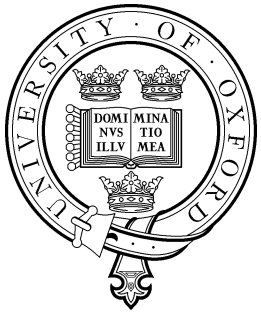
A Separation-of-Powers Model for a Trustworthy and Open Cloud Computing Ecosystem



Anbang Ruan

Kellogg College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Michaelmas 2014

Dedicated to my dear parents Yu Ruan and Ying Zheng.

Abstract

Security issues have become the key barrier to the adoption of Cloud Computing services. Most existing security enhancements lack a widely-agreed definition of trust. Trusted Cloud models have been proposed, which establish a Root-of-Trust inside the cloud and vouch for the trustworthiness of the cloud services. However, these are often impractical and ineffective due to the cloud's characteristics of complexity, heterogeneity, and dynamism. This dissertation thus focuses on how to effective manage the trust dynamics inside the cloud, and how to export trust to achieve practical cloud attestations.

Firstly, a *Separation-of-Powers (SoP)* model is designed. It separates the authorities of a Cloud Service Provider, and allows different independent roles to participate in managing trust inside the cloud. The collaborative-restrictive relationship among these roles encourages a trustworthy and open cloud ecosystem. Secondly, three core components for implementing this model are designed, solving the problems of: how to effectively determine a *Cloud Trusted Computing Base (cTCB)* for a cloud application; how to define a *Cloud Root-of-Trust (cRoT)* for managing the trust evidence for this *cTCB*; and how to construct a *Cloud Chain-of-Trust (cCoT)* from the *cRoT* to export the trust evidence, and achieve cloud application attestations. Thirdly, simulators and prototypes are implemented to evaluate these core components. A Trusted MapReduce (TMR) system is also built as a case study to demonstrate how to utilize the trust services achieved by the SoP model.

This dissertation demonstrates that, by correctly managing trust inside the cloud, the genuine behaviours of the cloud can be effectively inspected and verified. The SoP model builds trust from customers to the Cloud Services Providers. *Trustworthiness* supports security-critical cloud applications, which encourages a wider range of cloud users. *Openness* further brings a flourishing market to the ecosystem. It encourages many more diverse Cloud Service Providers to equally participate in the cloud ecosystem, regardless of their scale or capabilities. We believe that a model of this kind is important for achieving trustworthy governance in the cloud ecosystem. It could in turn help to promote a wider cloud model adoption.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud computing is becoming an essential technology for IT professionals and researchers alike. The cloud's on-demand provisioning of computing resources has ushered a shift in application deployment. Developers are no longer required to build and maintain applications on expensive in-house infrastructures. Instead, they may now leverage publicly accessible, multi-tenant systems administered by third parties to host their applications and data. The promise of reduced cost and maintenance has already excited many companies. According to the RightScale 2015 State of the Cloud Report [1], 88% of enterprises are using public cloud while 63% are using private cloud. Meanwhile, a promising growing potential is still expected, as 68% of enterprises run less than a fifth of their application portfolios in the cloud, and 55% of enterprises report that a significant portion of their existing application portfolios are not in the cloud, but are built with cloud-friendly architectures. Huge requirements from the enterprises have driven the rapid growth of the global cloud computing market. A recent report from the Bessemer Venture Partners' Byron Deeter [2] showed that the cloud computing market is growing at a 22.8% compound annual growth rate, and will reach $\$127.5$ billion in 2018.

However, security issues are still widely seen as the key barrier to adopting the cloud model [3, 4]. In recent years, substantial security threats to this rapidly growing cloud computing service model have attracted a great deal of attention. Massive data loss has drawn cloud customers to pay serious attention to the confidentiality and integrity of their data and computation when these are outsourced to a cloud provider. The scale of the software and data being processed, together with the complexity in providing the infrastructure, have complicated the cloud's correct implementation and management. Various attacks on the virtualization layer have aroused widespread concerns [5], and these are just one kind of a number of dangers possibly hidden from customers who move their assets to the cloud. It is also difficult to assure that even the most skillful administrators or sophisticated management software is error-free. Well-publicized incidents involving the loss of confidentiality

or integrity of customer data [6, 7, 8] have only revealed the tip of the iceberg concerning the potential threats to customers' interests.

Consequently, the massive demands for the cloud and the concerns over the security issues have become the two driving forces to contribute to the rapid growth in the global cloud security market. According to Transparency Market Research [9], the key factors driving the growth of the cloud security market include superior advantages of cloud security services, increasing demand for cloud computing by small and medium businesses (SMBs), and proliferation of handheld devices along with the rising trend of bring your own device (BYOD) and choose your own device (CYOD) policies. However, the lack of awareness and the enterprises' skeptical nature towards cloud services are restraining the cloud security market growth. From MarketsandMarkets' report [10], the global cloud security market has reached $\$4.20$ billion in 2014 and is expected to grow to $\$8.71$ billion in 2019. This represents an estimated Compound Annual Growth Rate (CAGR) of 15.7% from 2014 to 2019. According to the Transparency Market Research [9], the global cloud security market reached $\$4.5$ billion in 2014, and will hit nearly $\$12$ billion by 2022. That would be a compound annual growth rate of 12.8%.

Prevalant cloud security enhancements are drawn from a variety of paradigms. One line of work treats the cloud as a large-scale operating system, and ports modules from traditional secure operating systems [11, 12, 13, 14]. Authorization [11] and authentication [12] mechanisms protect customers' assets in the cloud from unauthorized access. Auditing modules [13] record actions performed in the cloud for evaluations at a future time. Another proposes new designs of cloud infrastructure to allow third parties or customers themselves to plant security enhancement modules into the cloud for protecting or inspecting targeted services or applications [15, 16, 17]. IDS and firewalls [18] protect internal cloud resource from attacks, and VMI (Virtual Machine Introspection) [19] interfaces enable fine-grained inner-VM protections from security services provided by the cloud, e.g. antivirus and rootkit detector. A full line of work for secure data outsourcing [14] has also been proposed. With a higher-level point of view, the Cloud Security Alliance [20] defines a comprehensive cloud security reference model, with twelve domains covering different levels of security mechanisms in the cloud.

However, most (if not all) of these security enhancements are implemented as *components of* the cloud system: they are in fact services provided by the cloud. It is still difficult to effectively prove that the advertised services, including the security enhancements, are actually present and enforced as expected. As a party cannot simply claim its own trustworthiness, most of the current solutions only delegate this challenge to a lower-layer one, instead of actually solving it. For example, auditing or access control modules

in an IaaS cloud can be easily circumvented when the virtualization layer is tampered with [19]. The system designed to protect this virtualization layer is in fact a lower-layer hypervisor, which is still error-prone [5]. Therefore, no matter how strong the existing cloud security solutions are, their correctly enforcement still need to be verified by the customers. Under this circumstance, effective trust establishment has become a more critical security issue, because without trust, the effectiveness of the security mechanisms themselves is in doubt.

This recursive trust dependency finally calls for the definition of a fine-grained *Trusted Computing Base (TCB)* to determine the critical cloud resource dependency, a reliable *Root-of-Trust (RoT)* inside the cloud to serve as the undeniable trustworthy authority, and an unbreakable *Chain-of-Trust (CoT)* to bind the trustworthiness of all relevant service and security components in *TCB* to this *RoT*. Accordingly, we propose that the foundations for solving cloud security problems include: 1) clearly identifying a cloud application's *TCB*, which comprises all the cloud service components supporting the application and is essential for enforcing its security policy; 2) effectively and efficiently *rooting* and *managing* trust inside the cloud infrastructure; and 3) allowing customers to *attest* the trustworthiness of this TCB, by deducing trust implications from the *RoT* and the *CoT*. These compose the main topics of this thesis.

In the rest of this chapter, we first review the basic cloud computing model, and then identify the threat model for this research. We further discuss the trusted cloud expectations regarding these threats. Accordingly, the thesis statement is presented.

1.1 Cloud Computing Concepts

While several cloud service models have evolved in the market, the Infrastructure as a Service (IaaS) clouds appear to be the most interesting to large corporations, as they offer a Virtual Machine (VM) [21, 22] hosting as a primary service, enabling complex application designs. The virtualization technology also enables strong isolations among different customers' applications, and among different components in one application. With this model, customers treat the cloud resource as traditional physical machines. They deploy their applications directly into the VMs they purchased, and configure the necessary supporting facilities, such as the networking and the storage, in a way that is not very different from how they manage their own physical data centre. This similarity greatly reduces the transition cost when migrating their existing infrastructure to the cloud. The Platform as a Service (PaaS) and the Software as a Service (SaaS), on the other hand, are two other

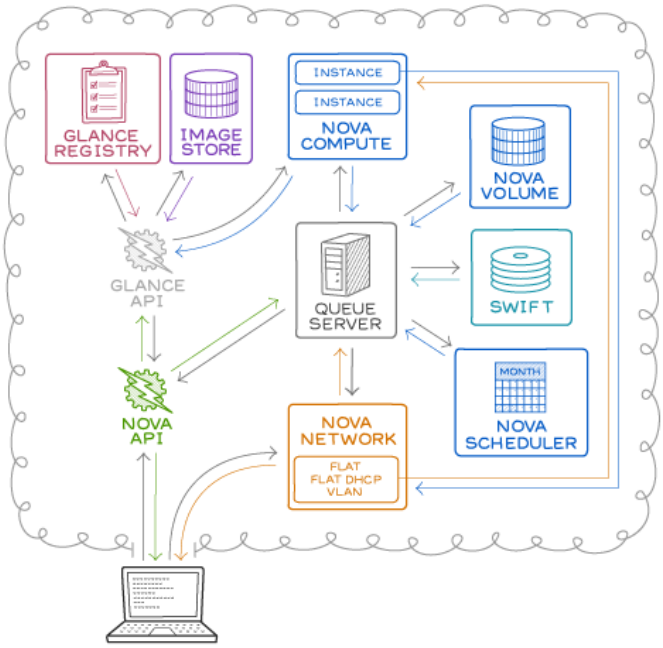
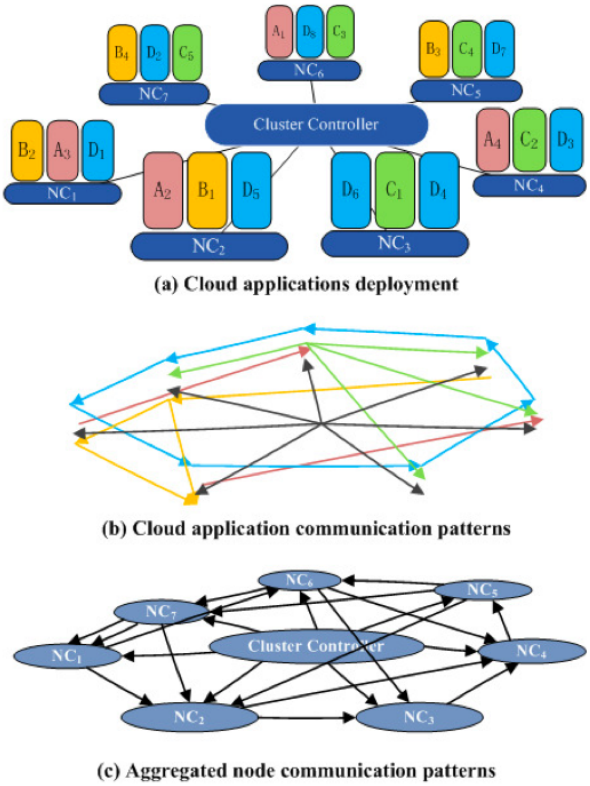


Figure 1.1: OpenStack Infrastructure Model

cloud models, which offer higher resource abstractions. PaaS presents the cloud resource as a set of APIs (Application Programming Interfaces), while SaaS offers a set of software functionalities. The underlying infrastructures of these two cloud models are much more opaque to the customers than the IaaS model. This makes their trust establishment and management much harder. For these reasons, in the current work, we focus only on the IaaS cloud model.

Figure 1.1 (from [23]) depicts the architecture of the OpenStack, one of the most widely used IaaS cloud systems. Cloud applications are implemented as a collection of *VMs*. *Compute* nodes (or *Compute* for short) are the basic computing units in the cloud. They host and manage the VMs. A cluster of *Computes* is managed by a *Scheduler* node. The permanent storage of VMs in a cluster is implemented and managed by a *Storage* node. Cloud providers also tend to implement additional security components, such as firewalls and IDSs (Intrusion Detection Systems). Therefore, the genuine enforcement of a cloud application depends on the integrity of the *Computes* hosting the VMs, the *Scheduler* node managing all these *Computes*, corresponding security components, and all other supporting services, such as the *Storage* node and the *Network* node, etc.

As discussed, virtualization technology and the cloud's dynamic resource provisioning allow customers to treat the VMs hosted in a cloud as independently managed physical machines. In reality, the multi-tenancy nature of an IaaS cloud indicates that VMs belonging to different customers may share a same *Compute*. As depicted in Figure 1.2(a), VMs from



(a) Cloud applications deployment

(b) Cloud application communication patterns

(c) Aggregated node communication patterns

Figure 1.2: Cloud Internal Communication Patterns

different cloud applications can be scattered among *Computes* (or the Node Controllers, NCs) in a cluster (managed by a Cluster Controller).

Meanwhile, applications have different internal communication patterns (Figure 1.2(b)). For example, in a typical web hosting application, communication traffic is made frequently from the front-end servers to the back-end database, while for some MapReduce-based data processing computations, traffic is made from a central master unit to several slave units. Therefore, the traffic of a single *Compute* is composed of all the traffic of its hosting VMs, as shown in Figure 1.2(c).

Recently, new systems have been designed to allow customers to implement discretionary infrastructure-level controls [16] inside the cloud. The VM Market [17] has also been proposed to allow customers to provide their VMs as services to others' VMs. Accordingly the cloud infrastructure has developed to support these advanced management controls and functional facilities. In Figure 1.3, we categorize three kinds of cloud services in an IaaS cloud system. They are organized in separate *service planes*, with the higher-layer services relying on the lower-layer ones.

- *Infrastructure Services (IS)* implement the basic cloud infrastructure. They manage the hardware resource and export a uniform hardware abstraction to support the VMs.

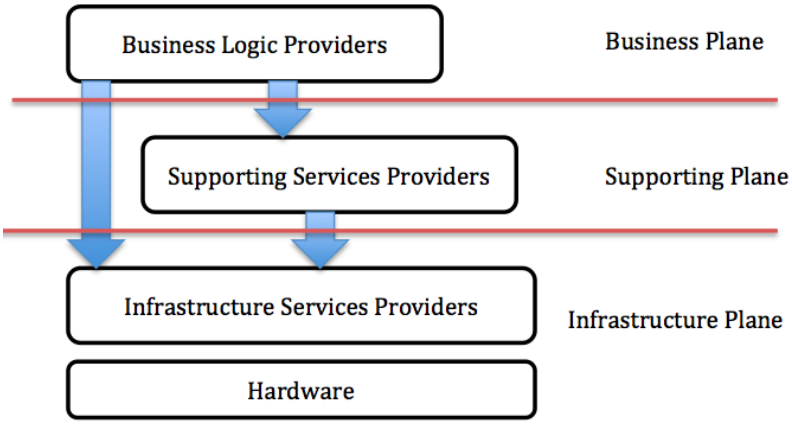


Figure 1.3: Cloud Service Model

They are the core services provided by the Cloud Service Provider (CSP). We identify the CSPs providing solely IS as the *Infrastructure Service Provider (ISP)*.

- *Supporting Services (SS)* are the supplemental services to enrich the functionalities of the infrastructure. We refer to their providers as the *Supporting Services Provider (SSP)*.

- *Business Logic Services (BLS)* are customers' applications, which utilize the infrastructure and supporting services in the cloud. They can serve only the customers' own requirements, or they can be provided as third party components to other customers' applications. Accordingly, these service providing customers are referred to as the *Business Logic Provider (BLP)*.

Supporting Services are usually additional security modules, which protect customers' assets, such as data encryption [14], or auditing services [13]. Supplemental functional services have also emerged, e.g. Software Defined Network services. ISPs usually serve as SSPs to enrich their cloud features. But it has also become common for the ISPs to export privileged controls and allow customers or third party service providers to implement their own cloud management and security-critical services [16, 15].

We define the cloud customers themselves as a type of service provider — the BLP. This definition facilitates an integrated presentation when we later discuss the threat model and the application's dependency in the cloud. On the other hand, cloud applications can also provide services to each other [17]. The providers of these applications are also regarded as the CSPs to the others.

1.2 Threat Model

Regarding the cloud service model, providers for the lower-layer services may violate their Service Level Agreement (SLA) and in turn compose of the upper-layer ones'. The lower-layer shared services may also be exploited by malicious upper-layer, in order to damage the interests of the other upper-layer service providers. Therefore, when outsourcing critical data and data processing applications to the cloud, customers may be concerned whether the CSP (Cloud Service Provider) will: 1) provide the promised services; 2) perform unauthorized access to customers' data or applications; 3) be strong enough to defend against attackers or malicious users. In particular, we categorize the following threats:

1) *Dishonest providers.* By providing lower-quality services, these providers gain benefits from the reduced costs. This is usually implemented by not enforcing the services with the expected quality for the target cloud customers. For example, the providers may enforce the data redundancy policy with a lower replication ratio, in order to save storage space. Meanwhile, the management and implementation complexity make unintended faults in the cloud hard to avoid. Providers unwilling to enforce extra measurements for reducing these faults will also cause SLA violations.

   Consequently, dishonest providers will cause the *loss of governance* for the cloud customers' digital assets. This is a typical general concern associated with cloud computing. It is linked to the transfer of management for critical data and computation to a cloud provider. It is further linked to the limitation in cloud management capabilities and transparency from the view of the user. The growing complexity and dynamisms have complicated the ways for customers to enforce correct controls inside the data or applications inside the cloud. Loss of governance violates the transparency of the cloud, which ultimately discourages cloud customers.

2) *Malicious providers.* Lower-layer providers usually have privileged controls over the upper-layer services. Attackers may abuse these privileges to launch malicious software and exploit the upper-layer services. Disaffected personnel may also abuse their privileges to compromise those services. For example, the customers' VMs' memory content may be maliciously examined by using the privileged Virtual Machine Introspection (VMI) techniques [24]. Customers' unencrypted data may also be accessed without authorisation.

   Accordingly, these providers introduce the *insider fraud*. Insider fraud addresses the risk that arises from the access of cloud administrators. Cloud administrators need the necessary access rights to fulfil their duties, but they also need to be prevented from

accessing critical data or introducing harmful software to the applications hosted in the cloud. This may be intentional or unintentional.

3) *Privileged malicious customers.* Customers may exploit the cloud with implementation vulnerabilities. For example, attackers may gain access to a general VM and run software that can attack through the hypervisor interfaces [19]. When they gain the privileged capabilities of the hypervisor, they are able to control other customers' VMs and expose confidential information.

   Therefore, these customers introduce the threat of *isolation failure*. Public clouds typically serve multiple tenants at the same time. Due to the use of virtualization technology, IaaS clouds can achieve an efficient use of hardware resources and load balancing. This includes the possibility that one machine hosts multiple VMs and potentially data and computation from different customers. Isolation failure addresses the general concern about data leakage or intrusion in between different tenant application environment hosted on the same cloud.

Accordingly, we observe that, these threats are mostly resulted from the *inabilities of the cloud customers to verify the genuine behaviours of the software loaded inside the cloud infrastructure*. When customers are able to attest the properties of the software loaded inside the cloud infrastructure, they will be able to make sure that: 1) the software services enforcing their SLAs have been genuinely enforced and correctly configured; and 2) no unexpected software components are loaded to access their digital assets inside the cloud without authorisation; 3) the software services hosting their VMs have not been tampered with. As we will define later, these software services or components constitute the cloud's *Trusted Computing Base (TCB)* for the cloud application.

On the other hand, in this work, we do not consider hardware attacks. We assume that the malicious users and the administrators cannot manipulate the hardware without being identified. We also do not consider insider physical attacks such as the cold boot attack [25]. Moreover, with our cloud model, we assume that direct interactions are needed for the malicious behaviours above to take effect, i.e. attacks can only be launched directly on the target nodes, or on the nodes that have direct interactions with the target nodes. We also do not address attacks launched from outside of the cloud's perimeter.

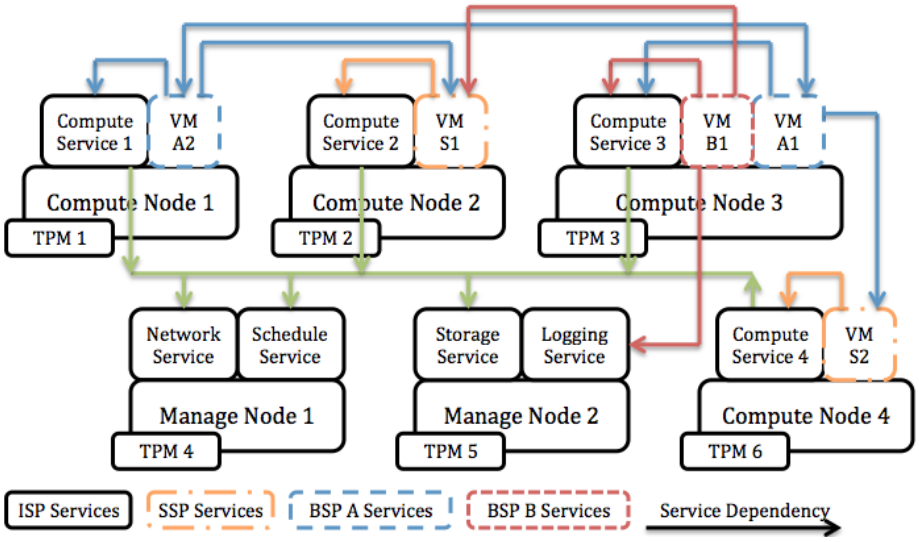


Figure 1.4: Cloud Attestation Scenario

1.3 Trustworthy Cloud Expectations

Trusted Computing technology defines the *trustworthiness* of a platform as its capability of genuinely recording and reporting the behaviours of all its loaded software components [26, 27, 28, 29]. We adopt this definition for the trustworthiness of a cloud system. Particularly, a trusted cloud allows cloud customers to determine the genuine behaviours of all the loaded cloud service components which have supported their cloud application or have the potential to enforce malicious behaviours.

Figure 1.4 depicts a simplified scenario where two cloud customers A and B, with conflict of interests, deploy applications on the same cloud infrastructure. As we focus only on the IaaS cloud model, we assume that all the cloud services are implemented as Virtual Machines (VMs), including the cloud management facilities, which are the default deployment structure in the Xen-Openstack installation [30]. Both applications depend on various supporting services inside the cloud. A's application includes *VM_A1* and *VM_A2*, while B's contains only *VM_B1*. *VM_A1* and *VM_B1* are hosted on one same compute node ($\#3$). Meanwhile, *VM_B1* and *VM_A2* both rely on an SSP's VM (*VM_S1*) deployed on compute node ($\#2$). In addition, *VM_A1* uses service from another SSP's VM (*VM_S2*), and *VM_B1* uses an add-on logging service provided by the CSP. All VMs depend on the Compute services running on their hosting nodes, and these Compute services in turn rely on the management services running on the Manage Nodes, namely the Network, Schedule, and Storage service.

In this scenario, the cloud TCB of A's application includes service VMs *VM_S1*, *VM_S2*, the Compute services $\#1 - \#4$, and all the management services. For B's application, the

cloud TCB includes service VM *VM_S1*, the logging service, compute services $\#2$ and $\#3$, and the management services. From A's perspective, it needs to verify the properties of all the components of its application and the cloud TCB to make sure: 1) the advocated services are genuinely enforced; and 2) no hidden information flow happens between its applications and B's through the shared services.

1.4 Thesis Statement

The thesis for this dissertation is as follows:

> ***By understanding the constituents of trusted platforms and cloud models, we can build a trustworthy cloud platform to allow users to query and verify its genuine behaviours in an effective and practical way. By carefully distributing and organizing these trust establishing responsibilities, we can achieve an open ecosystem where parties with diverse capabilities and credibility can collaborate to implement trustworthy cloud services.***

The rest of the dissertation starts with an introduction to various background knowledge and related work (Chapter 2). In Chapter 3, we will further discuss the difficulties of implementing a trustworthy cloud computing ecosystem due to the monarchical powers possessed by existing CSPs. The *Separation-of-Powers (SoP)* model is then presented to distribute CSPs' powers to three independent and mutual-restrictive roles. We will show that, by constructing the trust establishing procedures among these separated roles, this separation encourages a *trustworthy* and *open* cloud ecosystem.

To implement this model, this dissertation further presents the designs and implementations of three critical components. Firstly, in Chapter 4, the RepCloud framework is constructed to provide methodologies for dynamically identifying a fine-grained *Cloud Trusted Computing Base (cTCB)* for cloud applications. Secondly, in Chapter 5, the NeuronVisor framework is built to implement a *Cloud Root-of-Trust (cRoT)* for attesting the *cTCB* of each application. This *cRoT* implements a logical abstraction for managing trust evidence inside the cloud. It serves as root for effective attestations to the *cTCB* and the cloud application. Thirdly, in Chapter 6, the *Cloud Chain-of-Trust (cCoT)* is built to extend the trustworthiness from the *cRoT* to each component in the *cTCB* and the application. This facilitates effective attestations to the cloud applications.

Finally, to demonstrate the *SoP* model and the trusted cloud services, Chapter 7 presents the Trusted MapReduce (TMR) system as a case study. TMR is built by integrating trusted

computing semantics into the MapReduce workflows. We will further discuss how these semantics will be integrated with the trusted cloud implementation (the *cTCB*, the *cRoT* and the *cCoT*), and illustrate how trusted cloud components work together to reinforce critical workflows of a typical cloud application.

Chapter 2

Background and Related Research

In this chapter, we introduce the background of this research. We will start with the basic knowledge of the Trusted Computing, which forms the foundation to this work. We further discuss the distributed trust management system with the reputation systems, as their core ideas will be integrated with the Trusted Computing technologies in this work to implement scalable and flexible cloud attestation mechanisms. Finally, we survey the existing work on the designs and implementations of various Trusted Cloud systems.

2.1 Trusted Computing

The core idea of the Trusted Computing technology is to utilise a tamper-proof secure hardware to genuinely store and report the behaviours of a software system deployed and enforced on a hardware platform. To represent the behaviours of software, the cryptographic hash values of the software's binary codes and configuration files are calculated, as it is generally assumed that the software's behaviours are defined by its binaries and the configurations. These hash values are referred to as *measurement values*, and the processes of generating the measurement values and securely storing these values in the secure hardware are called *measure*.

The core technologies of the Trusted Computing therefore include *measuring*, *storing* and *reporting* these measurement values in a secure and trustworthy way. The combinations of these procedures help a challenger to determine the genuine behaviours of a remote computing platform. In the rest of this section, we will briefly introduce the basic concept of these core technologies, as they are most relevant to implementing the cloud attestations. We omit the other supporting Trusted Computing components, such as the hierarchical key management, sealing and binding operations, Trusted Network Connection (TNC), or the

other cryptographic components. These are very well documented in the specification: [31, 32].

