute applications enclaves which communicate with each other. To achieve this goal, the *local attestation* mechanism, introduced by the SGX technology, can be used to reliably establish the trust between enclaves. However, having a separate enclave per application could cause a considerable overhead for the overall system. This factor could motivate the implementation of a new IPC mechanism for processes being executed in parallel in the same enclave.

5.1.3 Prototype Realization

We decided to leverage an existing higher-level language, Python, to provide isolation between various components (Python modules), as shown in Figure 5.2. It is possible, then, to develop a number of different modules and deploy them in a Python interpreter that lives inside a single SGX enclave. Communication to this enclave (including deployment of further modules) is performed via TLS-encrypted sockets, while local storage of data is protected by an encrypted file system.
We decided to handle system calls using a custom version of an existing MIT-licensed libc implementation, *musl*. Instead of issuing the syscall directly, our version calls a proxy located outside the enclave, which forwards the parameters to the kernel and returns the return code to the enclave. When a parameter is a pointer to some other data (e.g., for the system call *write*) we take care of copying the data itself from or to the enclave.

5.2 CloudFPGA

In the SUPERCLOUD framework, CloudFPGA serves as an accelerator resource in the CSPs infrastructure, similarly to other compute, storage, and network resources. Since heterogeneous devices, such as FPGAs and GPUs are not yet common in public cloud offerings, at first, this may only be valid for private cloud service providers.

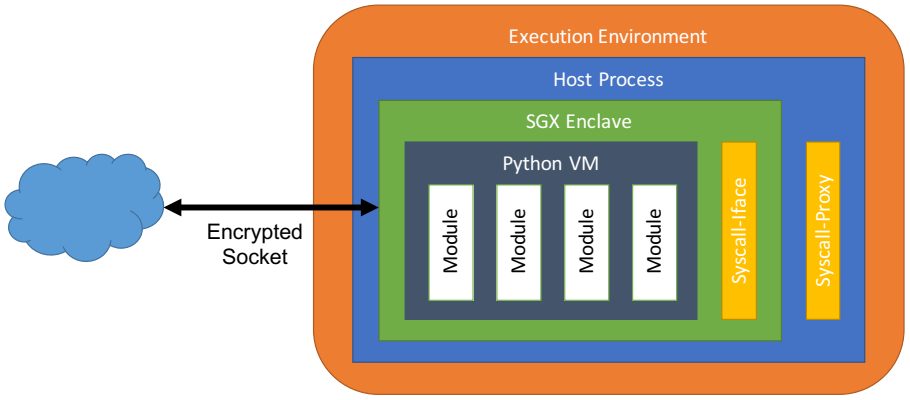

Figure 5.2: Python VM inside an SGX enclave.

5.2.1 Use Cases

Once the CloudFPGA becomes available in CSP infrastructure, the SUPERCLOUD framework can use it in two ways: (i) to accelerate workload processing in the abstract planes, and (ii) to accelerate SUPERCLOUD management tasks.

(a) Acceleration of Abstract Planes In the SUPERCLOUD framework, the CloudFPGA is vertically mapped across the three abstract planes as shown in Figure 5.3. CloudFPGA accelerates compute, data, and network processing in the corresponding abstract planes. In the compute abstract plane, user workloads are accelerated. For example, acceleration of medical record classification using text analytics in healthcare applications. In the data abstraction plane, the access to user data is accelerated. For example, using FPGA-based mem-cached appliances. In the network abstraction plane, various network virtualization functions can be accelerated. For example, by implementing VLAN tagging, packet re-writing, and tunneling (VXLAN) functions in FPGA.

(a) Acceleration of Management Functions CloudFPGA executes the management functions of the SUPERCLOUD platform. For example, security functions, such as data encryption, decryption, and monitoring can be executed in FPGA.

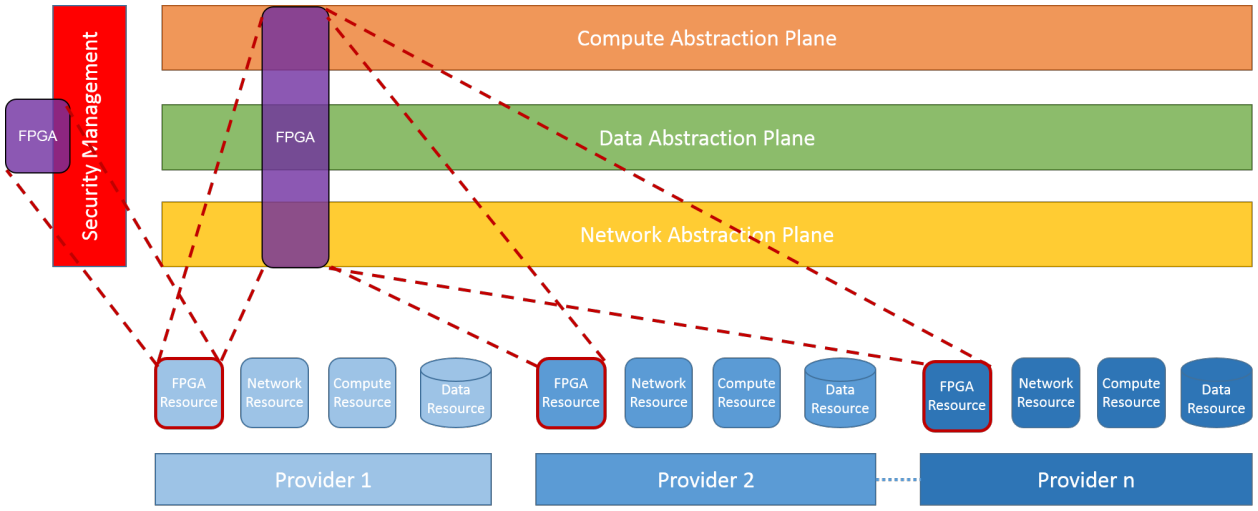

Figure 5.3: CloudFPGA in SUPERCLOUD

5.2.2 Technology Overview

FPGAs (Field Programmable Gate Arrays) are making their way into data centers (DCs) and are used to ofoad and accelerate specic service-oriented tasks, such as web-page ranking [73] , memory caching [29] , and high-frequency trading [59] . But these FPGAs are not yet available to general cloud users who want to get their own workload processing accelerated. This puts the cloud deployment of compute-intensive workloads at a disadvantage compared with on-site infrastructure installations, where the performance and energy efciency of FPGAs are increasingly being exploited. CloudFPGA solves this issue by offering FPGAs to the cloud users as an IaaS resource. Using the CloudFPGA system (Figure 5.4), cloud users can rent FPGAs, similarly to renting VMs in the cloud, and get their workload processing accelerated.