2.1.1 Trusted Platform Module and the Root-of-Trust

The Trusted Computing Group (TCG) [31] specifies a hardware module, the Trusted Platform Module (TPM), to serve as a *root of trust* for a platform. The attestation architecture of TCG aims at *measuring* all loaded executables by hashing each piece of software into the TPM before loading it. These values are securely stored in the set of tamper-resistant registers inside the TPM, the *Platform Configuration Register (PCR)*. The platform is bootstrapped by the Core Root of Trust for Measurement (CRTM), which is trusted by default and will initiate measurements to the BIOS and the boot loader to construct a *chain of trust*. This chain is then extended through every operating system component up to the applications and their configuration files. Once the executables are measured into the TPM, the TPM can reliably attest to the hash values by signing them with a TPM protected key, i.e. the *Attestation Identity Key (AIK)*. The signed values (i.e. the *attestation tickets*) are sent to the challenger (or verifier), who can then decide whether to trust the target platform. The tickets generated by the TCG framework are ***deterministic***, i.e. they can reveal the genuine state of the target platform, as they are based on the tamper-proof registers protected by the TPM. Meanwhile, the tickets are themselves ***tamper-proof***, as they are signed by the TPM protected key. However, these tickets are ***transient*** as they can only reflect the genuine state of the platform up to the time when they are generated. Hence, attestations should be performed regularly for updating the trust states.

As with the limited computation capability of a TPM chip, a complete attestation procedure between two nodes may require several seconds, the latency of which is unacceptable when attestation is performed regularly. Stumpf et al. [33] proposed the Timestamped Hashchain-based Attestation to compensate for this deficiency. To avoid performing expensive TPM operations for every attestation request, the server (the node to be attested to) uses a nonce issued by a Trusted Third Party (TTP) binding to the time-stamp value of the current time for performing the TPM_Quote instruction regularly and generates an attestation ticket each time. As the nonce is publicly available to every attesting client, instead of randomly chosen by every one, attestation tickets generated by this scheme are ***disseminable*** and can be reused by a different client. As long as the time has been synchronized, clients can still determine the trustworthiness of the server from the ticket and deduce the freshness of the attestation from the nonce.

2.1.2 Remote Attestations

The attestation mechanism proposed in TCG specifications [31] is referred to as 'binary attestation' as information that is collected and reported about the state of a platform is in the form of binary hash measurements. Binary measurements provide configuration and implementation details of the software components running on a trusted platform. The security properties and functions of the component can also be reasoned about. This can potentially lead to security and privacy issues. Moreover, as components keep changing all the time, e.g. their configuration changes and version updates, reference measurements have to be updated, which dramatically increases the number of possible expected values for a component.

Sadeghi and Stüble propose the idea of delegation-based attestation in [27]. They highlight the drawbacks of binary based attestation and propose a variety of solutions based on the existing trusted computing functionality. They leverage the existing TCG attestation mechanism and demonstrate how property based attestation may be realized on top of it. Poritz et al. [26] propose a scalable and privacy friendly property attestation mechanism. The authors provide an attestation architecture that extends the TCG attestation mechanism using security properties of platforms.

Haldar et al. [34] introduced a semantic attestation mechanism based on the Trusted Virtual Machine (TVM). The TVM based semantic attestation mechanism enables the remote attestation of high-level program properties. Shi et al. proposed a fine-grained attestation scheme called BIND [35]. It provides evaluation interfaces to attest the security-concerned segments of code. Jaeger et al. [36] introduced the Policy-Reduced Integrity Measurement Architecture (PRIMA) based on the information flow integrity checking against Mandatory Access Control (MAC) policies. Program execution attestation introduced in [37] is to attest whether a program is executed as expected. These semantic attestation mechanisms still require a know-good binary code repository.

Li et al. [38] describe a model for attesting the behaviour rather than the binaries of platforms. The main objective of the model is to determine if a system will behave in a way that is compliant with a verifier's expectation. Alam et al. propose a model-based behaviour attestation using trusted platforms in [39]. In this model, the trustworthiness of a target platform is based on the behaviour of its policy model. The authors then move on to demonstrate the applicability of this framework using the formal specification of the UCON [40] model.

Trusted Execution Technology (TXT) and Secure Virtual Machine (SVM) are introduced to provide a trusted execution environment. In recent years, there have been already several practices [41, 42, 43] exploiting TXT or SVM. Open Secure LOader (OSLO) [42]

leverages the dynamic root of trust to implement a bootloader based on AMD SKINIT instruction. Flicker [43] was introduced as an infrastructure for executing security sensitive code in complete isolation. It leverages the Secure Virtual Machine (SVM) of AMD processors and provides fine-grained attestation on program execution. LaLa [41] combines the latest hardware virtualization and trust technologies to deliver a more robust platform to support both instant-on system and a full-featured OS, and the flexible architecture enables a platform user to benefit from the advantages of a fast booting platform and a full-featured mainstream OS at the same time.

2.2 Distributed Trust Management

In many collaborative applications, such as peer-to-peer systems, a trust relationship needs to be established before critical operations, but trust can only be determined after the operations have been performed. Reputation-based trust management systems are hence introduced to allow parties to build trust, or the degree to which one party has confidence in another within the context of a given purpose or decision. In these applications, trust only represents a "personal" view of a peer, and reputation systems [44, 45] are used for a peer to infer trust towards a stranger by consulting other peers it trusts or by aggregating all the others' views, on the assumption that most peers will "tell the truth". A typical reputation system usually defines the following components [44]: 1) *formulation,* the mathematical presentation of the reputation metric, together with the sources of input to that formulation; 2) *calculation,* the algorithm to calculate the formulation for a given set of constraints; and 3) *dissemination,* the mechanism for storing and disseminating the reputation value. As this "personal" view is very easy to forge, various attacks to the reputation systems exist, while several countermeasures have also been proposed [44].

The EigenTrust [45] reputation system was motivated by the need to filter out inauthentic content in peer-to-peer file sharing networks. EigenTrust calculates a global reputation value for each peer in the system based on the local opinions of all of the other peers. The local opinions of nodes are aggregated into a matrix format and the global reputation values are obtained by calculating the left principle eigenvector of that matrix. The TrustGuard framework [46] uses a strategic oscillation guard based on a Proportional-Integral-Derivative (PID) controller to combat malicious oscillatory behaviour. Fake feedbacks are prevented with the help of a fake transaction detection component which binds feedback to unforgeable transaction proofs. Scrivener [47] is based on principles from neoclassical economics which state that naturally rational clients must be given an incentive in order

to cooperate and not cheat the system. The goal of Scrivener is to enforce fairness among all participants in a peer-to-peer file sharing system. The reputation of each host is not a globally calculated value, but rather is specific to individual pairwise sharing relationships within the overlay network.

The P2PRep [48] reputation system is designed to mitigate the effects of selfish and malicious peers in an anonymous, completely decentralized system. The system uses fuzzy techniques to collect and aggregate user opinions into distinct values of trust and reputation. In P2PRep, trust is defined as a function based on an entity's reputation and several environmental factors such as the time since the reputation has last been modified. Credence [49] was motivated by the need for peers to defend against file pollution in peer-to-peer file sharing networks and has been deployed as an add-on to the LimeWire client for the Gnutella network. The system relies on the intuitive notion that honest identities will produce similar votes as they rate the authenticity of a file, implying that identities with similar voting patterns can be trusted more than identities with dissimilar voting patterns.

Several classes of attacks to reputation systems have been identified:

1. *Self-promoting.* Attackers manipulate their own reputation by falsely increasing it. Even if source data is authenticated using cryptographic mechanisms, self-promotion attacks are possible if disparate identities or a single physical identity acquiring multiple identities through a Sybil attack [50] collude to promote each other.

2. *Whitewashing.* Attackers escape the consequence of abusing the system by using some system vulnerability to repair their reputation. Once they restore their reputation, the attackers can continue the malicious behaviour. Often attackers will attempt to re-enter the system with a new identity and a fresh reputation [51]. The attack is also facilitated by the availability of cheap pseudonyms and the fact that reciprocity is much harder to maintain with easily changed identifiers [52].

3. *Orchestrated.* Attackers orchestrate their efforts and employ several of the above strategies. One example, known as an oscillation attack [46], is where colluders divide themselves into teams and each team plays a different role at different times.

4. *Denial of service.* Attackers cause denial of service by preventing the calculation and dissemination of reputation values. For example, some distributed storage components such as DHTs may have their own vulnerabilities [53] and they can be in turn exploited by an attacker to create denial of service against the reputation system.

2.3 Trusted Cloud Systems

This section discusses various related work on Trusted Cloud. It surveys existing proposals, and identifies their strengths and weaknesses. Finally, it summarizes the criteria for implementing a practical trusted cloud infrastructure.

The issue of establishing trust in the cloud has been discussed by many authors (e.g. [54, 55, 56, 57, 58]). Much of the discussion has been centred around reasons to "trust the cloud" or not to. Khan and Malluhi [57] discusses factors that affect consumers' trust in the cloud and some of the emerging technologies that could be used to establish trust in the cloud including enabling more jurisdiction over the consumers' data through provision of remote access control, transparency in the security capabilities of the providers, independent certification of cloud services for security properties and capabilities and the use of private enclaves. The issue with jurisdiction is echoed by Hay et al. [56], who further suggest some technical mechanisms including encrypted communication channels and computation on encrypted data as ways of addressing some of the trust challenges. The work in [54, 55] focus on identifying the properties for establishing trust in the cloud.

Trusted Cloud systems have been proposed to utilize the TCG trusted computing technology in the cloud infrastructure. Trusted Virtual Datacenter (TVDc) [59] incorporates trusted computing technologies into virtualization and system management software. It provides strong isolation between workloads by enforcing a Mandatory Access Control (MAC) policy throughout a data centre. TVDc also provides integrity guarantees to each workload by leveraging a hardware root of trust in each platform to determine the identity and integrity of every piece of software running on a platform. In addition, TVDc allows centralized management of the underlying isolation and integrity infrastructure.

Trusted Cloud Computing Platform (TCCP) [60] enables users to attest to the IaaS provider and determine whether or not the service is secure before they launch their virtual machines. A Trusted Coordinator (TC) maintained by an External Trusted Entity attests to the nodes inside the cloud, and controls the critical operations of the nodes with a set of protocols, such as VM migration and instantiation.

Private Virtual Infrastructure (PVI) [61] is a management and security model for cloud computing. The PVI data centre is under control of the information owner while the cloud fabric is under control of the service provider. A cloud Locator Bot pre-measures the cloud for security properties, securely provisions data centres in the cloud, and provides situational awareness through continuous monitoring of the cloud security.

The Cloud Verifier (CV) [58] service generates integrity proofs for customers to verify the integrity and access control enforcement abilities of the cloud platform that protect the

integrity of customers' application VMs in an IaaS cloud. Cloud customers can verify that the cloud verifier satisfies their integrity property requirements and that the properties of the cloud verifier vouches for being enforced on its hosts satisfies the customer's properties for those components as well.

However, these systems generally assume the homogeneity of a cloud service deployment. As we discussed in Section 1.1, existing cloud systems have evolved to support much more diverse services to satisfy different customers' needs. This introduces a heterogeneous cloud TCB for the applications, which hampers the effective enforcement of the trusted cloud mechanisms. Therefore, it is difficult for the CSP-implemented centralized attestation delegate to attest to the complicated trust dynamic inside the cloud. It is also difficult for the customers to verify the effectiveness of those delegates.

The work of [58, 60] provides remote attestation for either the entire cloud infrastructure or for the physical resources hosting a specific VM. However, we argue that it is not practical to attest to the entire cloud infrastructure considering its huge and distributed resources, neither is it practical to attest to a specific set of physical resources considering the dynamic nature of clouds where VMs can move between different physical resources. In addition, these systems require users to understand to some extent the cloud infrastructure, i.e. they do not provide transparent cloud infrastructures.

Shamon [62] is a distributed system in which MAC (Mandatory Access Control) policies can be enforced across physically separated systems. By bridging the reference monitor between those systems, different logical compartments enforced by MAC policies can be achieved across physical machines. Trust in the MAC enforcement capabilities of a remote system is established using remote attestation. Shamon emphasizes only the access control mechanisms. It employs the remote attestation only to ensure the integrity of the MAC capabilities instead of examining the real properties of the applications. Therefore, it is not capable of controlling the access from the applications with authorized credentials but having altered behaviors, e.g. when being tampered with. However, it compliments the system we design in this thesis, as we focus on ensuring the expected behaviors of a loaded application.

Parno et al. [63] focus on bootstrapping trust in commodity systems. Various aspects of the TPM-based chain of trust and correspondingly secure storage and remote attestation schemes are discussed. As this work is based only on building trust on a single commodity system, particular concerns arise when implementing a trusted cloud architecture. As discussed in previous sections, the complexity and dynamics of clouds require the implementation of the compositional chains of trust, in which resources with various security

properties can exist inside a cloud to satisfy different scheduling requirements, while similar nodes are organized to significantly reduce management overheads.

2.4 Summary

In this chapter, we introduced the foundation to this work, mainly including the Trusted Computing technology (Section 2.1) and the Distributed Trust Management Systems (Section 2.2). The integration of these two lines of trust management schemes helps us to design the *Decentralized Attestation*, which will solve the problems when implementing cloud attestations with the existing systems introduced in Section 2.3. In the next chapter, we will start with the introduction of a *Separation-of-Powers* model, which sets the design principles for building our trusted cloud system.

Chapter 3

Separation-of-Powers Model

> *"When the legislative and executive powers are united in the same person, or in the same body of magistrates, there can be no liberty... there is no liberty if the powers of judging is not separated from the legislative and executive... there would be an end to everything, if the same man or the same body... were to exercise those three powers."*
>
> — Montesquieu, *The Spirit of Laws.*

3.1 Overview

When customers outsource their private data or applications to third-party cloud computing infrastructures, they have to make critical decisions on how much trust they can put on the Cloud Service Providers (CSPs). With already abundant and still rapidly growing numbers of CSPs in the market, they seem to have plenty of choices. However, technology barriers and insufficient regulations have limited the range of these choices, and will finally force them to make compromises: this is mainly because CSPs have become the omnipotent authority for *defining*, *enforcing* and *inspecting* the services they provide. When customers deploy their applications or data to the cloud, they have no measurements to effectively determine what service components the CSPs have enforced to meet their requirements. They also have no measurements to genuinely examine the properties of those components. Customers need to know whether the services they purchased have been genuinely enforced, and whether only authorized operations are performed on their outsourced data or applications. However, existing cloud systems lack the mechanisms to effectively provide this knowledge for the customers to verify the *trustworthiness* of the CSPs and the services they provide.

Under this circumstance, customers have no choice but to trust the CSPs whom they *believe* to be relatively trustworthy. Most of the time, they choose the big CSPs, in the

hope that they will be less likely to tamper with customers' benefits, as the big CSPs may value their reputations more. We define this kind of trust as *Blind Trust*. Blind Trust leads customers to big CSPs. It makes the big ones bigger; and the small ones ultimately extinct. This eventually brings into existence the *Monarchies* in the cloud computing ecosystem. As Sun's former CTO Greg Papadopoulos once commented [64]: *"there will be, more or less, five hyperscale, pan-global broadband computing services giants"*. These powerful monarchies will eventually control every detail of customers' activities in the cloud and leave them no choice but to compromise.

Various cloud auditing schemes [65, 66, 67, 68] have been enforced by authorities, which have established social trustworthiness, e.g. the government sectors. They evaluate the cloud implementations against certain criteria, and certify CSPs' trustworthiness. However, these evaluations are usually performed in a one-off manner, and with a relatively long interval, e.g. once a year. They examine the entire infrastructure from a higher-level point of view. As we will discuss in Chapter 4, the complexity and dynamic nature of a cloud will prohibit the effectiveness of these schemes. It is also difficult for a customer to determine, on a session-basis, whether their exact SLAs have been fulfilled. Moreover, these new authorities have gained the powers on both *defining* and *inspecting* the genuine behaviours of a cloud. They have the capabilities to dictate the cloud services' properties. Customers still have no choice but to *believe* that they are generally honest, and are free from insider attacks or political scandals.

Several authors have proposed the idea of a *Trusted Cloud* [60, 58], which allows customers to validate the genuine enforcements of the cloud services with Trusted Computing technology. Trusted Cloud integrates cloud systems with the Trusted Computing infrastructure [31] (Section 2.1). In these systems, hardware resources are deployed with built-in tamper-proof trustworthy hardware chips: the *TPMs* [32]. The TPM will facilitate the reliably recording and reporting of the genuine behaviours of a platform. *Attestation* services are hence designed to gather the TPM-generated trust evidence, and to testify the properties of the cloud services.

However, existing Trusted Cloud systems mostly suggest CSPs themselves to implement the *attestation delegates* inside their cloud infrastructure. These delegates attest all the services inside the cloud to vouch for their genuine behaviours. Customers further attest the delegates to infer the cloud's trustworthiness. As CSPs' manage these delegates, their powers are still not controlled effectively. Customers' attestations to the delegates only prove their *existence* rather than their *effectiveness*. In other words, additional attestations to the detailed configurations of this delegate are necessary for testifying that the delegates are genuinely configured to fulfil the customers' attestation requirements. This necessity is

especially eminent when considering the *Supporting Services (SS)* and the *Business Logic Services* (Chapter 1), as they introduce different properties in the cloud infrastructure to different customers' point of view. Consequently, this attestation is especially complex given the gigantic-scale and multi-tenancy nature of a cloud infrastructure, especially when considering the limited knowledge possessed by customers.

We refer to this problem as *Blurred Trust*, where the trustworthiness of a CSP is attested by a third party, but not enough information is given to actually prove the establishments of the trust. For example, in the Trusted Cloud Computing Platform (TCCP) [60], customers may perform attestations to the cloud nodes, that host their VMs before they deploy the VMs to the cloud. However, they still have no knowledge on whether the hosts' supporting services are running correctly, such as the *Network* or *Storage* services running on other nodes supporting their VMs' hosts. On the other hand, in the Cloud Verifier (CV) [58] or the Trusted Compute Pool [69], the central attestation service attests the integrity of a cloud infrastructure's entire trusted zone, without giving the customers the information of the detailed properties for each of their VMs. Blurred trust sheds a light on a new era where the CSPs' authorities can be challenged, and their powers controlled. But the lack of effective and practical models and technologies still allow the CSPs to rule customers' activities inside the cloud.

From a sociological perspective, people's rights are better protected when the powers of governance are separated and enforced by cooperative and mutual-restrictive authorities. This power is further limited when people have the rights to substitute the misbehaving parties. We observe that a similar philosophy exist in a cloud ecosystem. When a CSP has become the only authority to testify the enforcement of its own behaviours, customers have no means to verify whether the services they acquired are enforced. When a CSP has become the only authority to define the properties of its own behaviours, customers have no means to verify whether the enforced services will fulfil their requirements. Therefore, we argue that the ultimate causes of the trust issues come from the CSPs' overpower; and not until these powers are separated will customers establish effective trust in the CSPs.

In this chapter, we address the issues of breaking down the Cloud Monarchies. We present a *Separation-of-Powers* model to disaggregate the authorities of the CSP to three independent and collaborative roles. We further introduce restrictions enforced among these roles to achieve *Balance-of-Powers*. This model allows the cloud computing ecosystem to achieve the following:

1. *Trustworthiness*. The collaborations and restrictions enable effective identifications for each role's misbehaviours. Customers are also endowed with the *Freedom-of-*

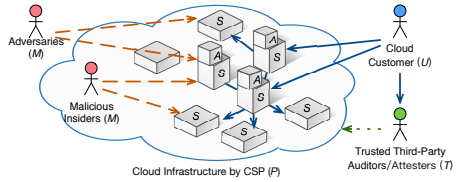


Figure 3.1: A Simplified Cloud Scenario

Choices, as they are able to substitute any role's enforcement party which does not satisfy their requirements.

2. *Openness*. CSPs with diverse scales and established credibility are all able to take part in the ecosystem, as they are all capable of obtaining effective trust from customers. *Separation-of-Powers (SoP)* also encourages two new communities for implementing the services of *behaviours inspections* and *property definitions*.

In the next section, the core *SoP* model is firstly introduced. Section 3.4 presents the Extension models, which are built on top of the *SoP*. Section 3.5 further presents a general framework for designing the *SoP*-compatible cloud infrastructure. Detailed designs and implementations of each component are introduced respectively in the following chapters.

3.2 Case Study and Motivations

Figure 3.1 depicts a simplified cloud infrastructure. Customer U deploys her cloud application A in a CSP P's cloud infrastructure. P enforces cloud services S_i to implement its SLAs with U. As U no longer has full controls over the hardware infrastructure, she needs to be assured that: 1) adequate S_i are loaded so that her SLAs are fulfilled; 2) malicious insiders or adversaries outside the cloud boundary have not tampered with both A and S_i.

3.2.1 Audited Clouds

To prove its trustworthiness, P may invite third-party auditors to evaluate its cloud implementation against various cloud audit schemes. These auditors publish evaluation reports according to P's capabilities of fulfilling certain criteria [20], so that U can make informed decisions on whether to use P's cloud services. However, concerns may still arise when U examines these evaluation certificates. First of all, these evaluations are generally too coarse-grained for making assertive judgements on the satisfactory fulfilment of specific requirements. This is because they usually consider all the services implemented by the

CSP, but U only acquires a small subset. For example, the certificate may identify P's implementations of different levels of logging services but it is not clear to U whether the level she requires is actually assigned to serve her applications.

Secondly, these audits only certify the cloud system's *implementation* status, instead of the *enforcement* status, the later of which is U's major concerns. System faults or insider attacks may result in violations of the certified service properties. For example, even if U is able to determine the employment of a logging service from P's audited certificates, she still lacks confidence of the logging service's correct enforcement during the entire life cycle of application A.

Thirdly, the certificates usually have a relatively long validity period. They only vouch for the cloud's properties at the time the evaluations are performed. As cloud systems are evolving rapidly, with new features being implemented and new vulnerabilities being identified, the acquired cloud services may have very different properties from those at the time of certification. It is hard for U to judge how the cloud infrastructure has changed since its latest evaluation.

Finally, although most schemes aim to facilitate a transparent cloud system where customers can see though its genuine properties, the auditing processes are usually opaque. U still lacks the knowledge for the detailed evaluation process. Consequently, the auditing schemes direct customers' trust towards P to new authorities: the auditing organizations. Customers could only *blindly* trust the honesty of these organizations, in the hope that the evaluations are free from enforcement faults or political scandals.

3.2.2 Attested Clouds

Trusted Computing technology aims to equip end users with tools to validate the remote software platform's genuine behaviours. It defines the *trustworthiness* of a platform as the capability of recording and reporting the genuine behaviours of all the loaded software components [26, 27, 28, 29]. This capability is implemented by embedding each platform with a secure hardware device, the TPM (Trusted Platform Module) [32], and modifying the platform bootstrapping process to genuinely record the metadata of all the loaded software to the TPM. The TPM records these metadata details in its private storage called the PCRs (Platform Configuration Registers). The *TPM_Extend* instruction (Eq. 3.1) is used to produced a chained hash to represent the integrity of a list of vales with a single hash digest. This recording process is generally referred as *measure*, and the resulting recorded values are called the *measurement values*. The *attestation* architecture is then applied to securely report all the measurement values to the remote users. These values help the users to

determine the platform's exact software configurations, which in turn imply the platform's behaviours.

$$TPM\_Extend(PCR_{New}) = SHA\text{-}1(PCR_{Old}||Input\_Digest) \tag{3.1}$$

Accordingly, several cloud attestation systems have been proposed. Trusted Cloud Computing Platform [60] enables users to attest to the IaaS provider and determine whether or not the service is secure before they launch their virtual machines. Cloud verifier (CV) [58, 70] generates integrity proofs for customers to verify the integrity and access control enforcement abilities of the cloud platform that protect the integrity of customer's application VMs in IaaS cloud. Santos et al. [71] proposed a new abstraction to let data be sealed and unsealed only by nodes whose configurations match a predefined policy. Abbadi and Ruan [72] propose the *combined chain-of-trust*, which builds a single chain-of-trust to attest to a cluster of nodes who have the exactly same configuration. Stefan et al. [73] extend the OpenAttestation to attest the IMA [74] measurement list of a node, and integrate the IMA-Appraisal to enable the Secure Boot method [75].

However, these systems all rely on a central service to attest the entire Trusted Cloud infrastructure. The cloud's immense scale will ultimately make this impractical in the production environment. Moreover, these centralized architectures create a super-power in the cloud ecosystem who is capable of dictating the trustworthiness of any cloud service but is itself very difficult to verify due to its complexity and heavy weight as it accommodates the entire cloud's attestation needs. Therefore, customers have no way to effectively determine whether this trust-establishing mechanism is honest or error-free.

In the scenario depicted in Figure 3.1, to enforce cloud attestations, P deploys the centralized attestation delegate to regularly attest every node and examine all the nodes' genuine behaviours. U then attests these delegates to ensure that they are running as expected. However, U's attestation will not examine the detailed configuration status of the cloud services she is concerned with, such as the logging service. It is also impractical for the P to allow U to inspect the attestation delegate's configurations to ensure that the logging service is attested as expected.

3.2.3 Trustworthy and Open Cloud Ecosystem: the Expectations

Accordingly, a trustworthy and open cloud ecosystem is desired, such that U will be able to:

1. determine the exact cloud services (or malicious software) that have participated in serving her cloud application A;

2. determine the exact properties of each service (or malicious software) that have been identified;

3. choose freely based on her own preference among a wide-range of third-party providers for obtaining the above information.

In the following text, we will explain how the *Separation-of-Powers* model helps U to achieve the above goals.

3.3 Models

The core *Separation-of-Powers (SoP) Model* comprises three parts: the *Role Model*, which defines three roles to achieve the *separation-of-powers*; the *Collaboration Model*, which specifies how these roles collaborate to implement trustworthy cloud computing services; and the *Restriction Model*, which enforces the mutual restrictions among these roles for achieving *balance-of-power*.

3.3.1 Role Model

Figure 3.2 depicts the Role Model. The basic idea of the *Separation-of-Powers* Model is to distribute the powers of *executive*, *judiciary* and *legislature* to three independent roles, so that they collaborate to attest the properties of the cloud services, while restricting each other's behaviours. These roles include:

- the *Cloud Services Enforcer (CSE)*, which implements cloud computing service components to meet customers' requirements;
- the *Trust Evidence Reporter (TER)*, which reports the behaviours of service components loaded by the *CSEs*;
- the *Software Property Definer (SPD)*, which serves as the authority to define the properties for each service component from its recorded behaviours.