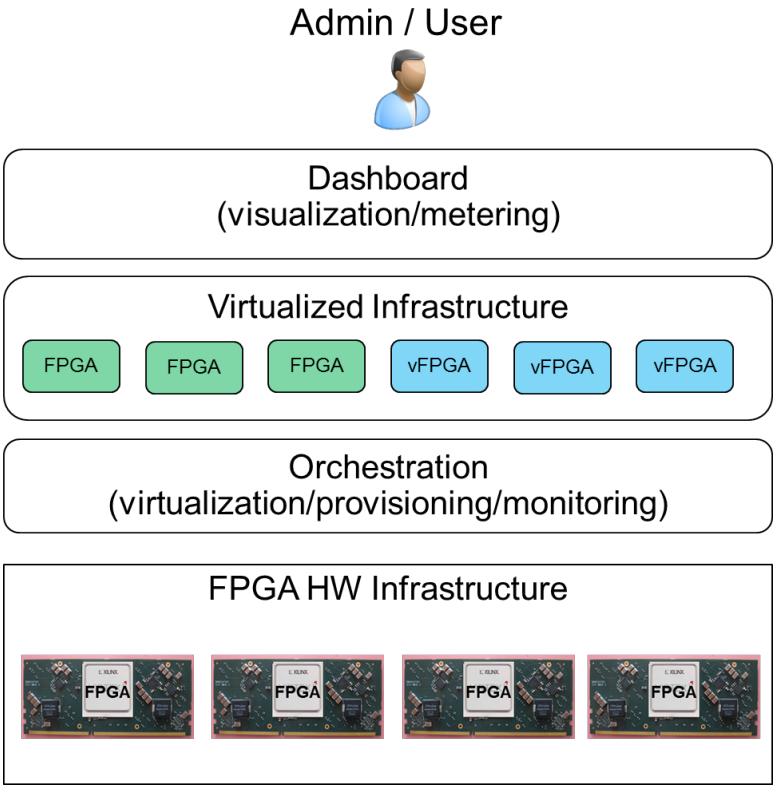

Figure 5.4: CloudFPGA System

5.2.3 System Architecture

The CloudFPGA system is built based on 3 main concepts: (i) standalone FPGA, (ii) hyperscale infrastructure, and (iii) OpenStack accelerator service. In this section, the CloudFPGA system architecture is explained based on those three concepts.

5.2.3.1 Standalone FPGA

The concept of standalone FPGA is built on two main initiatives: (i) changing the traditional way of attaching an FPGA to a CPU by moving from PCIe-attachment to network attachment, and (ii) making the FPGA self-managed without needing to have a host server for the management. This section explains the concept of standalone FPGA.

(a) Fundamental Shift from PCIe-Attachment to Network-Attachment

Mainly, there are three approaches for connecting an FPGA to a CPU. One option is to incorporate the FPGA onto the same board as the CPU when a tight or coherent memory coupling between the two

devices is desired (Figure 5.5-(a)). Such a close coupling is not expected to be generalized outside the scope of very specic applications, because, first, it breaks the homogeneity of the compute module in an environment where server homogeneity is sought to reduce the management overhead and provide exibility across compatible hardware platforms. Second, in large DCs, failed resources can be kept in place for months and years without being repaired or replaced, in what is often referred to as a fail-in-place strategy. Therefore, an FPGA will become unusable and its resources wasted if its host CPU fails.
The common approach for deploying FPGAs in a server is by tightly coupling one or two FPGAs to the CPU over the PCIe bus (Figure 5.5-(b)). However, this PCIe-attachment has two major issues in DC deployment. First, the power consumption of a server is order of magnitude higher than that of an FPGA. Hence, the power efciency that can be gained by ofoading tasks from the server to 1 or 2 FPGAs is very limited [42] . Second, in DCs, the workloads are heterogeneous and run at different scales. Therefore, the scalability and the exibility of the FPGA infrastructure are vital to meet the dynamic processing demands. With PCIe-attachment, a large number of FPGAs cannot be assigned to run a workload independently of the number of CPUs, and also those FPGAs cannot be connected in exible user-dened topologies. Some large-scale FPGA deployments [73] get around this issue of scalability and exibility to a certain extent by having a secondary dedicated network connecting multiple PCIe-attached FPGAs together. However, a dedicated secondary network breaks the homogeneity of the DC network, and increases the infrastructure management overhead.
The other approach for deploying FPGAs is by attaching them directly over the DC network (Figure 5.5-(c)), which signicantly improves the scalability and the exibility of the FPGA infrastructure compared with the PCIe-attachment. There are few previous attempts [2] [3] [6] ,which directly attach an FPGA to the network. Even though those attempts provide a network connection, the FPGA always remains physically attached, hosted and controlled by a dedicated server. Instead, the authors of [81] at IBM Research Zurich proposed the concept of standalone network-attached FPGA to completely disaggregate the FPGA resource from the server. This approach frees the FPGA from the traditional CPU-FPGA attachment and tightly couples the network and application processing in the same FPGA device. We believe that this is the key enabler for large-scale deployments of FPGAs in DCs.

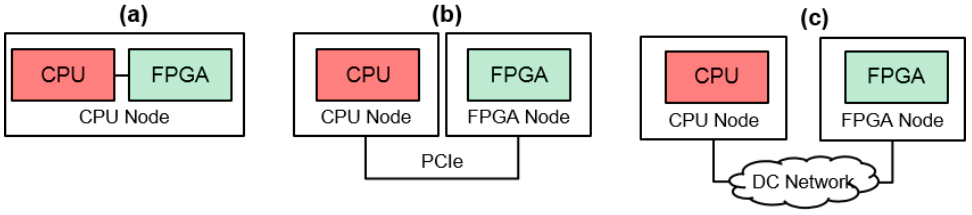

Figure 5.5: Approaches for Connecting an FPGA to a CPU

(b) Standalone Network-Attached FPGA

The high-level architecture of the standalone network-attached FPGA concept proposed in [81] is shown in Figure 5.6. It contains an FPGA and an optional off-chip memory. The FPGA is split into three main parts: i) a user logic part used for implementing customized applications, ii) a network service layer (NSL), which connects with the DC network, and iii) a management layer (ML) to run resource-management tasks.
User Application: One or more user applications can be hosted on a single physical FPGA (pFPGA), somehow similar to one or more VMs running on the same hypervisor. Each user gets a partition of the entire user logic and uses it to implement its applications. This partitioning is achieved by a feature called partial reconguration, a technology used to dynamically recongure a region of the FPGA while other regions are running untouched. We refer to such a partition of user logic as a virtual FPGA (vFPGA).
Network Service Layer (NSL): The NSL provides the network connection for vFPGAs to commu-

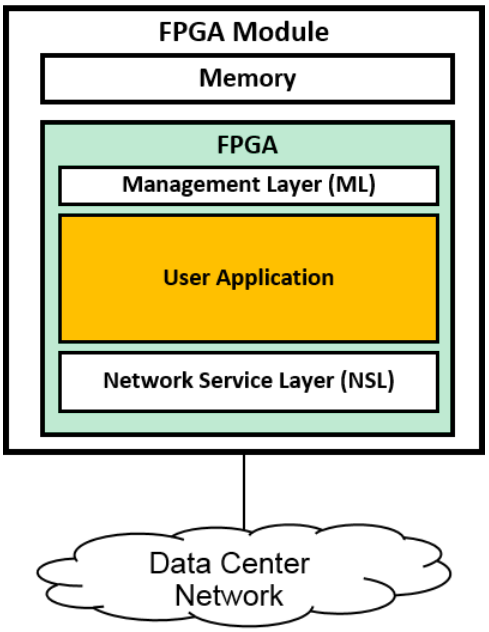

Figure 5.6: Standalone Network-Attached FPGA

nicate with servers and other vFPGAs over the DC network, similarly to the power service layer (PSL) of IBMs coherent accelerator processor interface (CAPI), which provides a PCIe-based communication link for its accelerator function units for communicating with SW applications. The NSL consists of (a) an application interface layer (AIL) and (b) a network protocol stack (NPS). The AIL executes two main tasks: (i) It serves as a switch that multiplexes and de-multiplexes incoming and outgoing data path and control path network payloads to and from vFPGAs. (ii) It offers multiple FIFO-based TCP and UDP network interfaces to the vFPGAs through the vFPGA IF. The AIL can have one or more vFPGA IFs, so that the standalone network-attached FPGA can run multiple applications simultaneously. The NPS contains a network interface and a TCP/IP protocol stack to connect the FPGA to the DC network.

Management Layer (ML): The management layer contains a memory manager and a management stack. The memory manager enables access to memory assigned to vFPGAs and the management stack enables the vFPGAs to be remotely managed by a centralized management software. The memory manager contains a memory controller and a virtualization layer. The memory controller provides the interface for accessing memory from the vFPGAs. The virtualization layer allows the physical memory to be partitioned and shared between different vFPGAs in the same device. This layer is congured through the management stack according to the vFPGA memory requirements. The management stack runs a set of agents to enable the centralized resource-management software (see section 5.2.3.3) to manage the FPGA remotely. The agents include functions such as device registration, network and memory conguration, FPGA reconguration, and a service to make the FPGA nodes discoverable.

(c) Potentials of Standalone FPGA The standalone FPGA introduced above can be used to build applications that need only one (Figure 5.7-(a)) or several independent FPGAs (Figure 5.7-(b)). For applications that need to setup a large number of FPGAs, the architecture scales to large multi-FPGA fabrics (Figure 5.7-(c) and (d)). These multi-FPGA fabrics are created on-demand in a software-defined manner, hence we call it Software-Defined Multi-FPGA Fabric (SDMFF). We have seen the success of large-scale SW-based distributed applications such as those based on MapReduce and deep learning [36]. To run those kind of applications in an energy-efficient manner at a high bandwidth and a low latency, these multi-FPGA fabrics provide a scalable reconfigurable compute fabric.

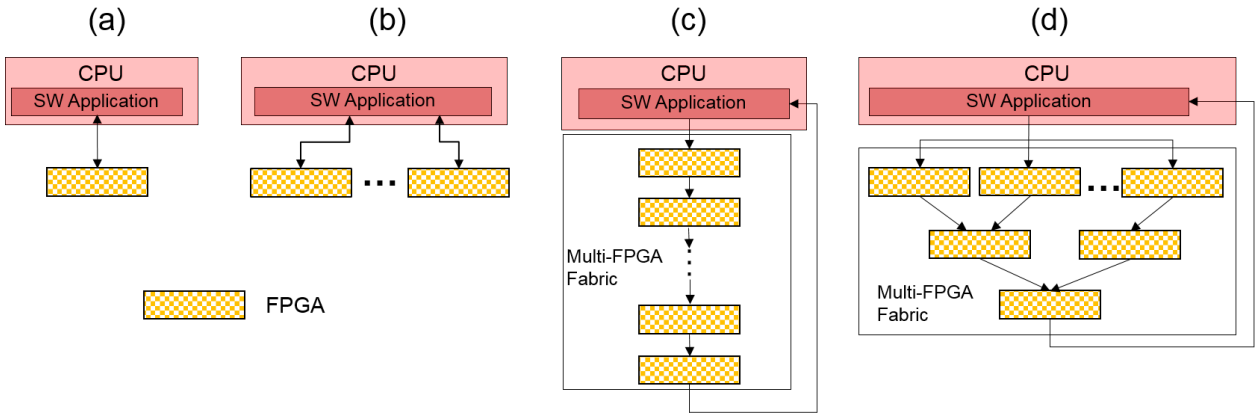

Figure 5.7: Use Cases of Standalone Network-Attached FPGA

5.2.3.2 Hyperscale Infrastructure

This section explains how the standalone FPGA concept introduced in Section 5.2.3.1 is mapped to real HW in DCs. In DCs, the straightforward way to build an FPGA cluster that consists of above-mentioned standalone FPGAs is using off-the-shelf HW. For example, such a cluster can be arranged by placing multiple off-the-shelf FPGA cards, which are connected to the DC network, in a PCIe-expansion chassis. By vertically placing multiple such chassis, an FPGA cluster with around 100 FPGAs per rack can be built. But, since those off-the-shelf HW are not purposely built for such a task, the scalability is poor in terms of the HW cost, the infrastructure cost and the physical space. The IBM hyperscale infrastructure proposed in [60] [81] solves this issue by redesigning the FPGA cards and the chassis base boards from the ground up. The next two subsections explain those two components: (i) hyperscale FPGA module and (ii) hyperscale base board.