The *CSE* is the CSP (Cloud Service Provider) with reduced powers. It implements the general cloud computing services, such as the VM hosting, highly-redundant storage, software-defined network, etc. It serves as the *executive*, whose sole responsibility is to execute the services acquired by the customers. Particularly, *CSE* includes the *Infrastructure Services Providers (ISPs)*, *Supporting Services Providers (SSP)* and *Business Logic*

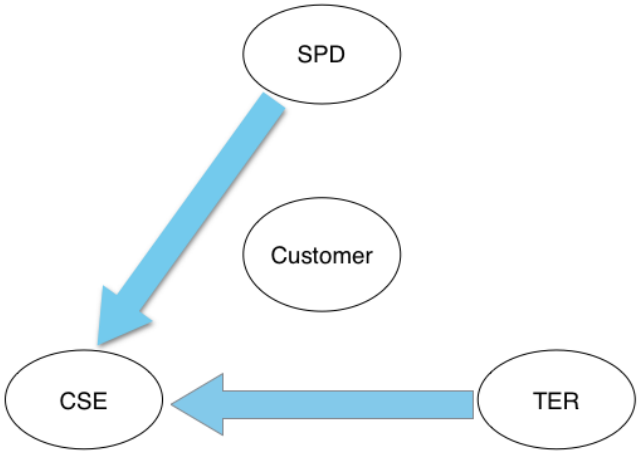


Figure 3.2: SoP Role Model. *CSE*'s powers are restricted by *TER* and *SPD*.

Providers (BLP) discussed in Section 1.1. *CSEs* may advertise their *Service Catalogue*, so that customers are able to choose among various criteria to meet their requirements. Customers reach *Service Level Agreements* (SLAs) with the *CSE*, indicating the categories and the levels of service they have purchased. Typically, a SLA is defined as a set of *properties* to be fulfilled by *CSEs*. Besides, *CSEs* also declare the *Service Manifest* (*Manifest* for short) to exhibit the behaviours of the service components they have enforced to fulfil the SLA.

However, in *SoP*, the only method for customers to effectively verify the fulfilment of the SLA is to consult the third parties, instead of examining the *Manifest* directly. Under their assistance, customers will determine: 1) whether the services in the *Manifest* are genuinely enforced; and 2) whether these services possess adequate *properties* to fulfil the SLA. Accordingly, *SoP* identifies two complementary roles to enforce *judiciary* and *legislature* respectively: the *TER* and the *SPD*.

The *TER* is effectively implemented as a *third-party attestation delegate*. It utilizes Trusted Computing technology to implement *trust evidence reporting*. This requires the cloud infrastructure maintained by the *CSE* to integrate with the Trusted Computing Infrastructure. The *TER* generates tamper-proof trust evidence recording an undeniable digital *Digest*, which represents the behaviours of the service components it perceives to have been loaded by the *CSE*. This *Digest* will attest the trustworthiness of the *Manifest* declared by the *CSE*. The separation of the *TER* and the *SPD* from the *CSP* enables fair judgements, so that the service providers will not be able to vouch for their own trustworthiness.

However, without the help from the *SPD*, the *TER* will not understand the properties represented by the *CSE*'s trust evidence. The *SPD* determines the properties of a piece

of software in general, and issues certificates to vouch for the trustworthiness of these properties. To determine the fulfilment of the SLA, customers further consult the *SPD*, who possesses the knowledge of defining the properties for each service component listed in the *Manifest*. Because the *SPD* is not aware of the detailed behaviours of any cloud component, which are gathered by the *TERs*, it will have no knowledge of how a cloud component is deployed and configured. This separation of the *SPD* and the *TER* further prevent either side to gain excessive powers. Otherwise, when a party is capable of both identifying the *CSE*'s behaviours and interpreting the properties of every behaviour, it will become a new power who can dictate the trustworthiness of any *CSE*.

The *SoP* concepts are inspired from the *Executive-Legislature-Judiciary* model in the *Political Philosophy* context. With *SoP*, the *CSE*'s behaviours are inspected by the *TER*, an independent third party. Therefore, impartial inspections can be achieved. The separation of *SPD* and *TER* further avoids *TER* from getting excess powers. Otherwise it can mandate the trustworthiness of any *CSE*, and become a new super power. However, in order to delegate authorities of *judiciary* and *legislature* to independent third parties, CSPs must open their internals. This introduces new threats, as in *SoP*, these third parties are not regarded as trusted by default.

Firstly, when the internal detail is inspected by third parties, the *CSE* will be concerned whether this exposure will encourage malicious behaviours. For example, the patch versions of the Linux kernel or critical services will facilitate targeted attacks when they are leaked to adversaries. This becomes especially critical when the *TER* is not regarded as a *trusted party* in the *SoP* model. Moreover, when malicious customers obtain the *CSE*'s infrastructure detail from the *TER*'s evidence, they may be able to implement advanced attacks.

As general customers are only concerned with the satisfaction of their SLAs, instead of the implementation and configuration detail of the cloud services, the *Manifest* should be opaque enough to allow only the *SPD* to interpret. To achieve this, *SoP* allows *SPDs* to implant software delegates, *translators*, inside the cloud infrastructure to implement information obscuring or property translations. Depending on the implementation strategy, *translators* can be as complicated as performing on-line property translations, or simply obscuring the representation of each manifest entry, so that only the *SPD* would know which property it represents when it is later required to translate.

Secondly, in order to attest to the fulfilment of a customer's SLA, the *TER* should inspect the service interaction relationship and return only the information covering all the service components that have participated in supporting the customer's application. This

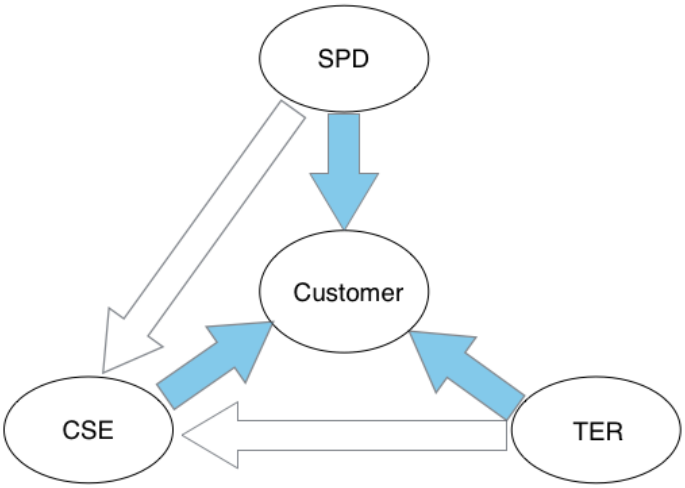


Figure 3.3: Collaboration Model.The three roles collaborate together to implement trustworthy cloud services.

requires the *TER* to implant software delegates, the *inspectors*, inside the cloud infrastructure for capturing the communication dynamics inside the cloud.

Moreover, the multi-tenancy nature of a cloud determines that excessively inspecting the cloud's internal may indirectly leak other customers' personal information. This exposure may also facilitate targeted attacks against these customers [76]. Meanwhile, too much information exposure will reduce the accuracy of the trust evidence. Therefore, it is necessary for the *TER* to hide irrelevant information and only gather enough information for testifying the *CSEs*' behaviours regarding the acquiring customers. In other words, the *TER* should only attest to the *Cloud Trusted Computing Base (cTCB)* of a target cloud application.

Thirdly, *inspectors* and *translators* require access to the cloud infrastructure's privileged controls. Insider attacks may be achieved when they behave maliciously. Therefore, their behaviours should also be inspected, and their properties should be clearly defined.

3.3.2 Collaboration Model

Fig. 3.3 depicts the collaborations among the three roles. To verify the trustworthiness of the cloud services, customers should gather information from each role respectively, including:

- an *Enforcement Digest* (or *Digest* for short) from the *TER*, recording the identities of cloud services loaded by the *CSE* for serving the target application;

- a list of *Service Manifest* (or *Manifest* for short) from the *CSE*, declaring the software composition of each cloud service;
- a list of *Property Definitions* (or *Definitions* for short) from the *SPD*, certifying the properties for each piece of service component.

The *Manifest* (M) lists the identifications of the software components (C_j) loaded by the *CSE* for constructing a cloud service (S). When employing Trusted Computing, these components are identified by their *measurement values*, i.e. the cryptography hash values ($\mathit{Hash}(C_j)$) of their executable codes or configuration files. Meanwhile, to hide the implementation details, a *Transforming Function (Tr)* enforced by the *translator* from the *SPD* must be applied to each *Hash*. This makes sure only the specified *SPD* can interpret M's entries. For example, when employing the Property-based Attestation schemes [77, 27, 26], these Transforming Functions can be implemented as the *measurement functions*, which directly translate each *Hash* to a set of *SPD*-defined properties. On the other hand, *SPDs* may simply implement the *Tr* as an encryption function, which makes sure that only the designated *SPDs* can decrypt the *Hash*. A *Manifest* is signed by the *CSE*'s key K_{CSE}^{-1}.

$$M_{SPD}(S) = \{\mathit{Tr}_{SPD}(\mathit{Hash}(C_j))\}_{K_{CSE}^{-1}} \tag{3.2}$$

The *Digest (Dig)* lists the identities of the cloud services (S_i) that the *inspectors* perceive to have participated in supporting the customer's application (A). It is signed by the *TER*'s key K_{TER}^{-1}.

$$\mathit{Dig}_{SPD}(A) = \{\mathit{id}(S_i)\}_{K_{TER}^{-1}} \tag{3.3}$$

A S_i is effectively identified by the aggregated *measurement value* of all its loaded software components. This value is calculated by the *inspectors* using the Trusted Computing facilities installed on each cloud service: by using the TPM's *TPM_Extend* instruction (Eq. 3.1) to iteratively *measure* the list of software components C_{ij} loaded by S_i. By using the *measurement values* as the components identities, the *Dig* also records each C_{ij}'s genuine behaviours.

$$\mathit{id}(S_i) = \mathit{TPM\_Extend}(\{\mathit{Hash}(C_{ij})\}) \tag{3.4}$$

The *SPD* maintains a database recording the property certificates for each software component (C_j). When it receives property querying requests, it returns the *Definition* ticket (*Def*), which binds a list of properties (*Pr*) to C_j. C_j is identified by the translated hash value ($\mathit{Tr}_{SPD}(\mathit{Hash}(C_j))$). *Def* is signed by the *SPD*'s key K_{SPD}^{-1}.

$$\mathit{Def}_{SPD}(C_j) = \langle \mathit{Tr}_{SPD}(\mathit{Hash}(C_j)), \mathit{Pr} \rangle_{K_{SPD}^{-1}} \tag{3.5}$$

With the *Collaboration Model*, a cloud user (U) verifies the trustworthiness for her cloud application's (A) supporting cloud services S_i as follows:

1. U fetches the *Dig*(A) from the *TER*'s inspectors regarding her application A; U then examines *Dig*(A)'s integrity with the *TER*'s signature;

2. For each *Dig*(A)'s entry representing a S_i's identity, U requests the *CSE* for the corresponding *SPD*-translated *Manifest* $M_{SPD}(S_i)$;

3. With each $M_{SPD}(S_i)$, U emulates the *TPM_Extend* function by iteratively hashing each entry. U then compares the final value with the *id*(S_i). This verifies whether the $M_{SPD}(S_i)$ represents the cloud service's genuine behaviours inspected by the *TER*;

4. When $M_{SPD}(S_i)$ is verified, U interprets each of its entry by requesting the *Def*(C_j) from the *SPD*. This translates the $M_{SPD}(S_i)$ to a list of properties Pr_j; U now compares Pr_j with her SLAs with the *CSE* to verify the satisfaction of her requirements.

The *Collaboration Model* forces the *CSE* to rely on two independent roles, having its behaviours inspected and defined by independent third-parties. Meanwhile, the separations of the *TERs* and the *SPDs* avoid the *CSE*'s trustworthiness from being maliciously arbitrated. In the *SoP*, the *TERs* could only obtain an opaque aggregated hash value, without the knowledge to understand how a cloud application is constructions. On the other hand, the *SPDs* could only be aware of a single software component's properties, without the capability to interpret how a cloud service is assembled. Therefore, without *CSEs*' *Manifests*, the *TERs* and the *SPDs* could not assemble enough information to peek into the cloud's internals. This gives the *CSEs* the opportunities to enforce access controls, and only reveal its properties to the most relevant parties: the customers. Moreover, the *SoP* allows more restrictions to be enforced between the *TERs* and the *SPDs*, so that their genuine behaviours can be examined. This is achieved by the *Restriction Models* explained next.

3.3.3 Restriction Model

The *Collaboration Model* forces the *CSE* to rely on two independent roles, having its behaviours inspected and defined by them. To achieve *balance-of-powers*, *Restriction Models* are designed to prevent *TERs* and *SPDs* from gaining excess powers.

Two threats are considered: *Malfunction* and *Conspiracy*. *Malfunction* denotes the malicious behaviours or accidental faults of a single role. As neither *TER* nor *SPD* is regarded trustworthy by default, their *malfunctions* may interpret the *CSE*'s good behaviours as bad, or vice versa. Moreover, as both *TER* and *SPD* install software delegates inside the *CSE*'s

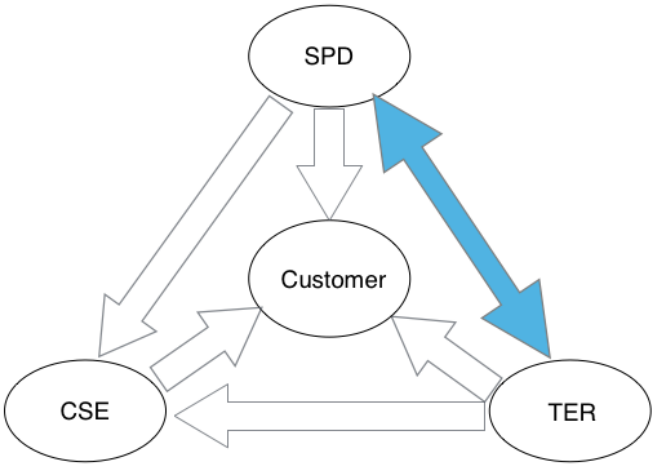


Figure 3.4: SoP Restriction Model with Mutual-Inspection. The *TER* and the *SPD* restrict each other's powers.

infrastructure, their misbehaviours may further facilitate insider attacks. Accordingly, *SoP* introduces two restriction models: *Mutual-Inspections* and *Multiple-Parties*.

Mutual-Inspection is the restriction the *TER* and the *SPD* enforce on each other (Figure 3.4). The *SPD* defines the properties of the *inspector*. Therefore, the *CSE* is able to examine which of the *inspectors* running on its infrastructure are behaving honestly. On the other hand, the *CSE* inspects the genuine enforcements of the *translator*. Every time when an attestation is performed, the measurements of the *translator* are returned. Therefore, both the *CSE* and the customers will know whether the *translators* are correctly running.

However, malicious *TERs* may still manipulate the *Digest* to substitute the correct measurement value for their misbehaving *inspectors*. Dishonest *SPDs* may also forge false *Definition* to assign expected properties to their malicious *translators*. This is mitigated by introducing the *Multiple-Parties*.

Multiple-Parties indicates that a *CSE* is allowed to employ multiple *Enforcement Parties* (or *Parties* for short) for each role (Figure 3.5). Accordingly, a *CSE* may have multiple *SPDs* to define the properties for its software components. It may also employ multiple *TERs* to inspect its enforced services. In general, these parties are independently operating organizations. Therefore, this rule introduces competitions and restrictions among the parties of a same role. It further reduces the chances of hidden dishonesty. On the assumption that most of the enforcement parties are willing to produce genuine services, by comparing results among the parties of a same role, adversaries are not difficult to identify. Moreover, the measurement values of an *inspector* of a *TER* will be inspected by the *inspectors* of other *TERs*. And the properties of a *translator* will also be defined by multiple *SPDs*.

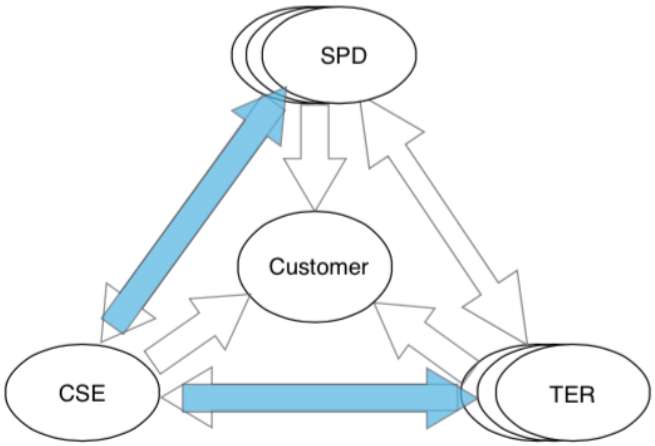


Figure 3.5: SoP Restriction Model with Multiple-Parties. The *CSE* restricts the *TER* and the *SPD*'s powers by enforcing reverse controls.

Conspiracy indicates the situation when two roles maliciously collaborate together to circumvent the *SoP* rules. Conspiracy between *TER* and *CSE* will allow *CSE* to allege arbitrary behaviours to meet the SLAs. Meanwhile, conspiracy between *SPD* and *CSE* will allow *CSE*'s misbehaviours be interpreted as satisfying the SLA, even if they are genuinely recorded by the *TER*. Finally, conspiracy among all three roles will ultimately devolve the model to the existing monarchical CSP, who mandates every aspect in the cloud computing ecosystem.

This is mitigated by the *Freedom-of-Choice* from the customers (Figure 3.6). In *SoP*, customers no longer solely choose a *CSE* for their application; the combination of *CSE-TER-SPD* is chosen instead. Any enforcement party regarded as less trustworthy will result in the whole combination being discarded. Moreover, the *Multiple-Parties* allows customers to employ multiple *TERs* or *SPDs* to verifying the trustworthiness of a *CSE*. As a result, *CSEs* are driven to employ *TERs* and *SPDs* who are more reputable than others. They are also encouraged to choose multiple *TERs* and *SPDs* to enrich the range of *CSE-TER-SPD* combinations, so that they may attract a wider scope of customers.

3.3.4 Discussion

The major difference of the *Collaboration Model* from the current Trusted Cloud proposals is that the *attestation delegate* is implemented by multiple third parties, instead of by the *CSEs* themselves. Although remote attestations promise to genuinely report the behaviours of a computing platform, the gigantic scale and dynamic nature of a cloud infrastructure make it difficult to attest to a fine-grained *cTCB* for a given cloud application. With current

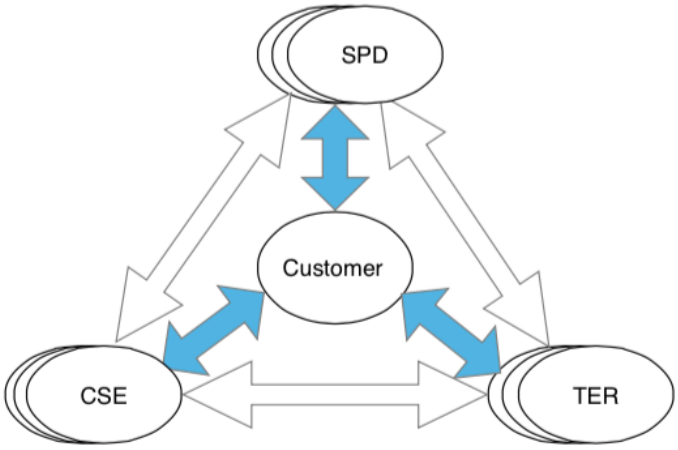


Figure 3.6: SoP Freedom-of-Choice. Customers gain the ultimate powers to choose trustworthy Cloud Service Providers.

solutions, this *cTCB* is attested to by the delegates implemented by the CSPs. In this case, cloud customers can only attest to whether this delegate is genuine enforced, without the capabilities of verifying whether they are attesting to the right *cTCB* for their applications. This results in the *Blurred Trust* we discussed in Section 3.1. With the *Collaboration Model*, the third party delegates run independently from the cloud infrastructure. *CSEs* will have no control over how these delegates perceive the internal dynamics of the cloud. This structure therefore encourages impartial and effective trust evidence reporting.

The *Restriction Model* further introduces diversities to the cloud ecosystem. With the restriction controls among different roles and different parties of a same role, malfunctions and conspiracies will be effectively identified. It further enables fair treatments to every party, as by following the rules, any party is able to gain equal trust from customers, regardless of its scale or established credibility. This encourages more service providers with diverse capabilities to participate in the cloud ecosystem. Meanwhile, the verifiable trustworthiness ultimately encourages a much wider range of customers to adopt the cloud computing model.

3.4 Extensions

Existing security mechanisms and cloud auditing schemes can benefit from the trust evidence produced by the *SoP* model. New security protection can also be implemented based on this paradigm. On the other hand, when the cloud dynamics are effectively understood

and monitored, risks inside the cloud can be evaluated, and damages can be effectively identified and compensated. Therefore risk models can be designed to aid customers for making more informed decisions when migrating to the cloud paradigm. Moreover, reputation models can also be designed to assign each participant in the cloud ecosystem with reputation values, based on its past interactions with the others. This also provides new criteria to evaluate the trustworthiness of each party in the *SoP*. This section introduces these extension models, and we leave the detailed designs and implementations to future directions.

3.4.1 Security Mechanisms

Existing cloud security mechanisms lack a well-defined root of trust. When customers are sceptical of the genuine enforcement of cloud services, they would still question the trustworthiness of the security mechanisms, as these mechanisms are also implemented as cloud services. Therefore, with their genuine behaviours verified, these security mechanisms are able to provide more credible and diverse services.

According to the cloud model in Section 1.1, the security enhancement components will be regarded as the *cTCB* of a target application, as long as they have interacted with the application's component. Therefore, their behaviours will be recorded in the *Digest*, and their properties will be attested to by the *Collaboration Model*. Therefore, when customers attest to their applications, the genuine enforcements of these components will also be examined. For example, when customers applications are protected by an Intrusion Detection System (IDS) deployed in the cloud, the IDS's enforcements will be regarded as part of the SLAs, and will be examined through cloud attestations.

On the other hand, verifiable trustworthiness will support new security controls. Firstly, as *CSEs* are able to verify the properties of the security mechanisms, they can open their privileged interfaces to third party cloud security providers, so that more powerful security services can be built. For example, VM Introspection (VMI) [24] interfaces can be exposed to third parties to implement diverse VM introspections and build advanced security controls.

Secondly, security services can further bind the *Digests* with the artefacts they produced. This allows customers to evaluate the execution environment of those services when consuming the artefacts. Equation 3.6 presents a *Trusted Artefact* composition, with which the artefact is signed by a key binding to the properties of the enforcement environment. This is achieved by *sealing* [31] the key to the specified *Digest*, so that it can only be loaded when the signing platform has the particular properties. As in Equation 3.6, K_{Digest} is bound to a *Digest* D, and this binding is signed by the *TER* with its K_{TER}. Therefore, from the

trusted artefact, customers can further determine the enforcement properties of the security services when they receive the artefacts from them. For example, the IDS binds either the monitoring logs or the incident reports, which record the past networking behaviours or indicate potential attacks, with necessary trust evidence gathered from the trusted cloud infrastructure.

$$TA = \langle \{Artifact\}_{K_{Digest}}, Cert(K_{Digest}, D)_{K_{TER}} \rangle \tag{3.6}$$

Thirdly, new security mechanisms can be built based on the trust evidence. As the trust evidence reflects genuine properties, it can aid the security-related decision-making. Equation 3.7 gives a model for utilizing the *Digest* for making policy decisions. It defines the *Actions* to enforce when a target *Resource*'s hosting platform is in the state represented by the particular *Digest*. For example, cloud based anti-virus systems can implement virus scanning based on the trust evidence.

$$P = \langle Res_i, D, \{Actions\} \rangle \tag{3.7}$$

Moreover, the trust semantics can also be exported to cloud applications to implement fine-grained trustworthy communications. For example, trusted channels [78] can be built to allow an application's VM to only communicate with other VMs when they possess particular properties. Trusted provenance [79] can also be built to record the data generation lineage, and achieve trustworthy big data processing.

3.4.2 Cloud Auditing

In *SoP*, auditing authorities will act as the *TERs* or the *SPDs* to interact with the *CSEs* and the cloud customers. *SoP* provides support for these authorities to inspect the *CSEs*' conformance to the mandatory criteria. Criteria conformance can be expressed by attestations to the software components fulfilling the criteria. Therefore, with the support from the attestation infrastructure, not only can the complexity and efforts for implementing auditing be greatly reduced, the evaluation frequency can also be increased.

On the other hand, with the trustworthy and open infrastructure, the credibility of the auditing process is also increased. Moreover, customers are still free to choose other third party *TERs* or *SPDs* to implement the trust services. Customers are able to better evaluate risks or benefits when they choose among the *TERs* or the *SPDs*. We believe that the *Freedom-to-Choose* will ultimately facilitate an open and trustworthy cloud computing ecosystem.

3.4.3 Reputation Models

SoP is not designed with a centralized *Trusted Third Party*. The authorities for defining, executing and inspecting the cloud services are delegated to different roles. Multiple instances and mutual inspections are further designed to achieve *balance-of-power*. The trustworthiness of the overall ecosystem is achieved by its peer-to-peer based collaborating and restricting the communication related to all its interacting entities, i.e. the customers, the *CSEs*, the *TERs*, and the *SPDs*.

Therefore, participants in each interactions can rate a satisfactory value based on various criteria, and reputation models can be designed to calculate a credibility values for each entity based on the aggregated ratings for its past behaviours.

$$\begin{aligned} \textit{Rep}(E_i) = F_{\textit{Agg}}(F_{\textit{Eval}}(E_j, E_i) \cdot \textit{Cred}(E_j) \cdot F_{\textit{Rela}}(E_i, E_j)) \\ , where E_j \in \{\textit{Peer}(E_i)\} \end{aligned} \tag{3.8}$$

Equation 3.8 presents a reputation calculation framework. We define any participant in the cloud ecosystem as entity E. This includes the customers, the *CSEs*, the *SPDs* and the *TERs*. Security service providers are categorized as the *CSEs* according to our CSP model. Existing society or community authorities participate in the ecosystem as *TERs* or *SPDs*. In *SoP*, these authorities evaluate and certify trustworthy behaviours through the same interfaces as all the other *SPDs* or *TERs*. They are also evaluated by their interacting peers.

We define an *evaluation* function $F_{\textit{Eval}}$ to calculate a trustworthiness value of entity E_i based on its past interactions with entity E_j. To defend against various classic attacks on reputation models, the *credibility* of E_j ($\textit{Cred}(E_j)$), and its relationship with E_i are considered. The relationship is calculated by function $F_{\textit{Rela}}$, which determines the closeness of the two entities based on their past behaviours in common. The reputation value (*Rep*) of an entity (E_i) is then calculated by aggregating all the evaluations of its past interaction peers ($\textit{Peer}(E_i)$) with the function $F_{\textit{Agg}}$.

Reputations can be assigned to each party to encourage their genuine and high-quality services. Reputations can also serve as critical criteria when customers choose the *CSE-TER-SPD* combinations, and other supporting parties, e.g. the ones introduced by the extension models above.