(a) Hyperscale FPGA Module (FMKU2595) IBM hyperscale FPGA module (Figure 5.8) is called FMKU2595, which features a Xilinx KintexUltraScale FPGA with two independent DDR4 memory channels (816GB each). The FMKU2595 card is in the size of a double-height dual inline memory module (DIMM - 140 mm 62 mm). The card provides x8 PCIe Gen3, six 10G Ethernet links and two SATA interfaces. The card provides an extension connector that adds 128 Gbps of bandwidth over 8 lanes, and two I/O connectors for plugging an I/O mezzanine. For more information on FMKU2595, refer to the data sheet [48].

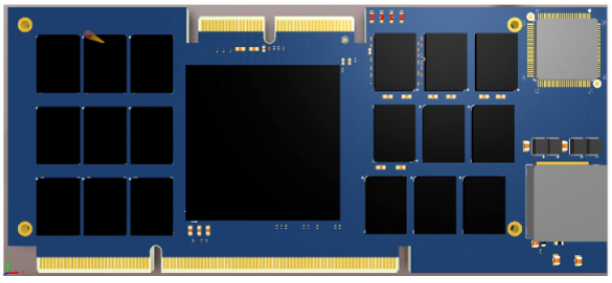

Figure 5.8: IBM Hyperscale FPGA Module (FMKU2595)

(b) Hyperscale Base Board (BB#2) IBM hyperscale base board (Figure 5.9) is a carrier SLED designed to host a cluster of above-mentioned FMKU2595 modules. Thirty two FMKU2595 modules plug into the SLED and are interconnected over 10GbE to the south side of an Intel FM6364 Ethernet switch (Figure 5.10), for a total of 320 Gb/s of aggregate bandwidth. The north side of the FM6364 switch connects to eight 40GbE uplinks, which expose the SLED to the data center network with another 320 Gb/s. This provides a uniform and balanced (no over-subscription) distribution between

the north and south links of the Ethernet switch, which is desirable when building large and scalable fat-tree topologies (a.k.a. folded Clos topology). Two SLEDs fit a 19" 2U liquid cooled chassis, and 16 chassis fit a rack. This amounts to 1,024 compute modules in a single rack which provides a total of 16 TB of DRAM and 2.7M Xilinx DSP slices. For more information on BB#2, refer to the data sheet [47].

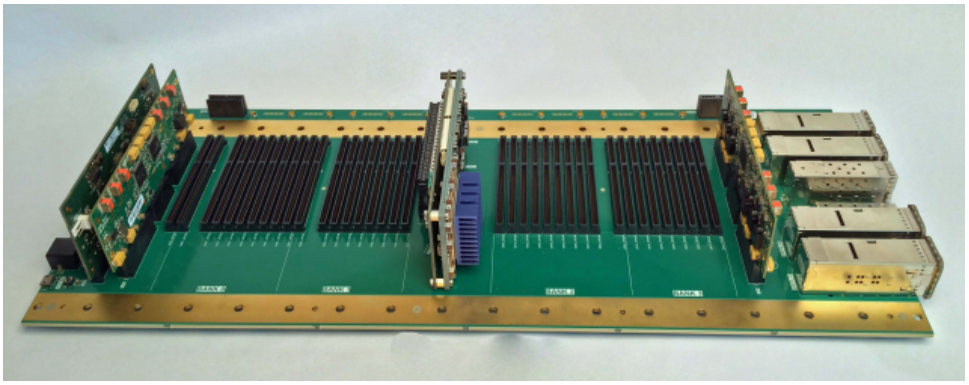

Figure 5.9: IBM Hyperscale Base Board (BB#2)

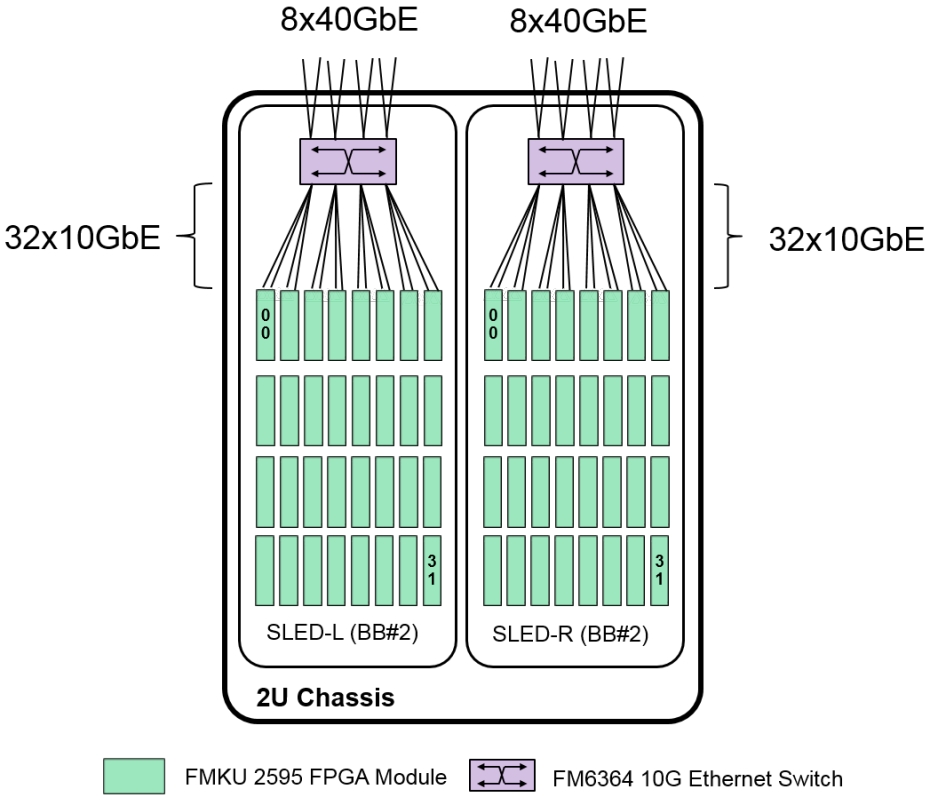

Figure 5.10: Arrangement of 64 FMKU2595 Modules in 2 BB#2 Base Boards. (Two BB#2 boards are hosted by a 2U chassis to build an FPGA cluster with 1024 FPGAs/Rack)