3.4.4 Risk Models

We argue the main cause for most customers' hesitations to switch to the cloud model is the cloud's opaqueness. As discussed, currently, customers have no effective ways for

identifying the threats inside the cloud against their digital assets. Therefore, they cannot evaluate the risks for migrating their applications or data to the cloud, nor can they get compensation when the actual damage has been discovered.

With the trust evidence implemented by *SoP*, the detailed dynamics inside the cloud can be well understood. Whether each entry in the SLA is achieved can be effectively determined, and continuously monitored. This allows the customers to appeal for compensation when the actual damages have been discovered. Insurance services can also be implemented to facilitate the compensation procedure. We argue that correctly designed compensation models, which help customers to offset their damages in digital assets with financial compensation, can increase customers' confidence towards the cloud model and reduce their security concerns.

3.5 Framework

In this section we discuss the design of a *SoP*-compatible cloud system. We first present the design principles to fulfil the *SoP* models. A reference implementation framework is then proposed, along with an introduction to existing building blocks.

3.5.1 Design Principles

We propose three design principles to implement a *SoP*-compatible cloud infrastructure:

1. *Fine-grained Cloud TCB (cTCB) identification.* The cloud infrastructure should support to the identification of an application's *cTCB* with affordable effort. This *cTCB* should include all the components to fulfil customethe r's SLA [80], and all the components that have the potential to launch malicious behaviours.

2. *Scalable Cloud Attestation.* The cloud infrastructure should support third-party *inspectors* to gather trust evidence inside the cloud and attest to an application's cloud TCB. Customers should be able to choose freely among available *TERs*, and obtain the same attestation results. Moreover, the behaviours of the inspectors themselves should be inspected by either the *CSE* or the other *TERs*.

3. *Flexible Property Translations.* The cloud infrastructure should allow *translators* to hide the identities of the software components before they are recorded and later reported to the customers by the Trusted Computing infrastructure. Customers should

be able to choose freely among available *SPDs* for interpreting the properties of the software components they are concerned of.

These three principles represent the responsibilities carried out by the three *SoP* roles respectively. Chapter 4 presents the *Decentralised Attestation (DA)*. DA helps the *CSEs* to gather the *cTCB* information for a target cloud application. It maintains sufficient information to explain how each of their services behave. This information is provided as the *Manifest*.

In Chapter 5 and Chapter 6, the *Cloud Root-of-Trust* and *Cloud Chain-of-Trust* abstractions are defined. They support the *TERs* to inspect the dynamics inside the cloud infrastructure and gather information for the attesting to the *cTCB* of the target application. This information is provided to customers as the *Digest*.

The *SPD* takes responsibilities to hide the detailed identities of the software components before they are recorded by the *TER*'s *inspectors*. It also translates the obscured information to properties. We will discuss the existing technology for implementing the property translations next.

3.5.2 Reference Framework

In our reference framework (Fig. 3.7), each cloud node is equipped with a TPM. *CSEs* maintain the *Manifest* for each cloud service. The *Manifest Querying Service* is implemented to return the *Manifest* for interpreting a given *Digest*'s entry. When considering multiple *SPDs*, only the translated version of *Manifest* should be returned. The *Inspector Layer* supports the *inspectors* implemented by *TERs*, and the *Translator Layer* hosts the *translators* from different *SPDs*.

Translators translate the measurement values of each software component's executable codes before they are *measured* into the TPMs or vTPMs (virtual TPMs). For example, when the *Compute* service is loaded on an OpenStack cloud node, the hash values of the service's executable, supporting libraries, and configuration files will be calculated. These values are then used to calculate a translated version by each supported *translators*, before they are *extended* into the TPM or vTPM. To assist the translators, the *Translation Proxies* are installed inside the cloud, so that property translation computations are delegated from each node. This reduces the TCB of each node, and eliminates redundant computations. Translation Proxies ultimately report to the *Definition Querying Services* implemented by *SPDs* outside the cloud boundary.

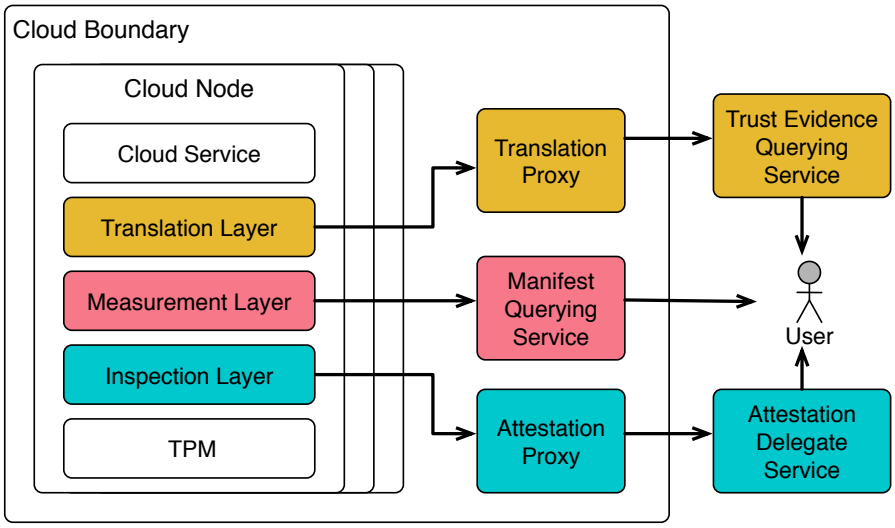


Figure 3.7: Reference Implementation Framework

Inspectors measure all the upper-layer services, including the *translators*. They also determine the dependency among cloud services hosted on different nodes. This is implemented by inspecting the upper-layer services' interactions and sharing this knowledge with other inspectors. This collaboration helps construct a communication topology, so that the dependency among the cloud services are deduced, along with the *Cloud Trusted Computing Base* for a cloud application [80, 81]. TERs deploy the *Attestation Delegate* inside the cloud, which will harvest the information possessed by individual inspectors and construct the *Digest* for a target application. The delegate then reports to the *Digest Querying Service* to provide trust evidence reporting service. Moreover, inspectors also examine the behaviours of its interacting peers, so that malicious ones will be identified. This implements the *Restriction Model*.

3.5.3 Building Blocks

In this thesis, we focus on how to design and implement the *inspector* to effectively attest to the services inside the cloud. This will be illustrated in detail in the following chapters. In this subsection, we briefly discuss existing research on defining and determining the properties of a given software component. We leave the integration of these system into the reference framework to future work.

3.5.3.1 Property Definitions

Determining the security properties of a software system has been a long existing subject. Trusted Computer System Evaluation Criteria (TCSEC) [82] defines sets of basic requirements for assessing the effectiveness of computer security controls built into a computer system. It categorizes different levels of requirements into various divisions and classes, which represent the strength levels of the security controls in a computer system. TCSEC is a part of a large set of criteria, the "Rainbow Series" [83], which defines a series of guidelines covering much wider aspects on evaluating the security aspects of various computing systems.

Many of the standards in the "Rainbow Series" have influenced, and have been superseded by the Common Criteria [84]. The Common Criteria is a framework that provides assurance that the process of specification, implementation and evaluation of a computer security product has been conducted in a rigorous and standard and repeatable manner at a level that is commensurate with the target environment for use. It is used as the basis for a US Government driven certification scheme and typically evaluations are conducted for the use of Federal Government agencies and critical infrastructure.

These criteria are used to evaluate the implementation of a software system, and certify property bindings to the implementation instances, which are usually represented by their executable codes. Equation 3.9 defines a general property certificate issued by the *SPD*. Based on these property certificates, Trusted Computing technologies employ *Root-of-Trust-for-Measurement* and *Root-of-Trust-for-Reporting* to genuinely record and report the executable loaded on a target system. Therefore, examining the property-executable bindings will help an attester to determine the target platform's properties. Property-based attestation methods further propose to deduce the properties from an executable before they are measured. The properties are measured and reported instead, so that the implementation detail of a target platform is hidden. A *white-list* may be maintained by the attesters, which is used to map the binary hashes of software components to their representing properties.

$$Cert_{Exe}(SC) = \langle \{Hash(exe_i)\}, \{Property\ Description\} \rangle_{K_{SPD}} \tag{3.9}$$

For customer-built software, a large number of different versions of executable codes may represent a same set of properties. Differences in compilers, compiling or target environment, and compiling arguments may all contribute to the discrepancies in the final executable representations. This imposes significant difficulties for mapping the executable to properties.

One line of the author's previous work proposed a *Source-code Oriented Attestation* scheme (SCOBA) [85]. SCOBA builds a Trusted Building System (TBS), which collects

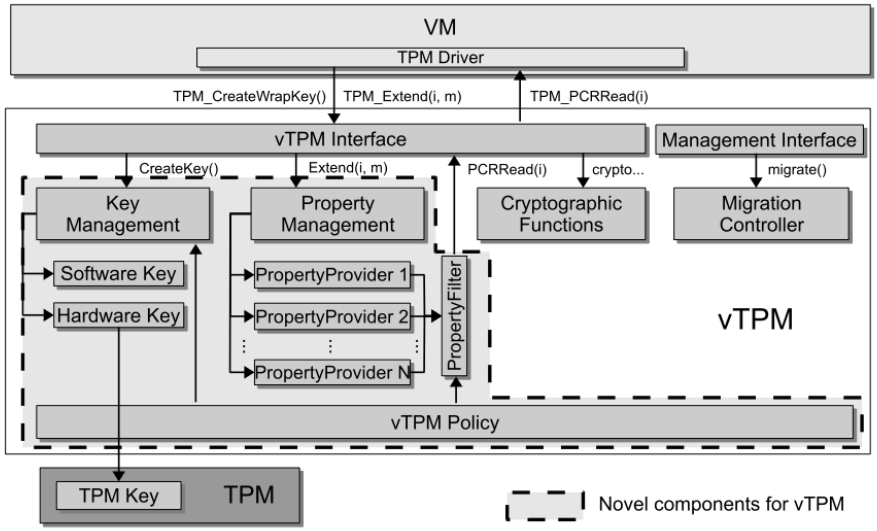


Figure 3.8: Property-based vTPM Framework

provenance data to link the final executable codes to its origin source codes and a set of compiling profiles. By further linking the properties with the source codes and the compiling profile, attesters can ultimately determine the customer-built binaries' properties.

Therefore, SCOBA opens opportunities for attesting to configurable open source software, which has become big building blocks for prevalent IT infrastructures. Moreover, the open source paradigm also allows more organizations to participate in defining the properties of software. *SPDs* of different scales may all be able to evaluate their properties and issue *Definition* certificates.

3.5.3.2 Property-based vTPM Framework

A property-based vTPM Framework has been proposed by Sadeghi et al. [77]. It supports third parties to install *Measurement Functions* in the vTPMs to perform *property translations* before each measurement value is *extended* into the vPCRs. Accordingly this framework can be applied to support the *translators*.

Figure 3.8 (from [77]) depicts the general design of this framework. Each vTPM maintains an array of vPCR sets, with each set recording the property-based measurement values produced by each *Property Provider*. When a measurement value (m) of a software component is taken inside a VM, it is firstly *translated* by each measurement function. The translated values are then *extended* into the corresponding set of vPCRs. This framework fulfils the purposes of the *SoP* model, with the *SPDs* instantiated as the Property Providers, and the *translators* implemented as the *measurement functions*.

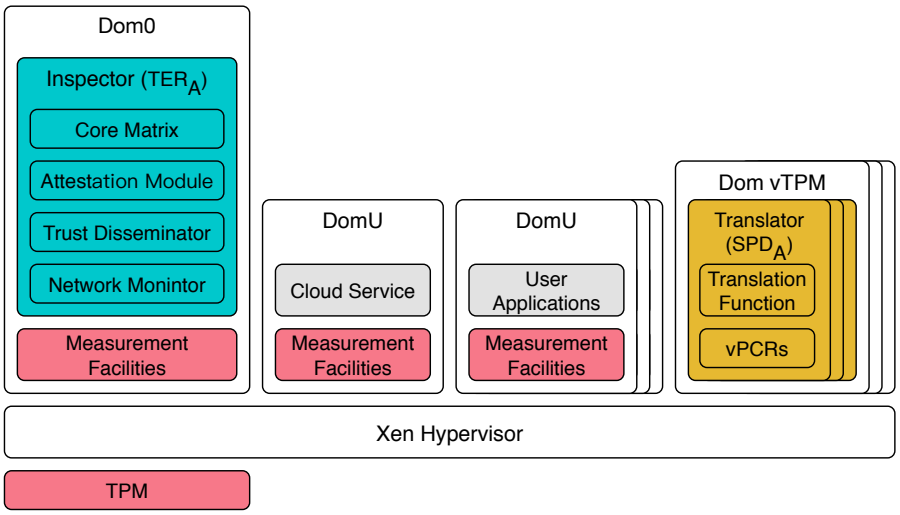


Figure 3.9: Preliminary Implementation Design with the XenServer-OpenStack deployment.

However, the current property-based vTPM proposal suggests to implement the property translations inside each vTPM. This will result in immense redundancy. The management is also complicated, as multiple copies of the same translators have to be maintained. It also complicates the vTPM manager's implementation, and enlarges the attack surface. In our reference framework, we propose to delegate the actual translating operations to the *Translation Delegate*, which is deployed as cloud services. Each translator initiates secure protocols to request the delegate to provide translations before the measurement values are recorded into the vTPMs. We leave further investigations in this direction to our future work.

3.5.4 Preliminary Implementation Design

Fig. 3.9 depicts a preliminary implementation design for a *SoP*-compatible cloud system. We will present the more implementations and evaluations in the following chapters. This design is based on the XenServer-OpenStack architecture [30]. All cloud services are deployed in DomUs, the less privileged VMs in the Xen's architecture. This deployment model reduces the cloud services' privilege and facilitates more effective service interaction inspections [81]. Measurement facilities are deployed inside each DomU to measure its *chain-of-trust*. Detailed constructions can be found in [73].

Each VM has an associated vTPM instance running in a separated *Device DomU* [86]. Each vTPM hosts the set of *translators* for the supported *SPDs*. Each *translator* maintains an array of vPCRs and implements a *translation function*. The *translators* interact with the

Translation Delegate from the same *SPD* to get assistance with the property translation or obscuring. This vTPM can be implemented by extending the TPM emulator [87].

The *inspector* is hosted in Dom0, the Xen's privileged VM, as it needs the privilege to inspect the network interactions among the DomUs [81]. The *Core Matrix* (or *Core*) stores the trust evaluations of the node's neighbours. It is maintained by the *Attestation Module* and the *Trust Disseminator Module*. The *Network Monitor* intercepts the communications among the DomUs and queries the *Core* for whether the communication target host have recently been attested. *Attestation Module* then attests the neighbouring nodes when necessary, and updates the *Core* with the gathered attestation tickets. Afterwards, *Trust Propagation Module* exchanges the updated *Core* with the neighbouring nodes by enforcing the RepCloud protocols [80].

When a CSP P constructs a cloud service VM S_i, it records a *white-list* for all the software components C_{ij} that are to be load in order in S_i. This *white-list* becomes S_i's *Manifest*. When S_i is deployed, a *Dom vTPM* is firstly instantiated and preloaded with the supported *SPDs*' *translator*. Afterwards, all the loaded C_{ij} inside S_i will be measured by S_i's *measurement facilities* and translated by each *translator*. When S_i communicates to its collaborating peers, the *inspectors* in the Dom0 initiates the RepCloud protocols to collect and disseminate the attestation tickets for building the *web-of-trust*. These tickets will include *measurement values* for the Xen hypervisor, the Dom0, S_i and its Dom vTPM. Therefore, the integrity of the *translators* and the *inspector* will be measured. The cloud service DomU and its Dom vTPM will also be measured, as they constitute the cloud node's TCB.

When attesting the cloud services, the customer U first chooses a *TER* she trusts and consults the *Digest Querying Service (DQS)*. The *DQS* contacts the corresponding *Attestation Delegate* who will in turn interrogate the appropriate *inspectors* to gather enough information for constructing the *web-of-trust*. From this *web-of-trust*, the *DQS* ultimately constructs the *Digest* for the target application's *cTCB*. Obtaining the *Digest*, U further requests P to return a *Manifest* for each identified cloud service. For each *Manifest* entry, P requests the *SPD* for the *Definition* certificate. As the measurement values for the *inspectors* and *translators* are also recorded, their behaviours are evaluated. Moreover, U may consult more supported *TER-SPD* combinations for different *Digest* and *Definition* versions. This give U the chance to challenge the authorities of a particular *TER* or *SPD*.

3.6 Summary

In this chapter, we designed a Separation-of-Powers model to support a trustworthy and open cloud computing ecosystem. We identified three roles, which cooperate together to implement trustworthy cloud computing semantics. Restrictions are also enforced to prevent any role from gaining extra powers and performing malicious behaviours. In order to achieve the *SoP* model, we first proposed the design principles and further designed the reference framework for implementing a *SoP*-compatible cloud system. We also identified critical components for implementing the framework. In the following chapters, we will present the design and implementation of these critical building blocks. We will also revisit the *SoP* model, and explain how it works with our prototype in the next chapter.

Chapter 4

Cloud Trusted Computing Base

4.1 Overview

In order to achieve effective attestations to the cloud infrastructure's behaviours, several authors have proposed to integrate cloud systems with the Trusted Computing infrastructure [60, 58]. Cloud *nodes* (i.e. physical servers) are equipped with built-in TPMs. A Centralized *attestation delegate* is implemented to gather the TPM-generated trust evidence from each node to attest to the properties of its hosted services. Currently, these schemes mostly assume the homogeneous property distributions among the cloud nodes [88]. For example, whether the entire cloud management nodes enforce ubiquitous non-discriminative VM scheduling policies, or whether all the compute nodes have the capabilities to enforce strong VM-isolation.

However, we observe that the cloud's nature of *complexity*, *heterogeneity*, and *dynamics* will hamper the effectiveness of these remote attestation schemes, especially when considering the attestations to the cloud service dependency of large-scale cloud applications.

- ***Complexity.*** Typical IaaS cloud systems have a huge amount of physical machine (or *nodes*), but only a small number of them will take part in supporting the life cycle of a cloud application. For example, as we discussed in Chapter 1, these nodes usually include the *Computes* for hosting the customers' VMs, along with other *Infrastructural Services*, such as the *Scheduler* nodes, the *Network* nodes and the *Storage* nodes. Understanding and attesting to these detailed dependency for every cloud application requires tremendous efforts. It is therefore difficult for the centralized delegate to identify all of these nodes for a target cloud application, in order to perform meaningful attestations. Therefore, many centralized attestation schemes (Section 2.3) generate trust evidence to vouch for the trustworthiness of the entire cloud. But this

is often too coarse-grained to be convincing, e.g. a single certificate declaring that the entire Amazon Web Service (AWS) [89] is trustworthy.

Some other trusted cloud approaches circumvent this difficulty by only allowing customers to attest to the properties of their VMs' hosts. However, as discussed, a VM's behaviours depend not only on its host, but also on the supporting facilities in a cloud, e.g. the VM's Storage or Networking capabilities are usually implemented on separated nodes [90]. Another line of approaches hide this dependency complexity by allocating the entire customer's cloud application in a separated zone, and vouching for the trustworthiness of all the services in the entire zone. For example, the *nova-verify* service in [70] attests to the properties of an entire Openstack scheduling domain. However, the evidence for this trustworthiness is still too coarse-grained to provide useful information.

- ***Dynamism.*** *Attestation Tickets* generated by the Trusted Computing infrastructure can only indicate the trustworthiness of a platform up to the time when the tickets are generated (Section 2.1). As the properties of a node are liable to change, attestations should be performed regularly. Moreover, VMs are migrating among different nodes, and these migrations are transparent to customers. Therefore, their dependency inside the cloud is also dynamic. This intensifies the difficulties of the dependency identification and attestation.

  Consequently, it is difficult for the currently central attestation delegate to keep track of the dynamism for each cloud application's dependency. It is also infeasible for the cloud to update the detailed locations of the VMs' hosts and all the potential supporting cloud services to customers, in order to allow them to verify their properties. This violates the cloud's flexibility and scalability achieved by the implementation detail hiding.

- ***Heterogeneity.*** *Supporting Services (SS)* and *Business Logic Services (BLS)* have become important supplemental features to a cloud system. To satisfy different security requirements, various security modules are implemented inside different parts of a cloud. Cloud applications may depend on different *SS* or *BLS*. Moreover, a same cloud service may also expose different properties in the view of different applications. This diversity therefore introduces the *heterogeneity* in the properties of different cloud applications' cloud dependency, even when they are deployed in one same cloud management domain.

For example, the traffic among the VMs of an application may be monitored by an internal firewall, while another application uses VMI-based anti-virus module provided by the cloud [5]. However, with the current schemes, it is hard for the delegates to keep track of all these dependencies, especially when considering the complexity and dynamics of the cloud. It is also very difficult for customers to determine the exact properties of their application's dependency by merely getting the knowledge of the properties of the entire cloud domain.

Consequently, these difficulties in achieving effective attestations to the cloud dependencies of cloud application call for the identification for a fine-grained *Cloud Trusted Computing Base (cTCB)* for cloud applications. Moreover, we observe that the difficulties in effectively identifying these dependency relationships come from the opaqueness of the cloud's internal communications to the attesters, e.g. the central attestation delegate. Though a cloud infrastructure is usually organized in a centralized deployment, its internal communication patterns are essentially distributed (Figure 1.2). The trust relationship among a cloud application's components and the supporting cloud services can be intuitively managed in a decentralized approach, by conforming to the services' interaction semantics. This decentralized trust management can be achieved by integrating the reputation systems [44], which are widely employed for managing and modelling trust for the *peer-to-peer* applications.

Reputation systems [44] have been designed for constructing trust relationships in large-scale distributed applications, which have highly interconnected nodes. Trust towards a node is evaluated separately by all other nodes that have interacted with it. Trust information is further disseminated and aggregated autonomously to form a global representation. As one trust relationship is managed locally, instead of by a centralized management node, the dependencies of each node are easy to identify and attest to. Moreover, in a web of nodes with a high level of connectivity and interaction frequency, corrupted nodes can still be efficiently identified.

On the other hand, as a single attestation ticket can undeniably reveal the genuine trust state of a node, it can be effectively and efficiently disseminated in the cloud with the reputation systems. Hence, by performing attestations among nodes inside the cloud based on their interactions, and then disseminating the attestation tickets, redundant attestation efforts can be saved. Moreover, from the mutual-attestation relationship among nodes, a global *web-of-trust* can be constructed. We observe that in the web with nodes regularly attesting to each other and exchanging newly generated tickets, corrupted nodes can be quickly identified. With the communication semantics maintained by the web, the dependency of an application can be easily identified. With additional attestations from a third

party to this web, malicious collaborators [44] can also be discovered. Therefore, by constructing and modelling this web, we will achieve fine-grained *Cloud TCB* attestation with low management complexity.

In this chapter, we aim to build a cloud trust management framework, the *RepCloud*, which will identify and attest to the cloud service dependency for a large-scale IaaS cloud application. RepCloud integrates the Trusted Computing framework with reputation systems. In RepCloud, *Decentralized Attestations* are performed among nodes inside the cloud, based on their interactions. Attestation tickets gathered by nodes are *disseminated* among logical related nodes to reduce redundant attestations. The attestation relationship represented by the tickets are further *aggregated* to form a global *web-of-trust*. This *web-of-trust* allows the attestation delegate to understand each node's dependencies inside the cloud. Accordingly, by querying and tracing this *web-of-trust*, the central attestation delegate is able to identify the cloud service dependency of a cloud application, and gather necessary tickets to vouch for the properties of all the depending services. In particular, we made the following contributions:

1. *Decentralized Attestation Scheme.* We integrated the reputation systems in the Trusted Computing framework. We managed *TCG trust* (i.e. trust relationship defined with the Trusted Computing Infrastructure) in the cloud in a decentralized and autonomous approach, and proposed to aggregate and disseminate the attestation tickets. Our experiments showed that, with reputation systems, the TCG trust is more effectively and efficiently transmitted among interconnecting nodes. Besides, redundant attestations are significantly reduced, while a low level of *properties-change-detection* delay is still maintained.

2. *Hierarchical Cloud TCB Definitions.* We defined the concept of *Cloud TCB (cTCB)* to illustrate the cloud service dependency for a target cloud application. We defined the *cTCB* in a hierarchical structure, with the scale of the concerning nodes minimized. These definitions help to better illustrate the dependency relationship among cloud components. Moreover, we observed that by monitoring the direct communication among nodes and performing decentralized attestation, the *cTCB* is easier to determine and continuously attest to.

3. *Fine-grained Cloud TCB Attestations.* We designed a new paradigm for implementing the *cTCB* attestation, taking into consideration the interaction patterns among related nodes. *cTCB* attestations help cloud customers to determine the security

properties of the cloud dependency nodes that support or may affect the genuine functionalities of their VMs. It implements an effective and practical way for establishing trust from the customers to the cloud systems and the Cloud Service Providers.

RepCloud overcomes the deficiencies of the Centralized Attestation schemes by preserving the communication semantics of the upper-layer services. This information will help the *inspectors* to define and attest to a fine-grained *cTCB* for a cloud applications. The decentralized structure will further support the flexibility for multiple inspector implementations.

This chapter is organized as follows. Section 4.2 presents the Cloud TCB definitions. The RepCloud framework for implementing the *Decentralized Attestation* is then proposed in Section 4.3. Section 4.4 discusses how the RepCloud is utilized to support the *inspectors* and facilitate *cTCB* attestations. Evaluations and discussions on the RepCloud protocols are presented in Section 4.5.

4.2 Cloud Trusted Computing Base Definitions

We define the concepts of *Cloud Trusted Computing Base (cTCB)* at three layer:

Core TCB of a Node. Each cloud node provides essential services for constituting a cloud infrastructure. We firstly define these services as the *Core Service* of the node, which usually includes the *Infrastructure Services (IS)*, and the *Supporting Services (SS)* deployed on the node to enrich the IS. The SS or BLS deployed on the node but only service a particular cloud application are not considered as the Core Services.

The Core TCB of a node thus contains the Core Services, along with all the software components in the chain-of-trust [31] deployed on the node to bootstrap the platform into an expected state and to support the Core Services. These include the CRTM, BIOS, the bootloader, Operating System Kernel and other critical modules.

A VM relies on the virtualization layer, including the hypervisor, the management console and other privileged software modules. Components in this layer also have their own dependencies. Accordingly, all the components forming the chain-of-trust [31] up to the virtualization layer are crucial for implementing a VM. On the other hand, each cloud node implements essential services for constituting a cloud infrastructure. We define these services as the *Core Service* of the node, e.g. the *Compute* service on each VM host, or the centralized *Schedule* service. They also effect the genuine behaviours of a target VM.