5.2.3.3 OpenStack Accelerator Service

Standard OpenStack does not provide services to integrate and provision heterogeneous devices such as FPGAs in the cloud. Therefore, we introduce and build a new service into OpenStack, which we

call accelerator service, to make the FPGA cluster explained in section 5.2.3.2 available in the cloud. This section explains the OpenStack accelerator service.
In previous work, FPGAs [30] [33] and GPUs [35] have been integrated into OpenStack by using the Nova compute service in OpenStack. In those cases, heterogeneous devices are PCIe-attached and are usually requested as an option with virtual machines or as a single appliance, which requires a few simple operations to make the device ready for use. In our deployment, in contrast, standalone FPGAs are requested independent of a host. Therefore, similar to Nova, Cinder and Neutron in OpenStack, which translate high-level service API calls into device-specic commands for compute, storage and network resources, we build the accelerator service shown in Figure 5.11, to integrate and provision FPGAs in the cloud. In the gure, the parts in red show the new extensions we build for OpenStack. To setup network connections with the standalone FPGAs, we need to carry out management tasks. For that, we use an SDN stack connected to the Neutron network service, and we call it the network manager. Here we explain the high-level functionality of the accelerator-service and the network-manager components.
Accelerator Service: The accelerator service comprises an API front end, a scheduler, a queue, a data base of FPGA resources (DB), and a worker. The API front end receives the accelerator service calls from the users through the OpenStack dashboard or through a command line interface, and dispatches them to the relevant components in the accelerator service. The DB contains the information on pFPGA resources. The scheduler matches the user-requested vFPGA to the user logic of a pFPGA by searching the information in the DB, and forwards the result to the worker. The worker executes four main tasks: i) registration of FPGA modules in the DB; ii) retrieving vFPGA bit streams from the Swift object store; iii) forwarding service calls to FPGA plug-ins, and iv) forwarding network management tasks to the network manager through the Neutron service. The queue is just there to pass service calls between the API front end, the scheduler and the worker. The FPGA plug-in translates the generic service calls received from the worker into device-specic commands and forwards them to the relevant FPGA devices.
Network Manager: The network manager is connected to the OpenStack Neutron service through a plug-in. The network manager has an API front end, a set of applications, a network topology discovery service, a virtualization layer, and an SDN controller. The API front end receives network service calls from the accelerator-worker through the Neutron and exposes applications running in the network manager. These applications include connection management, security and service level agreements (shown in red in the network manager in Figure 5.11). The virtualization layer provides a simplied view of the overall DC network, including FPGA devices, to the above applications. The SDN controller congures both the FPGAs and network switches according to the commands received by the applications through the virtualization layer.

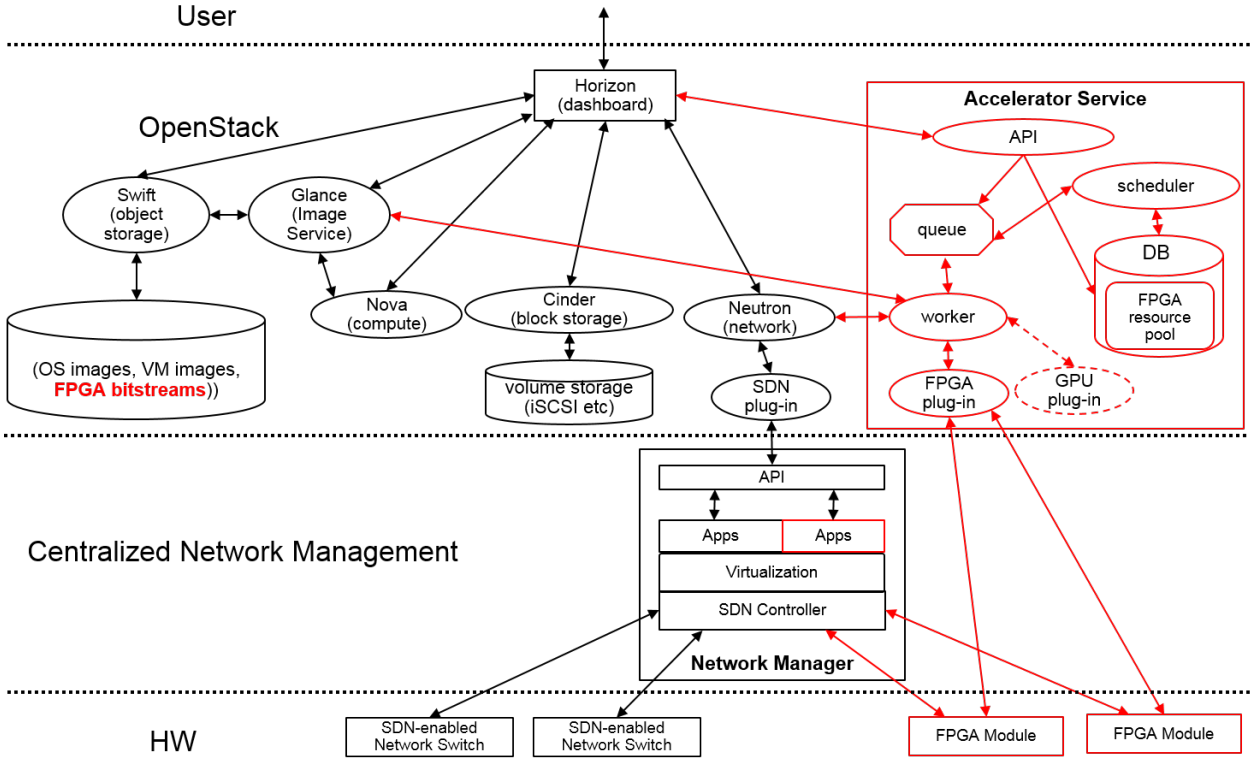

Figure 5.11: OpenStack Accelerator Service

Chapter 6 Summary and Conclusion

In this document we have described the technical implementations of components realizing the secure computation infrastructure and self-management of VM security in the SUPERCLOUD architecture. The presented prototypes address a broad spectrum of aspects necessary for deploying the architecture on a real computation infrastructure. These include the orchestration of computational environments deployed over the physical infrastructure of several different cloud providers as well as the management of trust across different layers of architectural abstractions. Also low-level aspects of computation environment isolation utilizing hardware mechanisms and enabling access to hardware-based computation services were discussed. In addition, two use case prototypes related to the management of the geolocation of services as well as realization of network function virtualization were introduced. The self-management of security in SUPERCLOUD was addressed through two prototypes related to security policy modelling and enforcement.
In future work, the technical insights related to these prototypes focusing on the computational aspects of SUPERCLOUD will be developed further and integrated with components of other sub-architecture layers of the overall SUPERCLOUD architecture. Through this prototyping-centric approach we envisage to create a solid technical basis for an integrated technical prototype architecture fulfilling the technical requirements set towards the SUPERCLOUD system in deliverable D1.1.

Chapter 7 List of Abbreviations

ACPI	Advanced Configuration and Power Interface
AHCI	Advanced Host Controller Interface
AIK	Attestation Identity Key
AIL	Application Interface Layer
AOP	Aspect-Oriented Programming
API	Application Programming Interface
BIOS	Basic Input/Output System
CA	Certification Authority
CAPI	Coherent Accelerator Processor Interface
CoT	Chain of Trust
CPU	Central Processing Unit
CSP	Cloud Service Provider
DB	Database
DDR	Double Data Rate
DC	Data Centre
DIMM	Dual In-line Memory Module
DRTM	Dynamic Root of Trust Measurement
EC	European Commission
EE	Execution Environment
EINIT token	Enclave Initialization Token
EK	Endorsement Key
FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array
GB	Gigabyte
Gb	Gigabit
GbE	Gigabit Ethernet
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HW	Hardware
Hypapp	Hypervisor Application

I/O	Input / Output
IPC	Inter-Process Communication
ISVPRODID	Unique Product ID
ISVSVN	Security Version Number
KGA	Key Generation Authority
KLOC	Kilo Lines of Code
LE	Launch Enclave
LAN	Local Area Network
MAC	Message Authentication Code
MA-KGS	Multi-Authority Key Generation System
MD5	Message-Digest Algorithm 5
μTPM	Micro Trusted Platform Module
NFV	Network Function Virtualization
NIC	Network Interface Card
NPS	Network Protocol Stack
NSL	Network Service Layer
OS	Operating System
PAL	Piece of Application Logic
PCI	Peripheral Component Interconnect
PCR	Platform Configuration Register
PD	Protection Domain
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PSL	Power Service Layer
RAM	Random Access Memory
RDF	Resource Description Framework
ROM	Read-Only Memory
RoT	Root of Trust
RPC	Remote Procedure Call
RSA	Rivest-Shamir-Adleman
RTC	Real-Time Clock
SATA	Serial Advanced Technology Attachment
SC	Scheduling context
SDK	Software Development Kit
SDN	Software-Defined Networking
SGX	Software Guard eXtensions
SHA	Secure Hash Algorithm
SIGSTRUCT	Enclave Signature Structure
SLA	Service Level Agreement
SoC	System-on-a-Chip

SRK	Storage Root Key
SRTM	Static Root of Trust Measurement
SW	Software
TB	Terabyte
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TCP/IP	Transmission Control Protocol / Internet Protocol
TEE	Trusted Execution Environment
TPM	Trusted Platform Module
TRTM	TrustVisor Root of Trust Measurement
TVMM	Trusted Virtual Machine Monitor
U-Cloud	User Cloud
VM	Virtual Machine
VMM	Virtual Machine Monitor
VPN	Virtual Private Network
VXLAN	Virtual Extensible LAN
vTPM	Virtualized Trusted Platform Module
XMHF	eXtensible and Modular Hypervisor Framework

Bibliography

[1] Apache hadoop. URL: http://hadoop.apache.org/.

[2] *Apache Mesos*. URL: https://mesos.apache.org/.

[3] Apache spark. URL: http://spark.apache.org/.

[4] Atomic - docker & selinux. URL: http://www.projectatomic.io/.

[5] Defense4all - detection and mitigation ddos attacks. URL: https://wiki.opendaylight.org/view/Defense4All:Release_Notes.

[6] Docker. URL: https://www.docker.com/.

[7] Drakvuf - dynamic malware analysis. URL: http://drakvuf.com/.

[8] Hardening framework - hardening.io. URL: http://hardening.io/.

[9] Intel Software Guard Extensions (Intel SGX) SDK. https://software.intel.com/en-us/sgx-sdk. Last accessed: 31st of May 2016.

[10] jClouds. URL: https://jclouds.apache.org/.

[11] Kubernetes. URL: http://kubernetes.io/.

[12] Marathon - a cluster-wide init and control system for services in cgroups or docker containers. URL: https://mesosphere.github.io/marathon/.

[13] Moon - security management. URL: https://wiki.opnfv.org/moon.

[14] Oasis tosca. URL: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca.

[15] OP-TEE. https://wiki.linaro.org/WorkingGroups/Security/OP-TEE. Last accessed: 31st of May 2016.

[16] Open Virtualization for ARM TrustZone. http://www.openvirtualization.org/. Last accessed: 31th of May 2016.

[17] Openstack. URL: http://www.openstack.org/.

[18] Rightscale. URL: http://www.rightscale.com/.

[19] Security hardening for openstack-ansible hosts. URL: http://specs.openstack.org/openstack/openstack-ansible-specs.

[20] Xen Project Wiki. http://wiki.xenproject.org/wiki/Hypercall. Last accessed: 20th of July 2016.