Accordingly, we define the *Core TCB of a node* (*coreTCB*) to contain the *Core Services*, along with all the software components in the node's *chain-of-trust* [31] to bootstrap the platform into an expected state to host the VMs and the Core Services. Generally, these include the CRTM, BIOS, the bootloader, Operating System Kernel or hypervisor, and other critical modules, etc. On the other hand, collocated VMs may also affect the integrity of a target VM, e.g. by enforcing side-channel attacks, or exploiting the hypervisor's vulnerabilities. However, attesting to these VMs is both impractical and infeasible. Alternatively, cloud customers may determine the risks from these collocated VMs by examining the properties of the hypervisor and additional supporting services, e.g. whether strong isolation mechanisms are implemented, or violation precautions are enforced, etc.

Attesting to the *Core TCB of a node* will help cloud users to examine the integrity and security properties of their VMs' hosting layer, such as the hypervisor, managing software components, etc. This defends directly against the threats from the *malicious privileged customers*, who escape from the isolation barrier enforced by the virtualization layer, and attack the collocated VMs.

Cloud TCB of a Node. Cloud is a distributed system with a large amount of collaborating nodes. The functionalities of one node usually depend on the behaviours of the others. For example, a *Compute* node relies on the *Schedule* node to assign it with VMs; and the *Schedule* node further needs the *Image Service* node for managing the VM images. We therefore define all those nodes supporting a target node's *Core Services* as its *static dependency*. The static dependency relationship is usually defined by the cloud administrator during the deployment of the cloud infrastructure.

On the other hand, nodes in a cloud interact with each other due to the communications among their upper layer services. We define these nodes as a target node's *neighbours*. For example, VMs on a node may communicate to the VMs on the others, which will result in the communications among their hosting nodes. Moreover, we assume attacks require direct interactions: malicious behaviours in the neighbours may bring risks to the target node. Therefore, the behaviours of a node's *neighbours* may affect the node's functionalities. We define these nodes as the target node's *dynamic dependency*. In addition, *static dependency* nodes also need to communicate with the target to implement their responsibilities. We denote the nodes, serving both as *dynamic* and *static* dependency, as *active static dependency*. This prevents dishonest CSPs to deliberately specify service nodes as static dependency, but configure them to not providing services.

We therefore define the *Cloud TCB of a node* ($cTCB_{Node}$) to include the node's own *coreTCB*, the *coreTCBs* of all its dynamic dependency node, and the $cTCB_{Node}$ of all its active static dependency node. As static dependency nodes can be considered as the delegate

to implement parts of the target node's functionality, their $cTCB_{Node}$ should be considered as part of the target node's $cTCB_{Node}$. This recursive definition will ultimately build an active static dependency tree, containing all the nodes to support the target node's functionality, and all the dynamic dependency nodes which will affect the genuine behaviours of one target node and all its active static dependency.

Attesting the *Cloud TCB of a node* will help the customers to verify whether all a cloud service's depending functionalities are genuinely enforced. It also helps to identify unexpected communications to the target cloud service. Therefore, attestations at this scale defend against the threats from the *dishonest providers* and *malicious providers*, regarding a single cloud service or customer VM.

Cloud TCB of an Application. A cloud application usually contains a set of VMs, and possibly various supplemental services, such as *Supporting Services (SS)* or *Business Logic Services (BLS)*. The *Cloud TCB of an Application* ($cTCB_{App}$) is hence comprised of the $cTCB_{Node}$ of all the nodes hosting these VMs or services.

As VMs are migrating and the hosts of the additional services may also change, TCB at this scale changes dynamically during the application's life span. Attestations to this TCB thus only provide trustworthiness evidence for a specific point of time, which incorporates a static set of nodes and services. In the current work, we only focus the attestations to this TCB at a specific time, which has a static set of nodes. By integrating trusted migration protocols [60], attestation to the entire lifecycle can be achieved.

4.3 RepCloud Framework

In order to identify the *Cloud TCB of a node*, we propose to distribute the attestations effort to each cloud node. In RepCloud, nodes attest to each other according to their interaction patterns. Each node maintains a *Local Trust Vector (LTV)* recording the attestation tickets it gathered from its attestations to its neighbours. These attestations thus preserve the interaction relationship, which help identify a node's dynamic dependency. Trust relationships represented by these tickets are further aggregated to construct a global *web-of-trust*. This web is maintained by each node in its *Global Trust Metric (GTM)*. From the web-of-trust, the *attestation history* of each node can be deduced, which records all the attestations performed by other nodes *towards* the node.

The core idea of *DA* is that attestations to cloud nodes are performed on demand by other nodes based on their communication needs. As the communication patterns generally reflect the service dependency relationship, each node possesses the properties of its

depending nodes. Moreover, as nodes may have one same dependency, each node disseminates the attestation tickets it collects to its logical neighbours. This reduces the redundant attestation efforts. Moreover, a *web-of-trust* is constructed, which allows nodes to react faster to other nodes' property changes. The *web-of-trust* also helps the *attestation delegate* to understand the dependency relationship among cloud nodes. This facilitates effective and fine-grained *cTCB* attestations.

In RepCloud, each node maintains a *Local Trust Vector (LTV)* recording the attestation tickets it gathered by attesting to its communication peers, which we define as its *neighbours*. These tickets are then disseminated and aggregated iteratively among the neighbours to construct the global *web-of-trust*. A partial *web-of-trust* is maintained by each node in its *Global Trust Metric (GTM)*, as a node only concerns the properties of its *neighbours*. We will examine the formulation, calculation and dissemination of trust semantics in RepCloud respectively.

4.3.1 Notations

A node in the cloud is denoted as N_i. The attestation ticket generated by the attestation from N_i towards N_j (performed by N_i) is denoted as:

$$t_{i,j} = (pcr_{i,j}, ts_{i,j})$$

It represents the trust N_i places upon N_j. This means that N_i "trusts" that N_j has the behaviours defined by the values $pcr_{i,j}$, at the time defined by $ts_{i,j}$. $pcr_{i,j}$ is the set of the *PCR* values representing the aggregated behaviours of all the software components that have been loaded on N_j since its latest reboot. $ts_{i,j}$ is the timestamp recording the time the attestation was performed.

The *Local Trust Vector (LTV)* of N_i is hence denoted as follows. It is the list of attestation tickets t generated from all the attestations performed by N_i towards its connecting nodes.

$$L^i = [t_{i,j}], where N_j was\ attested\ by N_i$$

We further define the *Global Trust Metric (GTM)*, G, as follows. G stores all the attestation tickets generated from the attestations among all the nodes. As we will discuss next, we construct G in a distributed manner: each node maintains a local version of G, only storing the attestation tickets generated by its surrounding nodes. In the following text, superscripts are used only when necessary to denote that the data structure is maintained by the particular node. For example, the *GTM* maintained by N_a is denoted as G^a.

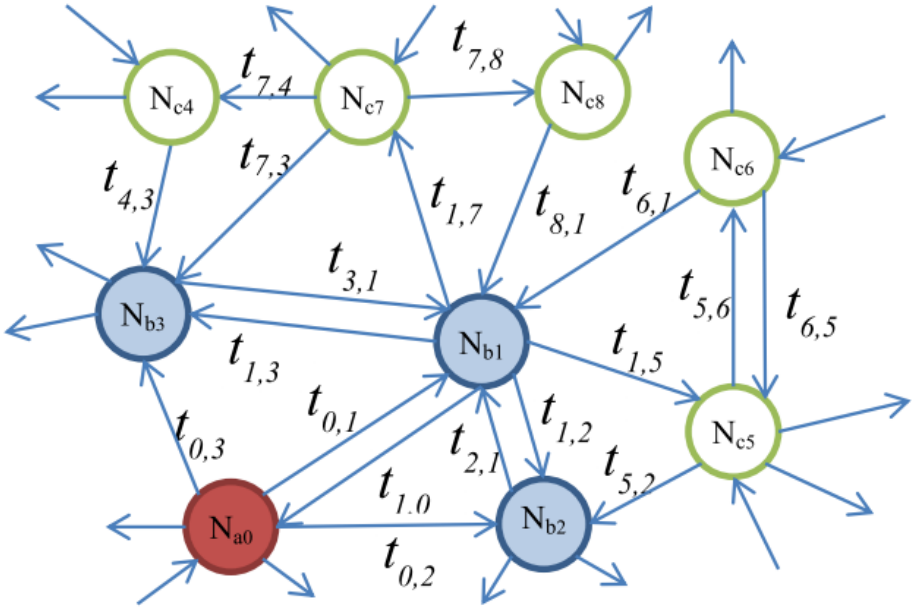


Figure 4.1: Decentralized local attestation topology.

$$G = [t_{i,j}]$$

As each node maintains only the trust information of its neighbours, only a part of its *GTM* has valid values. We define $t_{i,j} = (0, 0)$ when no attestation is performed by N_i to N_j. In other words, a node's (e.g. N_i) neighbours are the nodes that have been attested by N_i. We use Nbr^i to denote the neighbours of N_i:

$$Nbr^i = \{N_j \mid \forall j(t_{i,j} \neq 0)\}$$

We define one relationship "$>$" for comparing the freshness of the attestation tickets as below. "$\max$" and "$\min$" are hence defined in the usual way.

$$t_{i,j} > t'_{i,j} \iff ts_{i,j} > ts'_{i,j}$$

Figure 4.1 illustrates a simplified node attestation topology. Directed edges represent attestations, with their endpoints as the target nodes. Attestation tickets are specified as the weights. For better presentation, in the figure, nodes attested to by N_{a_0} are referred to as N_{b_i}. They are the neighbours of N_{a_0}, comprising the set Nbr^{a_0}. Nodes interacting with N_{b_i} but have no connection to N_{a_0} are depicted as N_{c_j}. With the two-level trust aggregation in RepCloud (Section 4.3.3), all these attestation tickets can be located in the *GTM* of N_{a_0}: G^{a_0}.

4.3.2 Local Trust Gathering

A node maintains the trust evidence of its cloud TCB in its *LTV* (Local Trust Vector) by performing attestations to its neighbours periodically. The three typical steps of a TCG attestation procedure are considered for gathering this local trust:

Measurement. Every node in the cloud model has an embedded TPM, which records the measurement of every component in its chain-of-trust. The entire core TCB is measured. Different runtime attacks exist to inject malicious codes directly into the memory of the node without being measured. However, as countermeasures [91] are being proposed and the TCG attestation framework is also evolving to mitigate these threats, we currently assume that all executables loaded on a node have been genuinely measured, and with the measurement values extended to the TPM.

Attestation. Three types of communication actions with different assumptions on trust are defined in our context: 1) ***Critical communication.*** Security-critical communication actions among nodes require immediate attestations, because previously generated attestation tickets may be obsolete. These are user-defined actions, and will trigger local attestation whenever they occur. For example, a user can define a VM storing confidential database as a critical resource, and require all the communications to this VM to trigger immediate attestations. These communications are therefore regarded as the critical communications. 2) ***General communication.*** For communication actions that do not require immediate attestation, trust evidence can still be gained as proof for service enforcement, e.g. SLA checking or provenance queries [79]. Attestations to a target node in this case can be performed a longer time ago, or by some other nodes that are trusted (attested) by the current node. 3) ***Trusted communication.*** Specific communication actions should be performed with pre-assumed trust. Otherwise they may lead to recursive attestations. These actions should be carefully designed and with limited interfaces, e.g. RepCloud protocol communications.

The Timestamped Hash-chain attestation protocol [33] is used (Section 2) in RepCloud to increase the throughput of the attestation operations and generate transmissible attestation tickets. The time of attestation is inferred from the ticket as the nonce is bound to a global time. Every time when an attestation is performed, the *PCR* value and timestamp will be stored in the *LTV* of the current node. Every attestation ticket is valid for a specific length of time, after which a new attestation should be performed. With RepCloud, as we will show in our experiments, redundant attestation effort will be significantly reduced, and only negligible trust management overhead will be introduced.

Verification. With local attestation, a node is regarded as trustworthy when the measurements of all components in its core TCB are identified in a global expected measurement list. Faulty nodes will be reported. As a node is identified with its AIK, a new AIK can be issued to the node after it is reset. It is hence regarded as a new node in RepCloud, and its previous distributed maintained attestation information will be discarded automatically when obsolete. All VMs hosted on a faulty node will be halted and the users will be informed. With RepCloud, with attestation to the core TCB, upper layer components can effectively be integrated to implement property-based [27, 26] TCB attestation and achieve heterogeneous security property architectures. The core TCB can be extended to contain the software components in the trust chain up to the hypervisor, together with an additional privileged component to introspect the trustworthiness of other privileged components and deduce their properties. These properties can then be disseminated and aggregated with our RepCloud infrastructure.

4.3.3 Global Trust Aggregation

As a node is concerned only with the trust state of its neighbours, a local part of the web-of-trust is maintained in its *GTM*. In this case, a two-level trust aggregation is enough – aggregating the trust that the neighbours' neighbours place towards the neighbours. For example, in Figure 4.1, N_{a_0} should obtain all the trust placed upon N_{b_i} by N_{c_j}. This section will present the algorithms for constructing the *GTM* and keeping it from becoming obsolete. Their overhead will be analysed in the next section.

4.3.3.1 LTV Aggregation

The *GTM* of a node is constructed by fetching and merging the *LTVs* of its neighbours immediately after attestations to them are performed. Aggregation can be implemented by simply overwriting the corresponding entries in the *GTM* with those *LTVs*:

$$G^a_{i,*} = L^i,\ \forall i(N_i \in Nbr^a)$$

Currently, we simplify the cloud model by assuming that nodes tend to interact with their nearby peers, i.e. neighbours of a node also communicate with each other, especially when VMs of a cloud application are more likely to be deployed within a same cluster in the cloud. Hence, the *GTM* of a node may also contain the trust its neighbours place upon each other. As we can see from Figure 4.1, by merging the *LTV* of N_{b_1}, N_{a_0} also gains trust information placed upon its neighbours N_{b_2} ($t_{1,2}$) and N_{b_3} ($t_{1,3}$). When considering scenarios where nodes have little common in communication peers, multi-level of trust

aggregation can be used, instead of the two-level in our case. EigenTrust [45] shows that this multi-level aggregation still has a fast convergence.

4.3.3.2 GTM Aggregation

To obtain more detailed attestation histories, the *GTMs* of a node's neighbours will be aggregated. To avoid recursively expanding the *GTM*, a Sliced Global Trust Metric (SGTM), denoted as $SG^{b,a}$, can be returned to N_a to only contain the trust information of all N_a's neighbours maintained by N_b.

When N_b receives N_a's request to return $SG^{b,a}$, it will first search its *GTM* (G^b) to identify all N_a's neighbours (Nbr^a). Then, N_b searches G^b to collect all the attestation tickets generated with each N_a's neighbour as either the attestation source or the attestation target. As we will demonstrate with our simulations, this will greatly reduce the attestation ticket dissemination costs, as only the concerned tickets are transmitted.

$$SG^{b,a} = [G^b_{i,a})] \cup [G^b_{a,i}],\ \forall i(N_i \in Nbr^a)$$

$SG^{b,a}$ is then extended into G^a. The $\max$ operator is used for updating the freshness of the trust information:

$$G^a_{i,j} = \max(G^a_{i,j}, SG^{b,a}_{i,j}),\ \forall b(N_b \in Nbr^a)$$

In Figure 4.1, as N_{a_0} is the neighbours of N_{b_1}, its *LTV* is maintained in N_{b_1}'s *GTM*. $t_{7,3}$ and $t_{5,2}$ are returned in SG^{b_1,a_0}, because N_{b_1} also maintains the *LTVs* of N_{c_7} and N_{c_5}. Moreover, as N_{b_1} is also performing *GTM* extension, $t_{4,3}$, $t_{8,1}$, and $t_{6,1}$ can also be returned. As the peers are at the same time extending their *GTMs*, trust is aggregated iteratively.

As defined, $SG^{b,a}$ also contains L^b, when $N_b \in Nbr^a$. Hence *SGTM* can be returned with each attestation instead of *LTV*. However, in the bootstrapping stage, when a node's *GTM* is mostly empty, *LTV* merging can still be used to achieve a faster convergence.

4.3.3.3 Trust Dissemination

As TCG attestation tickets are transient by nature, trust information in *GTM* will soon become obsolete. On the other hand, as nodes are attesting to each other due to occurring events, newly generated attestation tickets can be sent to the set of corresponding nodes to update the trust information maintained in their *GTMs*:

$$I^a_b = \{N_i \mid \forall i(G^a_{i,b} \neq (0,0))\} \tag{4.1}$$

I_b^a denotes the set of nodes that have also attested to N_b. As in Figure 4.1, a new attestation ticket $t_{0,1}$ from N_{a_0} to N_{b_1} is sent to the nodes that have also attested to N_{b_1}: N_{b_2}, N_{b_3}, N_{c_6}, and N_{c_8}. On receiving this ticket, those nodes first verify the signature of the AIK [31] signing the ticket, and then update $t_{0,1}$ in their *GTMs*.

4.4 Implementation

The RepCloud framework fulfils the first design principle for implementing an *SoP*-compatible cloud infrastructure (Section 3.5.1). By adapting the RepCloud, the *cTCB* of each node is identified and regularly attested. This implements the *inspector*'s main responsibilities. Based on this, *delegates* are implemented to gather the trust information and construct the *Digest* for a target application and its *cTCB*.

However, as we will discuss in the next chapter, RepCloud still lacks clear definitions for *Cloud Root-of-Trust* and *Cloud Chain-of-Trust*. This inhibits the effective application attestations for a large-scale cloud infrastructure. In this section, we first propose the design and implementation of the *inspectors* and *delegates* based on RepCloud. Further enhancements will be introduced in the following chapters.

4.4.1 Inspector Implementation

By implementing the RepCloud framework, each node carries two additional responsibilities: 1) to gather the trust evidence of its dependency nodes; and 2) to disseminate this evidence to help the other nodes to gather the evidence they need. We have implemented the RepVisor to fulfil these responsibilities.

RepVisor is deployed in each node to serve as the *inspector*. According to the *Reference Framework* in Section 3.5, each TER should implant a dedicated *inspector* on each cloud node. However, due to the homogeneity of the RepVisors, we propose the CSEs, i.e. the cloud infrastructure owner, to implement the RepVisors and allow each TER's delegates to attest their integrity.

To support the RepVisor, we used the OpenStack-XenServer [30] cloud system. Figure 4.2 depicts the XenServer-based cloud node structure. Each node is deployed with a Xen hypervisor [22]. The Dom0 is the privileged Virtual Machine (VM), which manages the underlying hardware and implements the virtualization layer management functionalities. On each node, we deployed all the *Core Services* inside the privileged space (Dom0), including the OpenStack facilities, e.g. Compute or Scheduler [90]. For example, for the

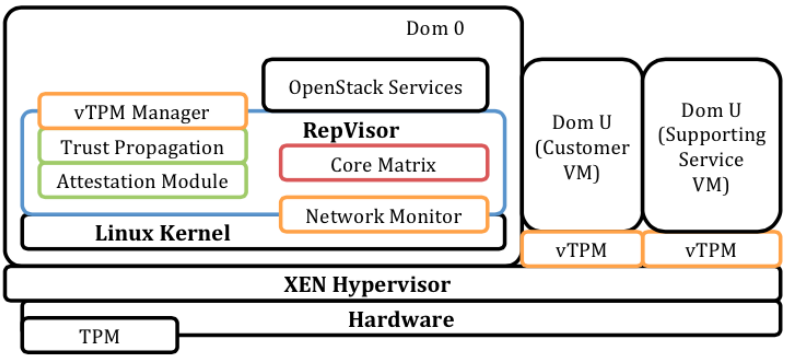


Figure 4.2: RepVisor Implementation Architecture

node serving the *Compute* role in the OpenStack infrastructure, the Dom0 runs the *Core Services* which include the OpenStack's *Compute* service and its supporting service components, such as the client-side applications for the *Network* or *Storage* services.

RepVisor is deployed inside the Dom0. It is implemented as a user-space application, except its *Network Monitor* module, which is a Linux kernel module loaded into the Dom0's kernel space. The *Trust Propagation* module implements the DA protocols to attest and disseminates the properties of the *Core TCB* of the connecting cloud node. As in our prototype, this *Core TCB* includes the entire Dom0, a core chain-of-trust [31, 74] should be measured to include the software components for bootstrapping the Dom0, together with all the service components loaded in these environments.

To measure this chain-of-trust, the XenServer's measured boot supplemental pack [30] is enabled. The IMA-enabled Linux kernel is also deployed in the Dom0 [74] to measure all the loaded software components inside. User space Trusted Computing services are also deployed, including TrouSers [92] for implementing the Trusted Computing services libraries, and the OpenPTS [93] for implementing remote attestations.

The *Core Matrix* (or Core, for short) stores the trust evaluations of the node's neighbours, particularly the *LTV* and the *GTM*. The Core is maintained by the Attestation Module and the Trust Propagation Module. The *Network Monitor* intercepts the communications among the DomUs, and queries the Core for whether the communication target hosts have recently been attested to. The *Attestation Module* then attests neighbouring nodes when necessary. It updates the Core with the gathered attestation tickets. Afterwards, the *Trust Propagation Module* exchanges the updated Core with the neighbouring nodes by enforcing the *Decentralized Attestation* protocols. We will discuss the details of these modules next.

The *vTPM Manager* module links the chains-of-trust of the VMs to the Dom0. It instantiates a vTPM for each VM. To support the translators for our future work, it will support the Property-based vTPM framework. Each vTPM should support a set of vPCRs for each supported translator. As discussed, we leave the design and implementation of the vTPM Manager to our future work. Currently, we deploy the TPM emulator to implement the vTPM, and we do not consider the property translations. The binary codes of the TPM emulator are measured by the IMA in Dom0. This implements the vTPM-TPM binding. Comprehensive research on this binding is described in [94]. We leave the integration of this work with our system to future work.

4.4.1.1 Active Attestation

We used the Timestamp Hashchain-based attestation scheme of [33] to reduce redundant attestations. Each node actively generates attestation tickets and binds them to a globally agreed timestamp. This allows a ticket to be reused by multiple attesters, while their freshness can still be verified.

In our prototype, each RepVisor is equipped with a timestamp certificate, which binds an initial nonce value and an initial timestamp. An interval value is also included, which specifies how frequently a new nonce is calculated. The *Attestation Module* thus actively *quotes* the local PCRs. It generates an attestation ticket with an updated nonce value using the nonce updating protocol in [95] (further introduced in Chapter 7):

$$T = \langle \{PCR[0-7], PCR[10], nonce\}_{K_{AIK}^{-1}}, step, M_{vPCR}, Cert\{AIK\} \rangle \tag{4.2}$$

Equation 4.2 presents the ticket T disseminated with the decentralized attestation. In our implementation PCR$[0-7]$ record the measurement values of the local chain-of-trust for bootstrapping the platform. PCR$[10]$ is used by the IMA for measuring the loaded application components, libraries and kernel modules. In addition, the *step* provides clues to deduce a timestamp from the *nonce*. M_{vPCR} refers to the vTPM matrix representing the measurement values for the Supporting Service VMs.

When the Property-based vTPM is implemented, these measurement values should include the matrix of vPCRs as proposed in [77]. These values can be gathered by installing attestation clients inside the DomU, e.g. the OpenPTS, and initiating attestation sessions to it to quote the vPCR values. However, this requires modifying every cloud service VM, and the added attestation service may also enlarge the attack surface. To reduce complexity, the Attestation Module can directly *quote* the vPCRs maintained by the vTPM Manager.

As the vTPM Manager facilities are not available to us, we leave the vPCR quoting to our future work. In our currently implementation, we omit this vPCR matrix.

When receiving attestation requests from the *Network Monitor*, the *Attestation Module* first asks the target node to return its newest attestation ticket, T. This target node is identified by its IP address provided by the *Network Monitor*. The *Attestation Module* then queries the Privacy CA to verify the AIK of the target node, and determines the freshness from the timestamp deduced from the nonce. Valid tickets will be updated to the *LTV* in the Core. We identify each node with its IP address. An *LTV* entry thus contains the IP address, the latest attestation ticket, and a deduced timestamp, which indicates the time the ticket was generated.

The *Trust Propagation Module* then enforces the DA protocols. It fetches the *LTV* and the *GTM* or the (Sliced *GTM*) of the target node, and merges them into its own *GTM*. The *GTM* records the attestation relationship. In our implementation, each entry contains the IP addresses of the attesting and the attested node, the attestation ticket, and the timestamp.

4.4.1.2 Network Monitoring

Decentralized Attestation targets are determined by the communication patterns, which are analysed by the *Network Monitor*. The *Network Monitor* intercepts all the communications of the host, and enforces the *DA* protocols when the locally maintained attestation ticket to the target node is not available or outdated.

We implemented the *Network Monitor* as a xtable-addon module: a xtable extension [96] to the Linux kernel in Dom0. Xtable monitors the communication traffic and performs user-defined addon actions when particular conditions are met. The *Network Monitor* intercepts the network traffic sending from all the DomUs. It also maintains a data structure in the kernel space to record the newest timestamp of every target node. It suspends the traffic when the timestamp indicates obsolete attestation, and outputs the IP address of the communication target node through a kernel device. The *Attestation Module* running in the user space monitors this device and performs attestations described above to the target node. This attestation also triggers the trust disseminations, and updates the Core Matrix. The resulting updated timestamps are also written to the kernel device and stored in the *Network Monitor* module.

The traffic made by the applications in the Dom0 is not intercepted by the *Network Monitor*. This includes the attestation and the trust propagation traffic. However, an exception is that the incoming attestation requests are intercepted. This is because the attestation target is determined by the IP address of the intercepted communication target, which usually is the DomU hosting the OpenStack modules or customer VMs. However, decentralized

attestations only consider the integrity of the Core Services, i.e. the Dom0 and the OpenStack domains on each node. Therefore, the *Network Monitor* intercepts all the attestation requests addressed to the Dom0's IP, and returns Dom0's integrity values (the PCR values).

4.4.1.3 Preliminary Performance Analysis

We have implemented the RepVisor. But as a large-scale physical data centre environment is not available to us, the effectiveness of the RepCloud and the *Decentralized Attestation* is not tested in a real system. In the next section, we will evaluate them with simulations, which are the general ways for evaluating the reputation-based systems.

Here we briefly discuss the RepVisor implementation's performance based on a two-node RepCloud system. Our focus is the network latency, which will be the RepVisor's the major overhead, as the network traffic is halted and the DA is enforced when the target platform's latest attestation ticket is made obsolete.

In our experiments, we run two communicating applications in two DomUs deployed in two connecting RepVisor-enabled Xen hypervisors. The Xen's network-passthrough feature was *not* enabled. Application A sends a $2G$ data file to application B over a 100*Mbps* LAN, with only the two machines attached. A then records the overall time interval for completion.

We first examined the time interval when the DA was disabled (the *without DA* in the Table 4.1). We then examined how the DA would affect the time interval. In this experiment, the DA on A's RepVisor is enabled, which will halt the traffic in every 0.6 seconds, and perform an attestation to B's entire Dom0. The reason for choosing the attention interval as 0.6 was because this is the minimum interval for consecutively quoting the PCRs of a TPM (further discussed in Section 7.3.2.2). This simulated the worse case scenario when enforcing the DA. In this experiment, the trust aggregation and dissemination mechanisms were disabled, as the RepVisor enforces them in parallel to the general traffics, so that no obvious network latency will be introduced.