[21] I. Abbadi. Clouds Trust Anchors. In *IEEE International Conference on Trust, Security, and Privacy in Computing and Communications (TrustCom)*, 2012.

[22] Mohammed Achemlal, Said Gharout, and Chrystel Gaber. Trusted Platform Module as an Enabler for Security in Cloud Computing. In *Network and Information Systems Security (SAR-SSI)*, 2011.

[23] ARM Security Technology. Building a secure system using trustzone technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009. Last accessed: 31st of May 2016.

[24] P. Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *SOSP'03*.

[25] M. Bartock, M. Souppaya, R. Yeluri, U. Shetty, J. Greene, S. Orrin, H. Prafullchandra, J. McLeese, and K. Scarfone. NISTIR 7904: Trusted Geolocation in the Cloud: Proof of Concept Implementation. http://csrc.nist.gov/publications/drafts/ir7904/nistir_7904_second_draft.pdf. Accessed: April 7, 2014.

[26] M. Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *OSDI '10*.

[27] Karyn Benson, Rafael Dowsley, and Hovav Shacham. Do you know where your cloud files are? In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, pages 73–82, New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/2046660.2046677, doi:10.1145/2046660.2046677.

[28] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized Trust Management. In *IEEE Conference on Security and Privacy*, Oakland, California, USA, 1996. URL: http://www.crypto.com/papers/policymaker.pdf.

[29] Michaela Blott et al. Achieving 10Gbps line-rate key-value stores with FPGAs. In *the 5th USENIX Workshop on Hot Topics in Cloud Computing*, 2013.

[30] Stuart Byma et al. FPGAs in the cloud: Booting virtualized hardware accelerators with openstack. In *Proceedings of the 2014 IEEE 22Nd International Symposium on Field-Programmable Custom Computing Machines*, FCCM '14, pages 109–116, 2014. doi:10.1109/.40.

[31] Ivano Cerrato, Alex Palesandro, Fulvio Risso, Marc Suñé, Vinicio Vercellone, and Hagen Woesner. Toward dynamic virtualized network services in telecom operator networks. *Computer Networks*, 2015.

[32] Melissa Chase. Multi-authority attribute based encryption. In *Proceedings of the 4th Conference on Theory of Cryptography*, TCC'07, pages 515–534, Berlin, Heidelberg, 2007. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1760749.1760787.

[33] Fei Chen et al. Enabling FPGAs in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, pages 3:1–3:10, New York, NY, USA, 2014. ACM. doi:10.1145/2597917.2597929.

[34] Victor Costan and Srinivas Devadas. Intel sgx explained. Cryptology ePrint Archive, Report 2016/086, 2016. http://eprint.iacr.org/.

[35] S. Crago et al. Heterogeneous cloud computing. In *2011 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 378–385, Sept 2011. doi:10.1109/CLUSTER.2011.49.

[36] Jeffrey Dean et al. Large scale distributed deep networks. In *Neural Information Processing Systems, NIPS 2012*.

[37] Beniamino Di Martino, Dana Petcu, Roberto Cossu, Pedro Goncalves, Tams Mhr, and Miguel Loichate. Building a mosaic of clouds. In *Euro-Par 2010 Parallel Processing Workshops*, volume 6586. 2011.

[38] Marcos Dias de Assunção, Rajkumar Buyya, and Srikumar Venugopal. Intergrid: A case for internetworking islands of grids. *Concurr. Comput. : Pract. Exper.*, 20(8):997–1024, June 2008. URL: `http://dx.doi.org/10.1002/cpe.v20:8`, `doi:10.1002/cpe.v20:8`.

[39] B. Kauer et al. Recursive Virtual Machines for Advanced Security Mechanisms. IEEE DSNW'11.

[40] D. Williams et al. Plug into the Supercloud. *IEEE Internet Computing*, 17(2), 2013.

[41] A. Fishman, Mike Rapoport, Evgeny Budilovsky, and Izik Eidus. Hvx: Virtualizing the cloud. In *HotCloud'13*.

[42] H. Giefers et al. Analyzing the energy-efficiency of dense linear algebra kernels by power-profiling a hybrid cpu/fpga system. In *2014 IEEE 25th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 92–99, June 2014. `doi:10.1109/ASAP.2014.6868642`.

[43] GlobalPlatform Inc. GlobalPlatform Device Specifications. `http://www.globalplatform.org/specificationsdevice.asp`. Last accessed: 31st of May 2016.

[44] Mark Gondree and Zachary N.J. Peterson. Geolocation of data in the cloud. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 25–36, New York, NY, USA, 2013. ACM. URL: `http://doi.acm.org/10.1145/2435349.2435353`, `doi:10.1145/2435349.2435353`.

[45] B. Huffaker, M. Fomenkov, and k. claffy. Geocompare: a comparison of public and commercial geolocation databases - Technical Report . Technical report, Cooperative Association for Internet Data Analysis (CAIDA), May 2011.

[46] Michael Hüttermann. Infrastructure as code. In *DevOps for Developers*, pages 135–156. 2012.

[47] IBM. *Base Board #2 for Zurich Microserver System*, Aug 2016. Rev. 1.0.

[48] IBM. *FPGA Module for Zurich Microserver System*, Aug 2016. Rev. 0.8.

[49] Intel Corporation. Intel software guard extensions programming reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, oct 2014. Last accessed: 31st of May 2016.

[50] Intel Corporation. Intel software guard extensions (intel sgx). https://software.intel.com/sites/default/files/332680-002.pdf, jun 2015. Last accessed: 31st of May 2016.

[51] P. Jain, S. Desai, S. Kim, M.-W. Shih, J. Lee, C. Choi, Y. Shin, T. Kim, B. B. Kang, and D. Han. OpenSGX: An Open Platform for SGX Research. In *Network and Distributed System Security Symposium (NDSS)*, 2016.

[52] L. Kapoor, S. Bawa, and A. Gupta. Hierarchical chord-based resource discovery in intercloud environment. In *Utility and Cloud Computing (UCC), 2013 IEEE/ACM 6th International Conference on*, pages 464–469, Dec 2013. `doi:10.1109/UCC.2013.91`.