Each experiment is performed 10 times, and the average results are calculated. The standard deviation (StDv) is also presented. As shown in Table 4.1 only the overhead of 2.44% was introduced. This is because of the introduction of the *Active Attestation*. As we will further discuss in Section 7.3.2.2, each *Active Attestation* only incurs the latency of 0.0016 seconds, when a node with less-frequency-changing PCR values are frequently attested. Moreover, when the trust dissemination and aggregation mechanisms are enforced, we expect the overheads to be even lower, as the attestation tickets are reused, so that the redundant attestations are greatly reduced.

Table 4.1: Average Time Intervals for Data Transfer

	Transfer Completion Time (Seconds)	StDv (Seconds)	Overheads
without DA	245	3	-
with DA	251	3	2.44%

4.4.2 Delegate Implementation

Delegates are implemented as user-space applications deployed in VMs. A *TER* may deploy multiple delegates in different compartments of a cloud, e.g. different clusters, so that it is able to cover the web-of-trust for different applications with fewer queries. When receiving attestation requests from a customer, the *TER*'s proxy contacts one of the delegates and specifies the target application's information. Different strategies can be implemented on selecting the delegate to satisfy various implementations' needs. For simplicity, our prototype chooses a delegate randomly.

When receiving the attestation requests from the proxy, the chosen delegate first locates all the attestation targets, i.e. the nodes hosting the VMs of the target application. It then constructs the *web-of-trust* by attesting to the targets and aggregating their locally maintained trust matrix, i.e. the *GTMs*. After analysing the web-of-trust, it generates and returns the *Digest*.

4.4.2.1 Locating Attestation Targets

To locate the attestation targets, the delegate queries the cloud infrastructure's central database. In this case the delegate first attest the integrity of the DBMS (the database management software), and all its supporting components, to make sure that the database is correctly managed, without unauthorised access. Accordingly, we modified OpenStack to expose its interfaces for querying the list of nodes hosting the VMs for a given cloud application. OpenStack maintains these location mappings in its central database and supports the querying through the *nova-api*, which is to allow the *nova-scheduler* to locate and deploy VMs. Access control rules are configured to expose these interfaces to the inspectors, as the mapping information may facilitate side-channel attacks [76] and is strictly controlled. Consequently, we added the delegates as endpoints to the *Keystone*, the identity management facility in Openstack, and configured access rules to enable their access to these interfaces. To enforce advanced security control, the properties of the delegate VMs can be examined regularly by Keystone.

4.4.2.2 Constructing Web-of-Trust

To construct the web-of-trust to cover the Cloud TCB of a target application, the delegate aggregates the *GTMs* of customer VMs' hosting nodes. As most of these nodes have communications with each other due to the application's internal communications, most of their trust evidence is recorded in each other's *GTM*. Therefore, gathering and combining the *GTMs* from a few selected nodes will be sufficient to attest to the Cloud TCB of a target application. A simplified iterative web-of-trust constructing procedure is listed as follows:

1. Customers specify the list of entrance VMs to the proxy. For example, they can specify the smallest subset of VMs, which have altogether interacted with the other VMs of their application. This helps to build a comprehensive web-of-trust with less effort.

2. The delegate attests to the nodes hosting these VMs, and fetches their *GTMs*. These *GTMs* are merged to form the initial web-of-trust.

3. From this web, the delegate determines whether all the VMs' hosts are covered. Additional attestations are performed to the missing nodes, with their *GTMs* merged. Now the web-of-trust contains all the hosts of the target application, and part of the hosts' dynamic dependency. These hosts are added to the critical node set C.

4. This step searches more dynamic dependency nodes according to the attestation relationship from the web-of-trust. Further attestations are performed to critical nodes. For example, for the nodes that have interactions with most of the other nodes, but have not been attested to by the inspector. Or the nodes that have frequently interactions to the node hosting critical customer VMs, but have only had little interactions with other nodes on the web. Among this search, newly identified semantic critical nodes are added to set C.

5. Now the delegate searches the *static dependency* for all the critical nodes. In our prototype, the static dependency relationship is defined manually by the administrators. For example, the *Scheduler* node is marked as the static dependency of the *Compute* node, which is managed by the *Scheduler*. Therefore, every node marked as the static dependency of the critical nodes is added to the critical node set C.

6. With the updated C, the dynamic and static dependency discovery procedure is iterated until the delegate is convinced that most of the relevant trust information for attesting to the target application's *cTCB* has been gathered.

Table 4.2: Response time under default Heartbeat Interval

VM Identity	Measurement Values	SPD Identity
VM_ID_1	$Hash_1$	SPD_ID
	$Hash_2$	SPD_ID
	...	...
VM_ID_2	$Hash_1$	SPD_ID
	...	...
...	...	...

Various attestation strategies can be developed to satisfy various application scenarios. Our current prototype only serves as an exemplar to demonstrate how the inspectors monitor the trust dynamic inside the cloud, and how the delegates form a general understanding for a target application's cloud TCB.

By strategically installing multiple inspectors in different locations inside the cloud, and allowing them to share trust evidence, a more efficient attestation scheme can be achieved. Moreover, with the Decentralized Attention, the integrity of the inspectors is regularly examined by other nodes, including the *CSE*'s attestation services, and the inspectors belonging to other *TERs*. This conforms to the *Mutual-Inspection* and the *Multiple-Instances* restrictions. When multiple inspectors of a *TER* are installed, these attestations can be performed in parallel, according to the geometry of the inspectors and the target nodes. We leave this to our future work.

4.4.2.3 Generating Digest

From the web-of-trust, the Delegate obtains the trust evidence for the necessary nodes. This information helps the *TER* to better understand the trust dynamics inside the cloud, and generate *Digests* accordingly to reflect this understanding. *TER* can therefore implement diverse services and generate Digests at various granularity to satisfy different customer requirements.

We now present an exemplar Digest generation scheme with the focus on attesting to the Cloud TCB of each VM. Table 4.2 presents the exemplar Digest. Each entry contains all the trust evidence for each VM, identified by the *VM_ID*. According to the definition in Section 4.2, the Cloud TCB of a VM is the Cloud TCB of its hosting node, which contains the Core TCB of its *dynamic dependency*, and all the Cloud TCB of its *static dependency*.

However, redundancy exists when listing the evidence in each entry. This is because different nodes may have the same Core TCB. This is due to the homogeneity of the *Infras-*

tructure Services. For example, most *Computes* in the same cluster have the exact same platform configuration. Therefore most of the dynamic dependency nodes of the target host have the same measurement value. Consequently, nodes with the same Core TCB are *merged* as a single entry.

On the other hand, as VMs are usually deployed in the same clusters, redundancy also exists among the entries. For example, VMs' hosts may share the same management node, or have management node exhibiting the same trust evidence. Therefore, the digest can maintain a list of all the different trust evidence, with each entry links to the evidence for the VM's Cloud TCB.

VMs having the same trust evidence list may indicate that they are hosted on the same node. However, as the *GTMs* hide the real identity of each node, this assumption is not assertive. One the other hand, VMs having different trust evidence lists imply that they are hosted on different nodes. Although this leaks partial information for VM deployments to the customers, we argue that it is more important for the customer to know that these VMs are consuming different services.

4.4.3 Separation-of-Powers Models Revisited

We now revisit how the *SoP* models are achieved with the proposed prototype. With RepCloud, the *RepVisor* servers as the *inspector*. Each RepVisor collects the attestation tickets of its interaction peers when their communications are maintained. These attestation tickets serve as the *Digest* for each of the node's neighbours. By gathering all the relevant *Digest* according to a nodes' trust dependency, a RepVisor constructs the *web-of-trust*, so that any misbehaviours will be effectively identified, and efficiently disseminated among concerning nodes. All the aggregated attestation tickets therefore further form the *Digest* representing the properties of the *web-of-trust*, which in turn represents the *Cloud TCB* of a target node. This structure allows any third party to discern the trust dynamics inside the cloud, without injecting heavy monitoring mechanisms. Therefore, the trust evidence can be genuinely gathered by the inspectors.

During cloud attestations, customers first attest the RepVisors' integrity by interrogating their underlying PCRs. They then request the RepVisor to return the collected attestation tickets representing the trustworthiness of each relevant nodes identified from the web-of-trust maintained in their *GTMs*. As each ticket only possesses the PCR values, which are the aggregated hash value, customers then consult the CSE for the explanations for those values. This requires the CSE to return the *measurement list* to the customer, which will help them to disaggregate each entries behind the PCR values. Given each entry, the customers will then request the SPD to translate them into more meaningful properties.

In order to support the *translators*, each RepVisor can implement the with the Property-based vTPM [77], which fits well with the *SPD*'s criteria. By implementing the translators as the *measurement functions* from the [77], the translations will be effectively implemented. Therefore, the *Collaboration Model* is implemented. However, we leave this implementation to our future work, as [77] has not provided an open source implementation of the Property-based vTPM.

As the trust evidence is maintained independently from a particular *TER*, it is accessible to every inspector. Nodes on a web-of-trust vouch for the trustworthiness for each other, so that they are all either "good" or "bad". By attesting to any node on the web, any inspector can then determine the trustworthiness of all those nodes, and gather the trust evidence when they are trustworthy. Moreover, multiple translators can also be implemented in the vTPM framework. Therefore, the *Multiple-Parties* restriction for the inspectors and translators is achieved.

Inspectors are deployed in VMs and hosted on cloud nodes. Translators are also implemented in each cloud node. As the RepCloud framework is employed, each node participates in the Decentralized Attestation. Therefore the inspectors and translators are attested to regularly by all the interacting nodes. These include the inspectors from different *TERs* and the attestation facilities implemented by the *CSE* itself, e.g. the Keystone, or the integrity-based Scheduler [88]. The properties of the inspectors and translators can be defined by different *SPDs*. It is up to the attesters to decide which property certificate to trust. Therefore, the *Mutual-Inspections* is achieved.

The prototype design supports inspectors and translators from multiple *TERs* and *SPDs* respectively to be implanted inside the cloud infrastructure. The *Freedom-of-Choice* is thus achieved as customers can choose among different *CSE-TER-SPD* combinations. This further encourages the *CSE* to cooperate with more *TERs* and *SPDs* to increase their combinations and attract more customers. As a result, as more parties are involved, it is harder to hide dishonest behaviours, so that a more trustworthy cloud ecosystem can be achieved.

4.5 Evaluations

4.5.1 Simulator

We implemented our RepCloud protocols in a simulator and simulated the cloud environment for evaluation. We counted the number of actions performed, e.g. total interactions and total attestation, and compared them with the central attestation scheme (Section 1).

We assume that the same action takes the same amount of time to complete in the production systems implementing either the RepCloud or the centralized attestation scheme. Therefore, these simulations only focus on the differences between these schemes, instead of illustrating the real-world performance of RepCloud, which we leave as the future work.

The RepCloud simulator is built on top of a P2P simulator, the PeerSim [97]. We use the overlay network in PeerSim to simulate the interactions among nodes in the cloud, e.g. NCs and CLCs. We further implemented a new layer of overlay network for the simulation of VM interactions.

Node Model: We differentiate two kinds of nodes: *managers* and *hosts*, representing CLCs (Cluster Controllers) and NCs (Node Controllers) respectively. SC (Storage Controller) can be implemented as part of the CLC. Security components participate in the communication of cloud applications and are hence regarded as hosts. VMs are grouped into cloud applications, and applications are assigned to hosts in a same cluster with round-robin deployment strategy [98]. Every host can run a number of VMs simultaneously. Each VM runs for a *length* of time, and halts afterwards. A cloud application halts when there is no VM remaining.

The *size* of an application is the number of VMs it contains. We specify a baseline for the size, and a *similarity* value to denote a percentage for the size of each application that can be larger than the baseline, i.e. applications have random sizes within the range from $size$ to $size \times (1 + similarity)$. Similarly, we specify the *VM similarity* as the diversity factor for the VMs inside an application. The length of a VM is also specified according to a base value and this factor.

The integrity state of a host is represented as an integer number. The changes in state can result from malicious behaviours, as we discussed in our adversary model, or from trustworthy operations, i.e. applying security patches, and they can only be discovered by attestations. However, in our experiment, we do not differentiate the trustworthiness of a state, as our main purpose is to evaluate how the changes in state are discovered. We also do not consider the internal states of VMs.

Network Model: A manager connects to a number of hosts to form a cluster. It communicates with them for management tasks, e.g. load-balancing instructions. Managers of the clusters are connected to a root manager to form the cloud, which is regarded as the CC (Cloud Controller). A VM only communicates to the VMs that belong to the same application as its, and cloud applications usually have different communication patterns. However, the aggregated patterns of a host tend to have a random distribution, especially when considering the multi-tenancy and dynamic nature of a cloud. For simplicity, we hence specify each VM to randomly communicate to others in the same application. We

define the *frequency* of a VM as the number of communication actions it performs to another VM within a cycle. Total communications in an application performed within a cycle hence equal $size \times frequency \times (size - 1)$. Frequency can also relate to VM similarity, in which case the total number should be calculated separately. Communications among VMs trigger communications among hosts, as they are enforced by the hosts.

Simulation Execution: In the initialization phase, hosts and managers are generated and linked accordingly. Cloud applications are deployed to occupy the full capacity of the cloud, i.e. the total number of VMs that can be run simultaneously. New applications are deployed regularly, to keep the overall load stable. VMs also migrate among nodes inside a same cluster for simulating the effects of load balancing.

Simulation is executed in cycles, in each of which every host executes all its VMs by fetching a list of target communication VMs and generating communication events (actions) to the hosts of the targets. We use the event and time model of PeerSim: time proceeds with the occurrence of events. Every event is generated with a timestamp, and is executed in sequence. The global time is set to the timestamp of the currently processing event. We define the time for performing a communication action as our basic time unit, which is mapped to a millisecond for better presentation. Complicated communications can hence be regarded as a sequence of actions. With the time mapping defined, our experimental results can be effectively mapped for real system analysis. In the following text, we refer to the time as our simulated time by default. In our simulation, different kinds of events use different random number generators, e.g. cloud app deployment, migration or attacks, to keep their indecency and achieve the same VMs interaction pattern.

4.5.2 Cloud Attestation Schemes

Three attestation schemes are simulated and evaluated: a *centralized scheme* (CEN), a *fully Decentralized Attestation Scheme without reputation systems (DECEN)* and our *Rep-Cloud Attestation Scheme (REP)*. We further modified CEN and DECEN to implement fine-grained cloud attestation for the evaluation purpose: i.e. to facilitate each node to determine the properties of its own *Cloud TCB*.

CEN is used in many trusted cloud systems [60, 61]. Attestations are performed from a central delegate to all the cloud nodes regularly. A node does not have the knowledge of its communicating peers' properties. They can only query the central delegate continuously to gather this information. In the worse case scenario, all the nodes will request the delegate to disseminate the other nodes' properties constantly. This equates having the central delegate to broadcast the attestation tickets it gathered to all the cloud nodes periodically.

Table 4.3: Basic Simulation Parameters

Simulation	Length of a simulation cycle (minutes)	1
	Total simulation time (hours)	12
	Bootstrap time (minutes)	10
	Attestation freshness (seconds)	10
Network	# of clusters (CLCs) in the cloud	3
	# of nodes (NCs) per cluster	100
	# of VMs per NC	20
	# of total simultaneous VMs	6000
Cloud Apps	app generation interval (minutes)	1
	# of app generated per interval	60
	# VMs per app	10
	length per VM (minutes)	10
	VM frequency	300
	App similarity	2
	VM similarity	0.6
Attackers	range of attack intervals (seconds)	1 - 10
	# of host to attack per interval	1

DECEN is used in some other cloud attestation schemes [58]. Attestation are performed among nodes. However, trust dissemination and aggregation are not used. Every node maintains its own view of the state of its neighbours. With RepCloud, before a host is communicating to another, it first searches its *LTV* and *GTM* for a valid attestation ticket to the communication target. When a search miss occurs, it performs the attestation, and disseminates the gathered ticket as we discussed in Section 4.3.

The configuration for our simulations is presented in Table 4.3. *Attestation freshness* denotes that attestation tickets can only be valid for 10 seconds. We simulated 600 applications running at the same time in the cloud, each of which contains 10 to 30 fully connected VMs. An application runs for 10 to 16 minutes with a new one deployed after it terminates. The node communication patterns hence change from time to time. Each VM generates 300 to 480 communication actions per second to every VM in the same app.

4.5.3 State-change Detection

We simulate attackers who change the state of a random host in random intervals after the bootstrapping phase. As we do not consider the effects of malicious behaviours in our simulation, trustworthy state-changes are also included. We evaluate RepCloud performance with the state-change detection overheads. Two important criteria are considered: 1) *Total*

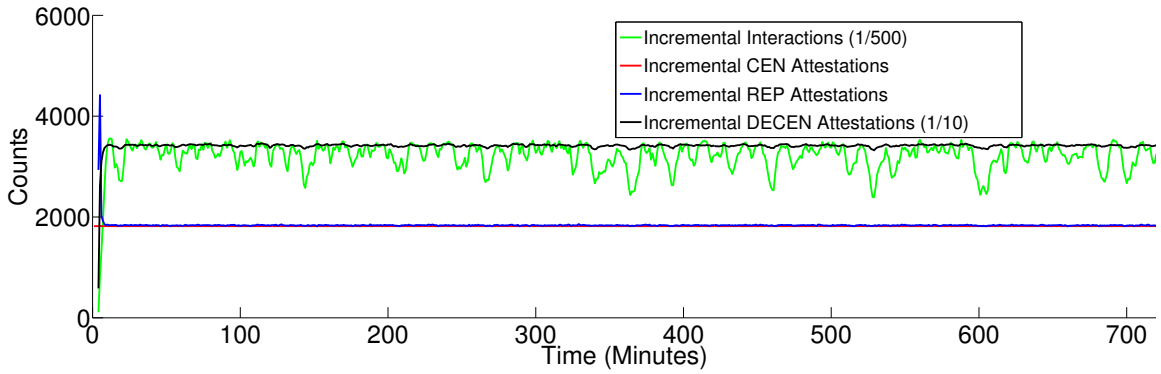


Figure 4.3: Incremental State Change Overheads

Table 4.4: Total Tampering Detection Overheads

	Interactions	Attestations	Tampered
REP	$1.15E9$	$1.30E6$	$1.79E6$
CEN	$1.15E9$	$1.30E6$	$1.78E6$
DECEN	$1.15E9$	$2.4E7$	$1.76E6$

Tampered Interactions: all communications made towards a host with its presumed state (the state deduced from the latest attestation) different from its current state, as they are relying on trust states that have already changed. 2) *Total Attestations:* the total number of attestations performed for discovering the state changes of nodes.

Figure 4.3 depicts incremental statistics of the state change detection overheads of the three attestation schemes. The data is refreshed every 10 minutes. Attestation counts of the centralized scheme (CEN) are a straight line, as managers perform attestations to their hosts in a pre-defined interval (10 seconds). With RepCloud, attestation counts boost to a high level in the first few minutes, as it is in the bootstrapping phase and the *LTVs* and *GTMs* are mostly empty. Later data shows that RepCloud still exposes stable attestation patterns while achieving only a slightly higher level of average attention counts. For the decentralized scheme (DECEN), as it does not share attestation results among nodes, every node will attest to its neighbour when the tickets are regarded as obsolete (after 10 seconds). Hence the attestation counts are much larger than the others (around 20 times from our results). Table 4.4 shows that, RepCloud achieves fine-grained cloud TCB attestation with only a slight increase in attestation overheads. Meanwhile, without reputation systems, redundant attestation effort have increased significantly (as in DECEN).

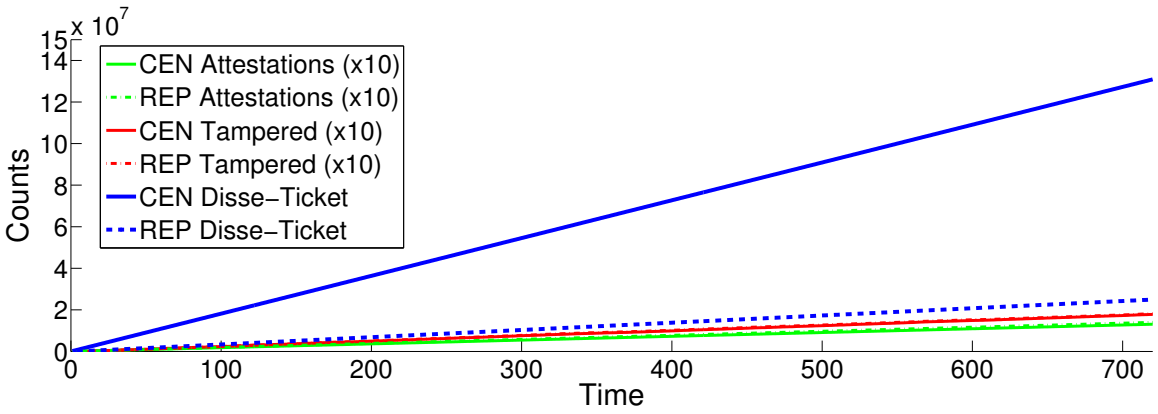


Figure 4.4: Total Ticket Dissemination Overheads

4.5.4 Trust Dissemination

Ticket reporting. With RepCloud, every time a new attestation ticket towards a target node is generated, it is sent to all the nodes that have also attested to the target, according to the attestation relationship in the *GTM*. However, in the centralized scheme, the manager does not possess the communication patterns of each host. Hence, it can only broadcast the new ticket to every node as we discussed before. As we can see from Figure 4.4, the tick dissemination count is around 6 times as that for RepCloud, while the total attestation and interaction counters remain at the same level. Besides, in the centralized scheme, VMs of an application can only be deployed within a same cluster to reduce the broadcast domain. However, in RepCloud, an application can be scattered among clusters, while retaining a low level of dissemination overheads, as tickets are disseminated according to nodes' communication patterns. More importantly, RepCloud reserves the mutual-attestation relationship, from which a higher level of trust is deduced (Chapter 5).

Moreover, as the communication patterns are changing, the communication relationship is also transient. For example, N_a may no longer interact with N_b as all related VMs on it are either halt or migrated out. But $PCR_{a,b}$ may still exist in other nodes' *GTM*, and new tickets towards N_b are still sent to N_a. An obsolete interval can be defined to further reduce the ticket dissemination overheads in RepCloud. Attestation tickets generated before the interval will not reveal an attestation relationship among the nodes.

SGTM fetching. Besides ticket dissemination, RepCloud incurs an extra overheads: the SGTM of a target node is fetched every time a local attestation towards it is performed. However, as newly generated attestation tickets are disseminated to the corresponding nodes, the *GTM* of each node is kept updating continuously. Hence each SGTM fetching may only contain a few updates, especially when the communication patterns of

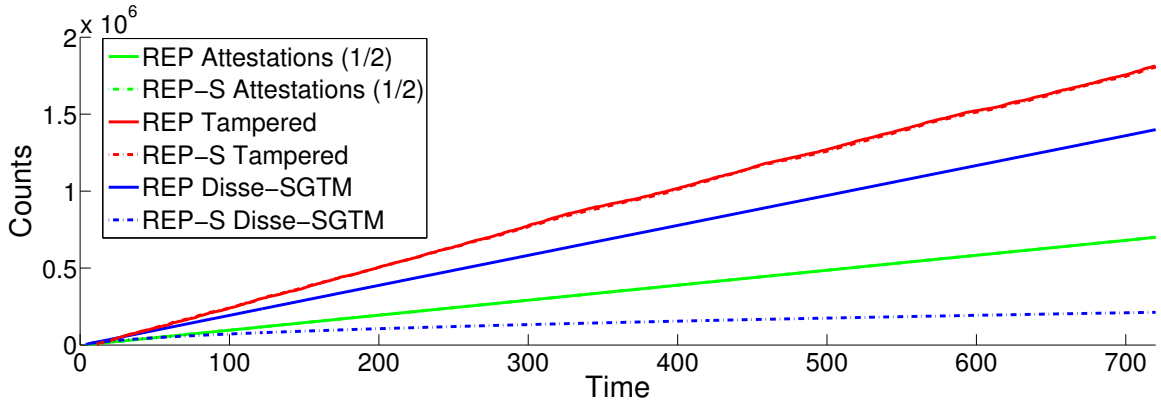


Figure 4.5: Total SGTM Dissemination Overheads

nodes are in a relatively stable phase. To reduce the overheads, SGTM fetching can be delayed for an interval of time (1 minute in our case) when the updated entries of the last time is lower than a threshold. We also specify a cultivated delay interval, i.e. the interval will be increased (by 1 minute) every time when both the current and the last update are lower then the threshold. The interval is reset once the update rate is higher than the threshold.

In our experiment, we specify the threshold as whether the size of *GTM* is changed (i.e. whether new entry is added), as it indicates the changes in communication targets of a node. All the other changes can be updated by ticket dissemination. Figure 4.5 shows that 86% of the total SGTM dissemination counts are saved (REP-S), while preserving the same level of total attestation and interaction counts.

4.5.5 Security Analysis

As reputation systems are used in RepCloud to disseminate TCG trust, existing attacks to them should be considered [44]. However, we argue that as RepCloud transmit tamper-proof TCG attestation tickets instead of raw reputation values, these attacks are avoided by default. For example, in reputation-based P2P trust management systems, nodes can disseminate forged trust information to promote their own reputation value or to degrade that of others. With RepCloud, the attestation tickets are signed by the AIK [31] of the platform, the private part of which can be accessed only by the TPM.

Malicious cooperative [44] is another kind of attack to reputation systems, with which several tampered nodes cooperate to promote the reputation value of each other. With RepCloud, when all the nodes hosting the VMs of an application are tampered with, they may report each other as trustworthy without being identified, and hence tamper with the target application. However, with the multi-tenancy nature of the cloud, each of these hosts may at the same time host VMs from other applications, which may communicate to nodes

in the cloud TCB of those application. The cooperative should hence also incorporate these nodes. We argue that the cooperative should be sufficiently large to contain a web-of-trust closure for tampering a target application, e.g. by tampering the entire cluster. On the other hand, in the cloud TCB attestation procedure, an entrance node to the web-of-trust is attested to by a trusted third party. Hence, the larger the malicious cooperative is, the higher probability it will be discovered.

Attacks to the TCG framework have already been concerned in the Timestamp-based attestation scheme [33], which we used as the local attestation in RepCloud. A nonce in the attestation ticket is used to avoid replay attacks, and session keys are used to counteract man-in-the-middle attacks. As we discuss in our adversary model, we do not consider runtime attacks to TCG mechanisms. We also do not consider hardware-based attacks.