[53] Ethan Katz-Bassett, John P. John, Arvind Krishnamurthy, David Wetherall, Thomas Anderson, and Yatin Chawathe. Towards ip geolocation using delay and topology measurements. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, IMC '06, pages 71–84, New York, NY, USA, 2006. ACM. URL: `http://doi.acm.org/10.1145/1177080.1177090`, `doi:10.1145/1177080.1177090`.

[54] H. Kazari and M. Lacoste. Towards Management of Chains of Trust for Multi-Clous with Intel SGX. In *Second ComPAS Workshop on Security in Clouds (SEC2)*, 2016.

[55] Taskin Kocak and Daniel Lacks. Design and analysis of a distributed grid resource discovery protocol. *Cluster Computing*, 15(1):37–52, 2012. URL: http://dx.doi.org/10.1007/s10586-010-0147-2, doi:10.1007/s10586-010-0147-2.

[56] R. Landa, J. T. Arajo, R. G. Clegg, E. Mykoniati, D. Griffin, and M. Rio. The large-scale geography of internet round trip times. In *IFIP Networking Conference, 2013*, pages 1–9, May 2013.

[57] Allison Lewko and Brent Waters. *Decentralizing Attribute-Based Encryption*, pages 568–588. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. URL: http://dx.doi.org/10.1007/978-3-642-20465-4_31, doi:10.1007/978-3-642-20465-4_31.

[58] Yanhuang Li, Nora Cuppens-Boulahia, Jean-Michel Crom, Frédéric Cuppens, Vincent Frey, and Xiaoshu Ji. Similarity Measure for Security Policies in Service Provider Selection. In *11th International Conference on Information Systems Security (ICISS)*, pages 227–242. Springer International Publishing, 2015.

[59] J.W. Lockwood et al. A low-latency library in FPGA hardware for high-frequency trading (HFT). In *2012 IEEE 20th Annual Symposium on High-Performance Interconnects (HOTI)*, pages 9–16, Aug 2012. doi:10.1109/HOTI.2012.15.

[60] R.P. Luijten and A. Doering. The DOME embedded 64 bit microserver demonstrator. In *2013 International Conference on IC Design Technology (ICICDT)*, pages 203–206, May 2013. doi:10.1109/ICICDT.2013.6563337.

[61] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *3rd ACM European Conference on Computer Systems (Eurosys)*, 2008.

[62] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy*, 2010.

[63] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure For TCB Minimization. In *3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, New York, New York, USA, 2008. URL: http://dl.acm.org/citation.cfm?doid=1352592.1352625.

[64] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *HASP*, 2013.

[65] Markus Miettinen, Ferdinand Brasser, and Ahmad-Reza Sadeghi. SUPERCLOUD, D1.1 - SUPERCLOUD Architecture Specification, 2015. URL: https://supercloud-project.eu/downloads/SC-D1.1-Architecture-Specification-PU-M10.pdf.

[66] B. Merrihan Monir, Mohammed H. AbdelAziz, AbdelAziz A. AbdelHamid, and El-Sayed M. El-Horbaty. Trust Management in Cloud Computing: A Survey. In *IEEE Seventh International Conference on Intelligent Computing and Information Systems (ICICIS'15)*, 2015.

[67] Sascha Müller, Stefan Katzenbeisser, and Claudia Eckert. *Distributed Attribute-Based Encryption*, pages 20–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. URL: http://dx.doi.org/10.1007/978-3-642-00730-9_2, doi:10.1007/978-3-642-00730-9_2.

[68] Open Platform for NFV (OPNFV). Available: https://www.opnfv.org/.

[69] Montida Pattaranantakul, Ruan He, Ahmed Meddahi, and Zonghua Zhang. SecMANO: Towards Network Functions Virtualization (NFV)-based Security MANagement and Orchestration. In *15th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2016.

[70] Zachary N. J. Peterson, Mark Gondree, and Robert Beverly. A position paper on data sovereignty: The importance of geolocating data in the cloud. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'11, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association. URL: `http://dl.acm.org/citation.cfm?id=2170444.2170453`.

[71] D. Pletea, S. Sedghi, M. Veeningen, and M. Petkovic. Secure distributed key generation in attribute based encryption systems. In *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 103–107, Dec 2015. `doi:10.1109/ICITST.2015.7412067`.

[72] G. J. Popek and R. P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM*, 17(7), 1974.

[73] Andrew Putnam et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 13–24, Piscataway, NJ, USA, 2014. IEEE Press.

[74] Kaveh Razavi, Ana Ion, Genc Tato, Kyuho Jeong, Renato Figueiredo, Guillaume Pierre, and Thilo Kielmann. Kangaroo: A tenant-centric software-defined cloud infrastructure. In *IEEE International Conference on Cloud Engineering*, 2015.

[75] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted Execution Environment: What It Is, and What It Is Not. In *IEEE Trustcom/BigDataSE/ISPA*, 2015.

[76] Joo Soares, Miguel Dias, Jorge Carapinha, Bruno Parreira, and Susana Sargento. Cloud4nfv: A platform for virtual network functions. In *CLOUDNET'14*, pages 288–293, 2014.

[77] S. Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. EuroSys'07.

[78] Trusted Computing Group. TPM Main Specification Version 1.2, 2011. Rev. 116.

[79] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. In *IEEE Symposium on Security and Privacy*, 2013.

[80] Bruno Vavala, Nuno Neves, and Peter Steenkiste. Secure Identification of Actively Executed Code on a Generic Trusted Component. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, Toulouse, France, 2016.

[81] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf. Enabling FPGAs in hyperscale data centers. In *2015 IEEE International Conference on Big Data and Cloud Computing (CBDCom)*, pages 1078–1086, August 2015.

[82] Peter Wright, Yih Leong Sun, Terence Harmer, Anthony Keenan, Alan Stewart, and Ronald Perrott. A constraints-based resource discovery model for multi-provider cloud environments. *Journal of Cloud Computing: Advances, Systems and Applications*, 1(1):1–14, 2012. URL: `http://dx.doi.org/10.1186/2192-113X-1-6`, `doi:10.1186/2192-113X-1-6`.

[83] T. Wu. The srp authentication and key exchange system. RFC 2945, RFC Editor, September 2000.

[84] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *23rd ACM Symposium on Operating Systems Principles*, Cascais, Portugal, 2011.