4.5.6 Discussions

Greedy deployment policies. A Greedy VM deployment policy [98] can be employed by cloud systems, which exhausts a node before moving on to another node when deploying applications. Hence, VMs from the same application may reside on the same host. As self-attestation (a node proves its own state to its VMs) is not considered, attestations among these VMs are not performed. However, we argue that it is highly improbable for a node in the cloud to have no communications with others. And any single interaction to others may initiate local attestations as with the adaptive nature of RepCloud. Besides, as we will discuss in Chapter 6, when all the VMs reside on a same host, the host becomes the entrance node to this application, and will be attested to when cloud TCB attestation is initiated. Moreover, the centralized attestation scheme can be incorporated with RepCloud. Managers can participate in the trust dissemination process, and will initiate attestation to a node when necessary.

Reputation vs. TCG trust. The meaning of trust in RepCloud is different from trust in Peer-to-Peer systems or Social Network Service systems, hence the purposes for incorporating the reputation systems are also different. Trust in those systems only represents a "personal" view of a node, while trust generated by the TCG framework is both deterministic and tamper-proof, i.e. one attestation ticket from a single node can represent the genuine state of the target node, and this statement cannot be forged or modified. Moreover, the evaluation of trust in reputation systems can only be performed after the actual action has been taken place, but may have a relatively long endurance. Meanwhile, TCG trust can be obtained before the action, but every node has the same probability of being corrupted in the next period of time. Hence the main purpose of RepCloud is to reduce unnecessary attestation overheads while preserving a low interval for repeating attestation. Still, as they

share similar trust dissemination and aggregation patterns, algorithms in reputation systems can still be adapted in RepCloud.

TCB identification based on Direct Communications. We identify the Cloud TCB of a node through its direct communications with other nodes. This is based on our observations that both service dependency and malicious behaviours require direct communications to the target. However, direct communications do not directly indicate a dependency relationship. This is because of the multi-tenancy characteristic of the cloud infrastructure, according to Figure 1.2. For example, direct communications among nodes (N_i and N_j) may indicate that the N_j is providing services for a particular VM hosted on N_i, but not for N_i's VM. In this case, we assume that attesting to a target node will evaluate not only all the loaded applications but also their configuration files. From the configuration, the attesters will further determine whether the loaded services is enforced to provide services to their cloud applications.

4.6 Summary

Decentralized Attestation (DA) organizes trust relationships according to a cloud's internal dynamics. DA helps the *Infrastructure Service Providers* to effectively identify property changes inside the cloud, while allowing each node to discover the trustworthy status of its neighbours and gather trust evidence accordingly. It achieves a similar *state-change detection rate* with the centralized attestation schemes, and incurs manageable overheads. As the trust evidence of the interacting peers is maintained locally in each node, its upper layer services are able to better react to the cloud dynamics. Interfaces can be exported to these services for enforcing communications according to the trustworthy state of the target nodes or services. This greatly reduces the *adversary windows*.

In addition, these neighbours constitute the *Cloud TCB of the node*. By assembling the *Cloud TCBs* of the nodes hosting all the VMs of a cloud application and supplemental services (e.g. the *Supporting Services* or the *Business Logic Service*), the *Cloud TCB of the application* is identified. Collecting the trust evidence from all these nodes, this TCB's properties can be determined.

In the next chapter, we discuss how to employ DA to deduce a *Cloud Root-of-Trust (cRoT)* for a cloud application according to the trust dynamics inside the cloud. *cRoT* implements an effective way of supporting Cloud TCB attestation for a cloud application. And in Chapter 6 interfaces are designed to allow upper-layer services to build advanced trust management inside the cloud.

Chapter 5

Cloud Root-of-Trust

5.1 Overview

In the previous chapter, we implemented the RepCloud system, which introduces a central attestation delegate to attest to the fine-grained *Cloud TCB* for customers' application. However, with RepCloud, customers still need to attest to the delegate to verify its trustworthiness. But these attestations only prove its existence rather than effectiveness. In other words, without additional evaluations to the detail configurations of this delegate, customers cannot be assured that their attestation requirements have been genuinely fulfilled. But this evaluation is especially complex, given the gigantic-scale and multi-tenancy of a cloud service provider.

Moreover, in RepCloud, the trust information of the entire software stack of the *Core Services* on each node is disseminated and aggregated. For example, the Decentralised Attestation protocol in RepCloud will disseminate the attestation ticket representing the security properties of a *Compute* node's hypervisor, the *Compute* management service, along with all the software components participated in bootstrapping and hosting them, such as the bootloader and operating system kernels, etc. This incurs large communication and management overheads. It further restricts the trust dissemination scale to one level, as a higher level will result in many more attestation tickets being disseminated at a much wider scale. Keeping these tickets up-to-date among all the nodes requires tremendous efforts. Though it is possible to install a central server to store all the tickets and provide querying services, this structure contradicts the design goal of the Decentralized Attention. It restricts the scalability and flexibility of the trust management framework, and introduces performance bottleneck and a single point-of-failure.

Consequently, the one-level trust dissemination forces the delegates to attest to multiple nodes for building the web-of-trust for a target application. This is because a cloud application usually contains multiple VMs, along with multiple supplemental service VMs. For

most of the time, communications only occur among a subset of these VMs, especially for a large application. In this case, the delegates have to attest to at least one node of each subset. This contradicts the cloud management perspective, which is to hide the underlying hardware details and present uniform access abstractions. The exposed information may also facilitate malicious attacks to the underlying cloud infrastructure or other customers' VMs hosted on the same nodes [76]. Moreover, cloud dynamics implies the bindings between hardware and virtual resource are changing constantly. Thus, the TPM set for attesting the *Cloud TCB of an Application* needs to be regularly updated, which also incurs severe management costs and further exposures to the cloud dynamics.

We observe that, from customers' perspectives, a desirable attestation scheme is to implement their own *attestation delegate* inside the cloud. These delegates have deeper understandings of the customers' cloud requirements, so that they can perform more accurate attestations. Customers then attest to their own delegates to validate their integrity. Cloud providers can also provide the delegates with necessary information of the supporting services, so that they can attest to the services regularly. This will help the delegates to determine whether the acquired services are genuinely enforced and correctly configured. To avoid overexposure of the cloud infrastructure's implementation details, property-based attestation can be employed.

However, the major obstacle of this scheme is that the attestations to the cloud nodes require customers' delegates to directly interrogate the TPMs of the related cloud nodes. This incurs significant management overheads, as in order to validate a TPM's signature, the delegates have to obtain its AIK (Attestation Identity Key) certificate. The management is further complicated when considering the dynamism and complexity of the cloud model [80, 88]. On the other hand, revealing TPMs' identities exposes the VMs' physical locations, as each TPM is uniquely associated with a physical machine. This contradicts the cloud's management premise, which is to hide the physical identities. It may also facilitate malicious behaviours, such as collocation attacks [76].

We argue that the difficulties in achieving effective and practical cloud attestations are generally caused by the problem of *insufficient abstraction to the Root-of-Trust model in one cloud*. As the cloud hides its underlying hardware infrastructure and exposes only a uniform view of virtual resources to a cloud application, the *RoTs* for the application should also have a uniform abstraction. An integrated logical *Cloud Root-of-Trust (cRoT)*[1] for an application is thus desirable, which manages the hardware *RoTs* (TPMs) of all the cloud nodes hosting the customer's cloud application and the supporting cloud services.

[1] In this work, we only consider the Root-of-Trust for Measurement, which supports the remote attestations.

This *cRoT* should scale with the application, and help the customers to collect the trust information maintained by each relevant TPM, without the need to validate the TPM's identity.

We define the *Cloud Root-of-Trust (cRoT)* for a cloud application as *the set of TPMs participated in attesting the properties of the Cloud TCB of the application*. In other words, this *cRoT* encompasses all the TPMs attached to the nodes hosting all the VMs of the application, and all the services (Figure 1.3) supporting the application (i.e. its *cTCB*), i.e. all the VMs' hosts' *Dynamic* and *Static* dependencies inside the cloud. In order to effectively attest to the applications' cTCB, these TPMs should collaborate together to share trust information, and restrict each other by preventing any of their hosts to silently change status. In this case, by interrogating any TPM inside this set, the properties of all the involving services can be examined. By verifying the *AIK* of any TPM, the genuineness of all the others will be deduced.

In this chapter, we aim to design the *Software Defined Cloud Root-of-Trust (cRoT)* abstraction, with which a customer attestation delegate will be able to attest the properties of cloud services, without interrogating their underlying TPMs. In particular, we present the NeuronVisor framework. A NeuronVisor (or *Neuron*[2] for short) is a Root-of-Trust (i.e. TPM) management software, deployed on each cloud node. A Neuron implements the basic functionalities to genuinely report the properties of its upper layer service components. Moreover, it attests the integrity of the Neurons on the cloud nodes hosting the interacting services. The attestation results are disseminated among the logical surrounding Neurons by enforcing the *Decentralized Attestation (DA)* scheme [80]. DA thus binds the interacting Neurons to a *Neuron Web*, within which any Neuron's property violations will be effectively identified and reported. Therefore, the attester can deduce the integrity of a target node's Neuron by determining how the Neuron is confined in a Neuron Web. This integrity in turn indicates the target node's capability of genuinely reporting the upper layer services' behaviours. Accordingly, the *Neuron Web* serves as the abstract *cRoT* for a cloud application.

As only simple properties of Neurons are managed, the disseminated trust evaluation of each node can be modelled simply as digital number, instead of the PCR values of each node's entire software stack. This greatly reduces the trust management overheads. Moreover, this lightweight trust evaluation model allows the evaluation values to be disseminated among the entire connected nodes, instead of only directly communicating nodes from the

[2]We make an analogy to the biological *Neuron* cells. As we will introduce in the following sections, NeuronVisor is also an autonomous entity, which collaborate with its neighbours, by forming connections and exchanging simple information, to exhibit unified behaviours.

RepCloud framework. Therefore, this scheme allows the *inspectors* to only attest to one entrance Neuron and query the trust evidence of the entire Neuron Web.

The conceptual Neuron model for achieving this *cRoT* is illustrated in Section 5.2. Section 5.3 introduces the Neuron Web model. Section 5.4 presents the NeuronVisor prototype, and in Section 5.5, simulation results are presented.

5.2 NeuronVisor Framework

A *Neuron* is a TPM management software layer. It measures and reports the properties of the upper layer services running on its host. It also possesses the capability of attesting to the integrity of other Neurons. In NeuronVisor, Neurons adapt the Decentralized Attestation scheme to attest to each other and share the attestation information (e.g. attestation results). This helps them to build trust relationship, which is represented by Neuron *connections*. The connected Neurons then share the trust information of their upper layer services (i.e. the vTPM measurement metric), and form Neuron webs, which in turn act as the *cRoT* for cloud applications and implement cloud attestation services. In this section, we discuss the establishment and maintenance of the Neuron connections. We also illustrate how the Neuron webs fulfil the *cRoT* semantics.

Two forms of attestations are implemented by each Neuron. Firstly, it implements the attestations to the security properties of the upper layer services, e.g. Virtual Machines or cloud management service components. This is implemented by the vTPM Management Module. Secondly, it adapts the *Decentralized Attestation* [80] (DA) to attest to only the integrity of the peer NeuronVisor layer on all the interacting nodes, based on their hosted services' communication patterns. It is achieved by the cooperation of the Attestation Module and the Network Monitor Module. By aggregating and disseminating this integrity information with the Trust Propagation Module, Neurons on the frequently communicating nodes form the *Neuron Web*, where the integrity of each Neuron is regularly examined. By sharing the security properties of the upper layer services among the attested Neurons, this web forms a dynamic and scalable software layer to attest to all the hosted interacting services with a uniformed interface. By attesting to and querying any Neuron in this web, the security properties of all dependent services are gathered. This dynamically formed Neuron Web thus achieves the cloud Root-of-Trust abstraction for attesting to a cloud application.

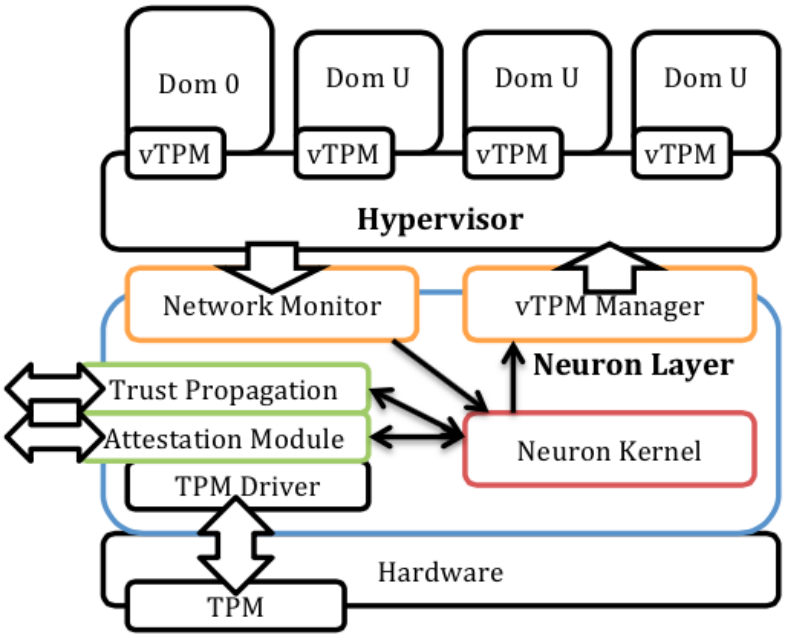


Figure 5.1: Neuron Structure.

Neuron Structure. Figure 5.1 depicts the structure of the Neuron layer. It can either be implemented as a component inside the hypervisor (including the hypervisor's management layer, such as in the Dom0 of the Xen hypervisor [30]), or as a lower layer hypervisor with nested virtualization [15]. A Neuron interacts with upper layer services through its Network Monitor and vTPM Manager. The Network Monitor intercepts network traffics, and determines the trustworthiness of the communication target from the data maintained inside the Kernel. The vTPM Manager exports the necessary Kernel data to vTPMs to facilitate cloud attestations. Besides, a Neuron is also able to actively query vPCR measurement values from the vTPMs it maintained. The Kernel is maintained by the Attestation Module and the Trust Propagation Module. Attestation Module attests target Neurons when necessary. The returned attestation information is updated to the Kernel. The Trust Propagation Module exchanges the updated Kernel data with other Neurons by adapting the Decentralized Attestation algorithms.

This chapter mainly focuses on this second form of attestation: how the decentralized attestations are implemented in the NeuronVisor framework to achieve the cloud Root-of-Trust abstractions for a cloud application.

Neuron Connections. Figure 5.2 depicts a simplified Neuron connection topology. With NeuronVisor, a node can communicate with the others only when its Neuron is *connected* to theirs. An established *connection* means the Neuron has successfully attested

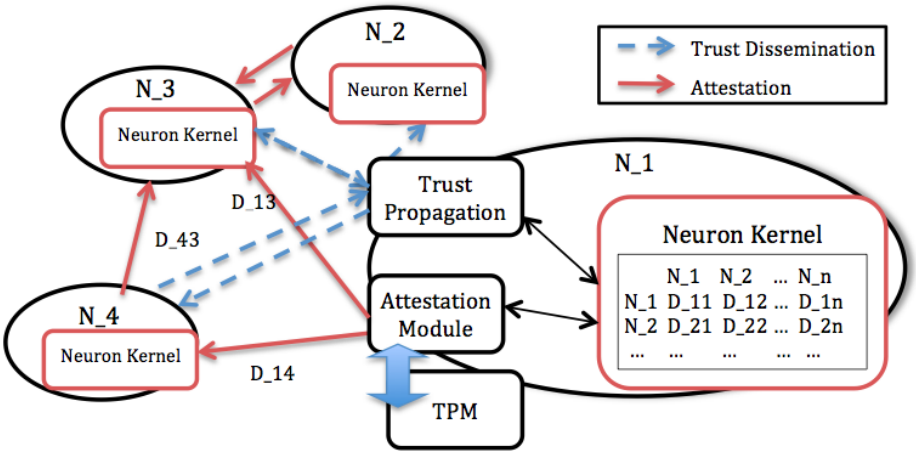


Figure 5.2: Conceptual NeuronVisor Model.

to the integrity of the target Neuron. The connected Neuron is referred to as the attesting Neuron's *neighbour*. These Neuron connections link semantically depending Neurons and form the Neuron Web.

An important property of a connection is its *strength*. It represents the *attestation relationship* from the Neuron to its neighbour. It is an integer indicating, in the view of the attesting Neuron, how often the neighbour is attested to. The higher the strength, the harder the connected Neuron can change its property without being detected. Moreover, as suggested in [80], in an environment where nodes frequently attest to each other, an attestation ticket can be effectively reused for better reflecting the trust dependency and reducing redundant attestations. A Neuron thus also determines the strength value for its connection to a neighbour by analysing the attestation relationship from other Neurons with the target neighbour. This relationship is gathered by the Trust Propagation module, and is maintained in the *Neuron Kernel*.

Trust Propagation. As a Neuron only concerns the trustworthiness of its neighbours, the Kernel of each Neuron maintains a *partial attestation graph* to record only the information of attestations performed to and from the neighbours. This is achieved by adapting the Decentralized Attestation (DA).

When a node initiates a new communication to another, its Neuron first attests to the other's Neuron. This attestation establishes a new Neuron connection by adding an entry inside the Kernel recording the attestation information. The Neuron then *fetches* all the attestation information from the neighbour's Kernel, and *aggregates* it into its own Kernel. Afterwards, the Neuron *propagates* back the updated data to all corresponding neighbours, who also depend on this data. These Neurons perform the aggregate-propagate procedure, until their Kernels are stable, i.e. a new aggregation only has negligible changes to the

Kernels. This iterative information exchange thus allows the partial graph to contain all the attestation relationship regarding the neighbours [80, 45]. It approaches a more accurate strength value assignment and achieves fast and accurate trust information dissemination.

For example, in Figure 5.2, after Neuron N_1 attested to N_3, it fetches N_3's Kernel. It updates related entries into its own Kernel, and propagate these updates to N_2 and N_4, since they have also attested to N_4, which implies trust dependency. These two Neurons aggregate the updated information with their own Kernel, and propagate the changed entries back, which may also contain information that N_1 depends on, e.g. new attestations they performed towards N_3.

Connection Strength. The *attestation relationship* from a source Neuron to a neighbour is modelled from the past attestation patterns as an integer. This value represents a *Direct Trust* (D) to the neighbour deduced by the source. For example, in Figure 5.2, the solid arrows represent the direct trust relationship from its source to the target. *Transitive Trust* is deduced by connecting the direct trust from the source to a third Neuron and from the third one to the target. The strength value for the connection to the neighbour is in turn modelled from both the direct and the transitive trust through all possible third Neurons. For example, in Figure 5.2, the strength value for the connection from N_1 to N_3 is calculated from the direct trust D_13 and the transitive trust, which is calculated from D_14 and D_43. It represents how the neighbour is attested by both *directly* and *indirectly* by the source Neuron. This strength calculation helps NeuronVisor to reduce redundant attestations, while preventing malicious collaborative attacks [44] (Section 5.5).

The strength value decreases constantly, since it is modelled ultimately from the attestations, whose credibility degrades over time. When it reaches a *threshold*, the Neuron determines whether its upper layer services still have communications to their counterparts hosted on its neighbours. If not, it tears down the connection by removing the related entries. Otherwise, it attests to the neighbour and updates the strength. The new attestation information is also propagated to other neighbours, which will result in a new series of information exchanging. This strength degradation and repeatedly attestations thus allow the connection strength to better reflect the trust dependency and communication dynamics among interacting nodes. When a neighbour failed the attestation, it is identified as "unhealthy" and the connection is torn down. This failed node is reported, and will be examined or re-initiated. This information is also propagated among other neighbours, who will also re-examine the unhealthy target, and tear down the connection when necessary.

Neuron Web. Connected Neurons form a *Neuron web*. By choosing a *Root Neuron* and specifying strength criteria, a *partial* Neuron web is identified, where all the connections have the strength satisfying the criteria. This indicates that all the Neurons on the web are

attested by the Root Neuron either directly or indirectly, with the frequency implied by the strength criteria.

In NeuronVisor, the connection strength reflects the attestation relationship, which in turn indicates the communication patterns of the upper layer services. Therefore, by setting the criteria according to the services' communication patterns, the generated partial web will cover all the Neurons hosting these services. As only attested Neurons are bound on a web, attestation information for the upper layer service components (e.g. vPCR values [94] representing the security properties of a service VM) maintained by each Neuron can be shared freely among the Neurons on the same web without interrogating the underlying TPMs. Therefore, by attesting the Root Neuron, the integrity of all the relevant Neurons on the web are verified, and the trust information they maintained will be gathered. This web thus acts as the Cloud Root-of-Trust (*cRoT*) for all the concerned services, which includes the connected application components and all participating cloud service components.

5.3 Neuron Web Model

In this section, we first present the basic notations. We then illustrate the procedures for maintaining the Neuron Kernel. From the Kernel, *Transitive Trust* is deduced to determine the connection *strength*, which facilitates the cloud attestations introduced in the next chapter.

5.3.1 Direct Trust

Neuron Kernel collects the results of the attestations performed by the Neuron and its neighbours. These attestations represent the *Direct Trust* relationship among Neurons. Direct trust is modelled from the past attestation tickets. Two parts of information from tickets are usually used for modelling trust [80]: the PCR values, which record the properties of the Neuron and a timestamp, t, which records the time the PCR values were collected.

However, since only the integrity of a Neuron is concerned, the PCR values are replaced by a binary value: "healthy" or "unhealthy". It is implemented by examining whether these PCR values are located in a known-good values list [31] for the NeuronVisor implementations. When an "unhealthy" Neuron is discovered, it will be isolated and reinitialized immediately. To further isolate maliciously collaborating unhealthy Neurons, their kernels will be examined, with their neighbours being attested.

As "unhealthy" nodes are eliminated from the Neuron Web, only the timestamp is necessary to represent this evaluation. A Neuron thus keeps an *Attestation History (AH)* for each healthy neighbour, recording all timestamps for the *recent* attestations (defined next) it performed to the neighbour.

In practice, repeatedly attestations have a minimum interval, which is determined by the time needed to fulfil a complete attestation [33]. We denote this interval as a *step* (τ). Thus the distance between two timestamps t_2 and t_1, can be expressed as the number of steps in between. It is denoted as $\Delta(t_2, t_1)$:

$$\Delta(t_2, t_1) = \lfloor \frac{(t_2 - t_1)}{\tau} \rfloor \tag{5.1}$$

With this definition, a *recent* ticket is defined as the one generated κ steps away from the current time, where κ is an integer constant chosen for a cloud implementation according to its needs to balance performance and security requirements.

We now model the direct trust from Neuron N_a to N_b at time t: $D_{a,b}(t)$. $D_{a,b}(t)$ is formed by combining the timestamps maintained in the attestation history of the neighbour (N_b). It is an integer interpreted as a *bitmap vector* with the length of κ. Every bit represents a timestamp one step away from its higher adjacent bit, and the most significant bit indicates the time t. A bit is set to 1 when an attestation is performed at the step it stands for. Thus the direct trust, calculated as below, reflects all the recent successful attestations up to time t. $AH_j(t)$ denotes the attestation history for Neuron N_j at time t. As a step is defined as minimum attestation interval, different timestamps t in *AH* cannot indicate a same bit index. We can thus safely use *summation* instead of *bitwise OR* ("|") for setting the corresponding bits.

$$D_{i,j}(t) = \sum_{n} 2^{\kappa - \Delta(t, t_n)}, \forall n(t_n \in AH_j(t)) \tag{5.2}$$

This definition allows two evaluation values be compared. The larger one indicates the more recent attestation is performed and hence higher trust credibility. On the other hand, the bit pattern represents the attestation pattern. These two properties are used for modelling *Transitive Trust* and *Combined Trust* respectively in the following sections.

Whenever an attestation is performed, the new evaluation value is calculated by shifting the original one rightwards $\Delta(t_{new}, t_{original})$ bits, and setting the highest bit to "1":

$$D_{i,j}(t_{new}) = 2^{\kappa} | \frac{D_{i,j}(t_{original})}{2^{\Delta(t_{new}, t_{original})}} \tag{5.3}$$

As each Neuron maintains a partial attestation graph, the attestation information regarding a pair of neighbours may be incomplete. Thus Neurons exchange their gathered information to approach a more accurate attestation relationship among Neurons. When two different versions for one same attestation relationship are maintained by two Neurons at different times, e.g. $D^a_{i.j}(t_1)$ $D^b_{i.j}(t_2)$, they are first adjusted to a common time, and then merged together with the bitwise OR operation. In the rest of the paper, we use the superscript to denote that the data structure is maintained by a certain Neuron. Each data structure also contains a parameter t to indicate that its value is calculated for time t. We omit these two notations when the context is clear.

$$D_{i,j}(t_{new}) = \frac{D^a_{i,j}(t_1)}{2^{\Delta t_{new},t_1}} | \frac{D^b_{i,j}(t_2)}{2^{\Delta t_{new},t_2}} \tag{5.4}$$

5.3.2 Neuron Kernel

The Kernel of a Neuron N_i is a matrix $K^i(t)$ recording the direct trust it gathers at time t.

$$K^i(t) = \{D_{a,b}(t)\} \tag{5.5}$$

We define $D_{a,b} = 0$, when no attestation is performed from N_a to N_b. We also define $D_{a,a} = 2^{\kappa+1} - 1$, the maximum direct trust value, as a Neuron always trust itself. The *Neighbours* Nbr_i of Neurons N_i are hence defined as:

$$Nbr_i = \{N_k \mid 0 < D_{i,k} < 2^{\kappa+1} - 1\} \tag{5.6}$$

Neuron Kernel is maintained as a *Global Trusted Matrix* in the Decentralized Attestation scheme in Chapter 4. Specifically, three steps are used to maintain the Kernel: 1) the Neuron first *gathers* the direct trust for its neighbours by performing remote attestations to them; 2) it then *aggregates* entries in the Kernel of the neighbours; and finally 3) it *disseminates* the updated Kernel back to corresponding neighbours.

However, in NeuronVisor, only the *Direct Trust* values are disseminated, instead of the entire PCR measurement values used in RepCloud [80]. This greatly increases the trust dissemination efficiency. For trust gathering, whenever a Neuron (N_i) attests to another (N_j), it updates the entry $D_{i,j}(t)$ in its kernel using Equation 5.3. Other entries are also refreshed to adapt to the new time t. Afterwards, the trust aggregation is performed by fetching the kernel of N_j: K^j. Entries in the retrieved K^j are merged with the corresponding ones in

K^i by using Equation 5.4. Finally, the updated entries are sent to the set of Neurons who also have dependency on this information. This set is determined by the I_b^a algorithm in [80].

5.3.3 Neuron Connections

The Connection Strength is calculated by aggregating the *Transitive Trust*. Transitive trust has been discussed in P2P systems [45, 49]. Trust towards a "stranger" can be determined by consulting a "friend", whose trustworthiness is known and who knows the trustworthiness of the stranger. Similarly, the integrity of a Neuron can be assumed when it is attested by a neighbour. Transitive trust reuses trust information and reduces redundant attestations. Currently, we employ it to model the trust evaluation in NeuronVisor for its simplicity. We agree that different reputation models from the P2P systems may be utilized to satisfy different usage scenarios, and we will investigate further in our future work.

We first define a *transitive attestation path*, which contains a sequence of Neurons, with the former one has attested the following one. As an attestation reflects the trustworthiness *up to* the time it is performed, the transitive trust should only represent the trustworthiness *up to* the time when all the Neuron on the attestation path are regarded as equally trustworthy. This means the value of this evaluation equals the smallest direct trust value along the path. Therefore, in NeuronVisor, currently only one-hop transitive trust is considered [99], as longer paths will only have very small evaluation values, which will be discarded when calculating the *Connection Strength* in Equation 5.8. Consequently, each path contains three Neurons. The transitive trust, $T_{i,k,j}$, thus denotes the trust implication towards Neuron N_j, regarding a path containing $\{N_i, N_k, N_j\}$.

$$T_{i,k,j} = \min(D_{i,k}, D_{k,j}) \tag{5.7}$$

As $D_{i,i}$ is defined as the maximum value, transitive trust calculation also incorporates the direct trust: $T_{i,i,j} = D_{i,j}$.

We now define the *Connection Strength* $S_{i,j}$ from Neuron N_i to N_j. It is the maximum transitive trust value for all possible transitive paths from N_i to N_j. It represents the most recent time N_j was iteratively attested to by N_i.

$$C_{i,j} = \max(\{T_{i,k,j} \mid D_{i,k} \neq 0 \wedge D_{k,j} \neq 0\}) \tag{5.8}$$

As the trustworthiness of the middle Neuron is also considered in the *Transitive Trust* calculation, malicious collaboration attacks are restricted. When the related entries in the

Kernel change, this trust value is updated. Every time when a node interacts with another one, this value is first adjusted to reflect the current time (by shifting rightwards Δ bits). It is then compared to the *threshold* value Φ. Only when it is larger, will the communication been enforced. Otherwise, the Neuron triggers a new attestation to the target. This attestation updates the transitive trust value, and is also disseminated to other Neurons, thus updating the corresponding values maintained by the other Neurons.

5.3.4 Discussions

Connections Strength Interpretation. Neuron Connections help forming a *cRoT* (the Neuron Web). As discussed, a partial Neuron Web is in fact a dynamically formed centralized attestation domain, with the Root Neuron of the web as the attestation delegate. The strength values of the connections indicate how often these Neurons are attested to, directly or indirectly, by the centre. Equation 5.9 indicates how a strength value is interpreted as attestation intervals. Consequently, after choosing a VM as the cloud attestation target, customers have actually chosen an attestation domain, with the VM's underlying NeuronVisor as the centre. After examining the integrity of this centre Neuron, customers are able to infer the integrity of the other Neurons from the returned strength value matrix. As Neuron Connections are formed according to one upper layer's communication relationship, this attestation domain preserves the application's dependency, which helps determining the *cTCB*.

$$\Phi = \frac{1}{2^{Interval}} \tag{5.9}$$

Centralized Communications. Attestations among Neurons are based on decentralized communication patterns. However, in a cloud implementation, centralized communications still exist. For example, in OpenStack, the central Scheduler node talks to every Compute node regularly. However, the decentralized attestations are still enforced, because of the trust dissemination and transitive trust aggregation. When the Scheduler attests to a Compute node, it fetches the Neuron Kernels of the Compute node, which contains attestation information for other Computes. Therefore, the Scheduler only attests to the Compute nodes that have not been attested to recently. Decentralized attestations thus distribute the attestation responsibilities from a central delegate to all the cloud nodes. This prevents the single-point-of-failure. Meanwhile it reduces the complexity for managing the centralized attestation delegate.

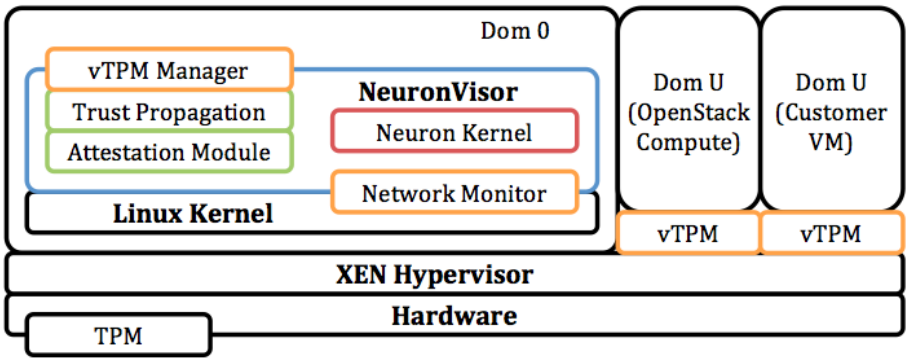


Figure 5.3: NeuronVisor Implementation with XenServer and OpenStack.

5.4 Implementation

5.4.1 RepVisor-based Prototype

We built a proof-of-concept NeuronVisor based on the RepVisor prototype in Section 4.4. In NeuronVisor, Decentralized Attestations are only enforced on the basic trust management facilities, e.g. the trusted computing software stack, vTPM manager, and the NeuronVisor itself. Therefore, all the OpenStack related services and libraries are moved into DomUs. This is well supported by the XenServer-OpenStack architecture [30]. Figure 5.3 depicts the implementation structure for an OpenStack Compute node [90]. The management nodes have a similar architecture.

NeuronVisor is currently implemented as a user-space application in Dom0. Dom0 is the managing domain for the Xen hypervisor [30]. It controls the VMs (referred to as DomUs in Xen) entire lifecycle and implements the key hardware device virtualization functionalities for Xen. Therefore it is regarded as part of the Xen hypervisor. This makes our current NeuronVisor implementation an extension to the Xen hypervisor. As oppose to implementing NeuronVisor with the nested-virtualization scheme, which runs under the Xen hypervisor, we chose this architecture mainly because it is simple to implement. In a production environment, this scheme also imposes the least changes to the existing systems. However, the nest-virtualization scheme will have a much smaller attack surface, as it will be running in the lower layer. We will discuss the design of the nest-virtualization architecture in the next section, and leave its implementation to our future work.

In the current implementation, the entire Dom0 and the underlying chain-of-trust, serve as the NeuronVisor's TCB. The decentralized attestations for attesting to each Neuron thus verify the integrity of this entire chain-of-trust [31, 74], including the bootloader, Xen hypervisor, and the entire software stack loaded inside Dom0. XenServer's measured boot

supplemental pack [30] is enabled to support the TCG-compliant chain-of-trust. The IMA-enabled Linux kernel is also deployed in Dom0 [74] to measure all the loaded software components inside. User space Trusted Computing services are also deployed in Dom0, including TrouSers [92] for implementing the Trusted Computing services libraries, and the OpenPTS [93] for implementing remote attestations.

On the other hand, the Active Attestation scheme is reused. The Network Monitor also remains the same. For trust disseminations, only the numeric Neuron Matrix is disseminated, instead of attestation tickets.

Each Neuron is deployed with an expected measurement list (i.e. the *white-list*), which records all the software components that are allowed to be loaded inside every Neuron's Dom0. The decentralized attestations thus examine whether each target Neuron's measurement list is included in this white-list, and deduce a binary attestation result: whether the Neuron is *healthy* or *unhealthy*. By combining the timestamps for each attestation, the *Directly Trust* is calculated. In our experiments, every Neuron is loaded with all the Attestation Identity Keys (AIK) for all other Neurons. We envisage a centralized Privacy CA [100] server to provide the AIK management service in a real production mode.

With our experiments, initial attestations to NeuronVisor and its entire TCB incur a relatively long delay (around 15 seconds). Major delay is caused by modelling and verifying the measurement list [31] with the IMA [74] and OpenPTS [93]. However, as Dom0 in this XenServer-OpenStack installation only contains a few core virtualization functionalities, its configurations seldom change. Therefore, by integrating the *active attestation* scheme in [33, 95], the periodically attestations to the NeuronVisor after the initial attestation are implemented by examining whether the PCR values remain the same as the previous one. The biggest attestation delay is thus reduced to less than 0.6 seconds [95], while with the most common periodically attestation intervals remain around 0.0016 seconds. As this NeuronVisor implementation shares the similar structure with the RepVisor, it has the same network overheads as illustrated in Table 4.1.

5.4.2 Nested Virtualization-based Design

We now discuss a implementation proposal for the nested virtualization-based design. We leave the implementation to our future work.

To minimize the TCB, NeuronVisor can be implemented in a layer below the Xen hypervisor. A nested virtualization structure can be employed [101]. This implementation can further integrate TrustVisor [91] and delegate the I/O intensive operations to Dom0 as PALs [91]. In this case, the chain-of-trust for the NeuronVisor will exclude the Xen hypervisor and the entire software stack in Dom0. Here we propose the architecture design

for this prototype, as actual implementation will require extensive work on the hypervisor, which is out-of-scope for this thesis.

NeuronVisor can be built by integrating the vTPM management components from TrustVisor [91] and the para-passthrough device driver model from BitVisor [102]. It is implemented as a nested virtualization layer [15]: it is a thin hypervisor to host only one VM, the actual hypervisor realizing the Hardware Abstract Layer (HAL) virtualization [30, 21].

In this implementation, NeuronVisor uses the TrustVisor to implements the μTPM abstractions. This can easily be extended to support full vTPM functionalities [94]. TrustVisor manages the μTPM and implements chain-of-trust extensions from the hardware TPM to it. TrustVisor also supports its attestation to the existence of isolated execution to an external entity: this facilitates the implementation of Neuron Attestations. On the other hand, BitVisor implements a parapass-through hypervisor architecture, allowing most of the I/O access from the guest operating system to pass-through the hypervisor. Meanwhile, BitVisor maintains the minimum access to the I/O flow necessary to implement additional security functionalities. With BitVisor, the network traffic can be monitored by the Neuron, while leaving the network device driver and the networking virtualization implemented by the upper layer hypervisor. This is realized by defining a set of access control policies with the BitVisor I/O mediation interfaces to initiate the evaluation of the target Neuron's healthiness before enforcing the network traffics.

The Attestation module requires additional support from the network devices to initiate active communication to other Neurons. This is implemented outside of the NeuronVisor, as the network device driver is controlled by the hypervisor. As shown in the figure, the attestation module runs as a VM on the hypervisor, but its critical operations are implemented as a PAL [91], which initiates dynamic root-of-trust (DRTM) provided by the TrustVisor to enforce strong isolation from the upper hypervisor. The property-querying module can also be implemented with this structure. This integration of TrustVisor allows the implementations of more advanced controls, while maintaining the NeuronVisor in a very thin layer.

Both TrustVisor and BitVisor have compact designs, which allow them to be implemented with 20 kilo lines of code (KLOC) [91] and 6 KLOC [102] respectively. Core NeuronTrust algorithms are implemented in the simulator with less than 2 KLOC; and we envisage less than 1 KLOC will be added to port them into real systems: adding codes for sending/analysing real network packages. On the other hand, TrustVisor imposes less than 7% overheads to protect security-sensitive code blocks. Though the evaluation against the network para-passthrough module on BitVisor is not available, they imposed around

36% on their ATA para-passthrough driver for intercepting and encrypting storage data. As most of the overheads is caused by the encryption operations, we envisage much smaller overheads for implementing the network traffic monitoring.

The design of NeuronVisor allows a highly decoupled layer to separate the trust management from the cloud infrastructure management. The benefits include: 1) Minimized management overheads, as attestations and trust evaluations are performed in an autonomous way; 2) backwards compatibility, as no extra modifications to the vTPM interfaces to the upper layer services are introduced; and 3) separation-of-powers, as trust layer can be operated without the interference from the infrastructure layer. We leave the details implementation and experiments of NeuronVisor to our future work.

5.5 Evaluations

In this section, we first discuss the threats to our NeuronVisor framework, and how it defends against them. We then evaluate NeuronVisor with simulations.

5.5.1 Security Analysis

With cloud attestations, customers are able to identify all the services that have interacted with their cloud applications. The properties of these services will also be examined. Therefore, by matching the attested properties with the presumed service level agreements, customers will determine whether their required services are genuinely enforced. Moreover, unrecognised programs will also be identified, as their properties cannot be located, or be defined as malicious. This helps customers to determine malicious cloud providers or customers. Further discussions on how remote attestation help determining genuine software behaviours can be found in [27, 26, 58, 28, 29, 60].

Therefore, when malicious parties only have privileges to tamper with the cloud infrastructure (cloud management, computing, or other functional supporting services) or customers' application, but not enough privilege to tamper with the lower-layer NeuronVisor (Figure 5.1), their behaviours will be identified by remote attestations. We will evaluate how effectively the NeuronVisor identifies malicious behaviours with simulations in the next section.

On the other hand, different from the centralized attestation scheme in most trusted cloud proposals, attestations in NeuronVisor are enforced autonomously. Neuron Kernels are maintained in a peer-to-peer manner. Therefore, well-known attacks [44] against this

decentralized trust management scheme need to be considered. Here we consider the scenarios when malicious parties gain enough privileges to tamper with the NeuronVisor directly, and trying to circumvent the decentralized attestations enforced by the NeuronVisor: 1) *Self-promotion.* A tampered Neuron's goal is to improve its own connection strength value in the view of others, hence preventing it from being attested to. It can do this by giving (i) positive evaluation to Neurons who have given it positive ratings, and (ii) negative evaluation to Neurons who have given it negative ratings. 2) *Slandering.* The goal is to ruin the reputation of a target Neuron N_x. The options are giving (i) negative evaluation to N_x, (ii) positive evaluation to Neurons who have given negative ratings to N_x, and (iii) negative evaluation to Neurons who have given positive ratings to N_x. 3) *Sybil attacks.* A malicious Neuron creates new identities. These give positive evaluation to it and to each other in order to improve their healthiness (self-promotion). Then, a slandering attack can be launched with the help of the new identities. 4) *Collusive attacks.* Similar to Sybil attacks, but the attackers gain multiple Neuron identities by controlling multiple tampered Neurons. These Neurons then collaborate together to implement self-promotion or slandering.

In NeuronVisor, a Neuron Kernel is updated with *trust aggregation* and *dissemination*. The the kernel of a target Neuron is only merged after the target is attested to. Hence only kernel from a genuine Neuron is aggregated. On the other hand, a Neuron can only disseminate attestation results it made. Thus self-promotion is avoided. To avoid slandering, reporting a target Neuron as "unhealthy" (the negative evaluation) will result in it being attested to by a management authority. When a false report is discovered, the reporter is attested to. This helps to identify the slandering initiator. Finally in NeuronVisor, the Neuron identity is represented by the TPM identify, which can only be created by a Trust Third Party. Thus Sybil attacks are prevented.

Collusive attacks need further examinations. As we assume that the Neurons have identical implementations, as long as attackers have successful exploited one Neuron, it is not hard for them to take control of more Neurons with the same techniques. When a large number of tampered Neurons exist, they disseminate false trust information to promote the connection strength of each other. This may result in other Neurons regarding them as also "healthy". To guard against this attack, NeuronVisor uses the Transitive Trust to calculate the strength value. Thus only when the dissemination source Neuron has a higher credibility, will its reported trust information be trusted. Moreover, during cloud attestation, as the centre Neuron is attested to by customers, the collusive Neurons will ultimately be identified. In this case, the larger the collusive group is, the higher chance it will be discovered. As a result, the damage of this attack is well controlled. The effects of the attacks to NeuronVisor are also examined with simulations next.

5.5.2 Simulations

5.5.2.1 Simulation Settings

We propose to examine the effectiveness and efficiency of NeuronVisor by simulations. Simulations implement a straightforward communication abstraction, and help better illustrate the cloud dynamics, especially when a large-scale TPM-equipped cloud infrastructure is hardly available for experiments. They are also the conventional methods for evaluating decentralized trust management protocols in the peer-to-peer community [45, 103]. We leave the formal validations of our Neuron Web models, and the evaluations with real system implementations to our future work.

We extend the RepCloud simulator [80] for NeuronVisor simulation. In our experiments, we specify a minimum attestation interval. In other words, when a second attestation request to a same node arrives with in the interval, the node returns the last attestation ticket [33], as its TPM halts when returning the previous request. This minimum interval defines the *step* for modelling direct trust.

Basic simulation settings are listed in Table 5.1. *VM Interaction factor* determines how many hundreds of interactions a VM has to the others in a simulation cycle. Thus a factor of 3 means each VM has 300 interactions to other randomly chosen VMs from the same Cloud application. We define an interaction as a basic communication unit. Each interaction takes 1 millisecond to fulfil, which is the basic time unit in our simulation. The *length* of an application is the total simulated time it will run. After an application terminates, a new one is deployed to balance the load of the cloud. This application scheduling also changes the communication patterns inside the cloud.

We evaluate NeuronVisor with the *Tampered Interactions (TI)*. TI occurs when a node interacts with a tampered target before the attack has been identified. For example, when a node is attacked at time t_a, which is in between two consecutive attestations at t_1 and t_2 respectively ($t_1 < t_a < t_2$), all the interactions with the node during the time period between t_a to t_2 are regarded as tampered. We evaluate NeuronVisor with Transitive Trust (NT) by comparing its TI counts with a Centralized Attestation Scheme (CEN) with the same simulation configuration.

The centralized attestation scheme (CEN) is similar to the scheme described in the previous chapter. In CEN, a central attestation delegate attests to every node inside the cloud infrastructure repeatedly with a predefined *interval*. As in NT, we do not consider the dissemination of the upper-layer services' properties, (i.e. NT only exchanges the *Direct Trust* values), the simulated CEN also does not broadcast its gathered attestation result, as opposed to the CEN simulation in the previous chapter. Here, CEN examines, in every

Table 5.1: Basic Simulation Settings

Simulation	
Length of a simulation cycle (minutes)	1
Total simulation time (hours)	2
Minimum Attestation Interval (second)	1
Network	
# of clusters in the cloud	1
# of nodes per cluster	50
# of maximum VMs per node	16
Cloud Apps	
App generation interval (minutes)	10
average VMs per App	4
average length per App (minutes)	10
average VM Interaction Factor	3

interval, whether all the nodes will report their PCR values, which are listed in a predefined white-list. On the discovery of any PCR values violations, the central attestation delegate will reinitialize the node immediately. Meanwhile, it will calculate the TI for this node by counting its total interactions from other nodes from its last attested time to the time when this violation is discovered.

5.5.2.2 Fault Detections

In this simulation, attackers only tamper with the integrity of upper layer services. The NeuronVisor on a tampered node is still genuinely enforced. This simulation evaluates NeuronVisor's capability of identifying services' property changes inside the cloud. This kind of attacks are performed by most dishonest providers or malicious customers, who have gained certain privileges inside a cloud, but are unable to change the lowest Neuron layer.

We first examine how the strength threshold (Φ) affects TI. Φ determines the attestation timing. Before a node communicates to another, its Neuron will attest to the other's when the strength of its connection to the target is below Φ. As the connection strength is calculated from the time *steps* apart from a given time, a corresponding attestation *interval* is deduced for the centralized attestation scheme. Therefore, regarding our simulation setting ($step = 1$), a Φ is mapped to an $interval$ value with the following equation.

Figure 5.4 illustrates the Tampered Interaction counts of NeuronVisor (NT) simulations with different Φ values. The TI counts for the centralized attestation scheme (CEN)

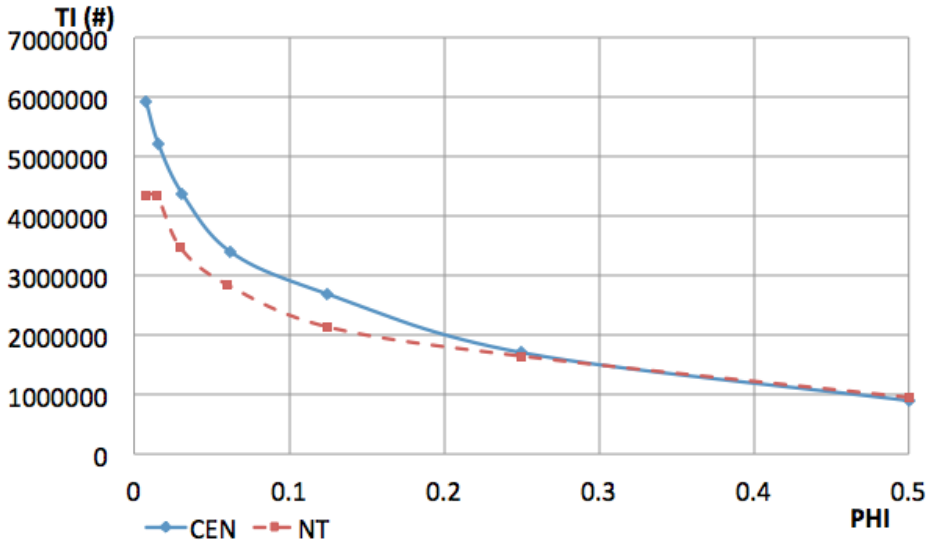


Figure 5.4: Tampered Interaction Counts under Different Connection Threshold Settings. (Total interaction counts for all experiments are the same: 1.04E+08.)

with equivalent $interval$ value is also presented. As seen from the figure, when Φ is low, NT achieves less TI counts with equivalent settings. This is because NT performs attestations according to communication needs. Thus attestations are distributed to better match communication patterns. For example, more attestations are performed to a Neuron when communications to it are more frequent. This results in less TI counts when the target node is tampered with, as the attacked will be detected quicker. However, CEN uniformly performs attestations to each node regardless their interaction semantics.

When attestation interval approaches the attestation interval limit (1 second in our simulation, and thus Φ equals 0.5), NeuronVisor achieves almost the same TI counts with CEN. This is because the NT's more frequent attestations to critical nodes only return the cached tickets. Therefore NT cannot detect the faults happened within the minimum interval. On the other hand, we argue that, for a real system, it is uncommon for a CEN scheme to approach the limit attestation interval. Hence NT produces better results for common cases.

We now examine whether NeuronVisor's effectiveness is determined by the simulation settings. We set Φ as 0.25 for NT and *interval* as 2 seconds for CEN. We first increase the interaction factor. In Figure 5.5 the larger *Interaction Factor* results in larger total interactions for a simulation. Thus each attack affects more interactions and results in larger TI counts. However, as shown in the figure, the differences between NT and CEN remain the same. We also change the average application size, which influences the trust dissemination scale for NT. However, in Figure 5.6, the differences between NT and CEN

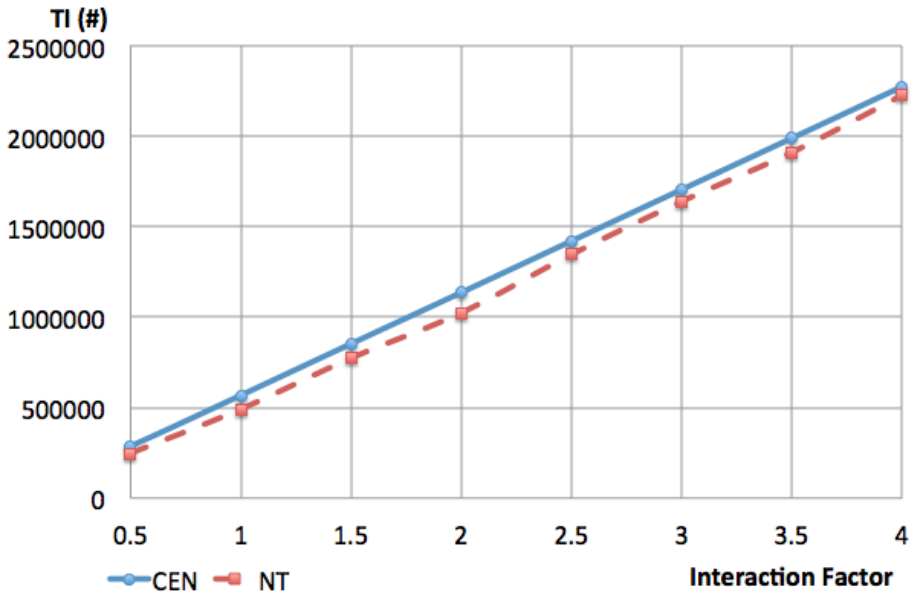


Figure 5.5: Tampered Interaction Counts under Different Interaction Frequency (IF) Settings. ($\Phi = 0.25, Interval = 2s$. Total interaction counts are the same for each equivalent NT and CEN under a same IF setting.)

still remain stable.

5.5.2.3 Targeted Attacks

In this evaluation, the Neurons on the attacked nodes are disabled. They do not perform attestations to the others, and do not disseminate trust. All the upper layer communications are enforced without evaluating the trustworthiness of the target node. On the other hand, we assume a tampered Neuron is fixed as soon as it is attested to. In a real system, this is achieved by re-instantiating the tampered node or replacing it with a backup. The fixed Neuron retrains all the capabilities for attestation and trust dissemination.

In each simulation, we increase the attack ratio, which indicates how many nodes are tampered in every attacking interval. We set the interval to 30 seconds to allow enough time to see how NT recovers. Thus attack ratio 0.5 means in every 30 seconds, 50 percent of nodes (25 regarding our simulation) are tampered at the same time. When all the nodes are tampered at any point of time, the simulation stops.

As shown in Figure 5.7, the differences on the TI counts between NT and CEN increase as the attack ratio increases. This is because the more Neurons are tampered, the fewer nodes are attested to. When 90% of Neurons are tampered, around 40% more interactions are tampered than the CEN scheme. But NeuronVisor is still able to identify all tampered nodes, as the simulation run through the whole simulation. Regarding the CEN

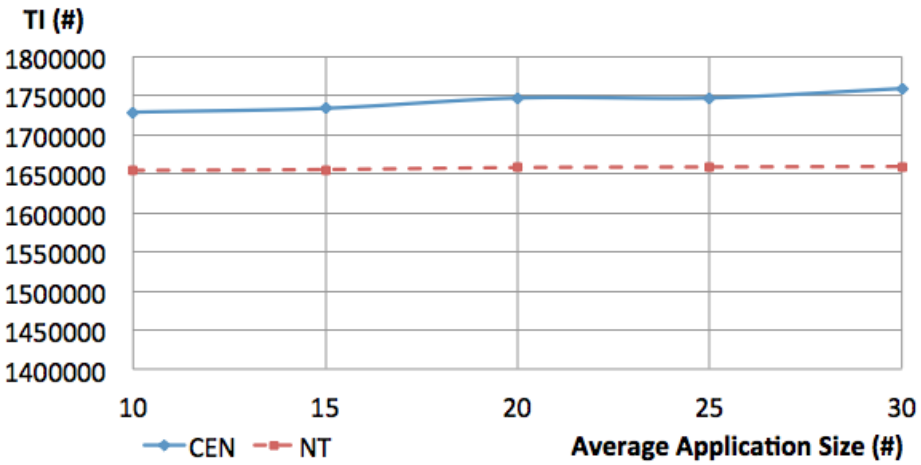


Figure 5.6: Tampered Interaction Counts under Different Average Application Size (AAS) Settings. ($\Phi = 0.25, Interval = 2s$. Total interaction counts are the same for NT and equivalent CEN under a same AAS setting.)

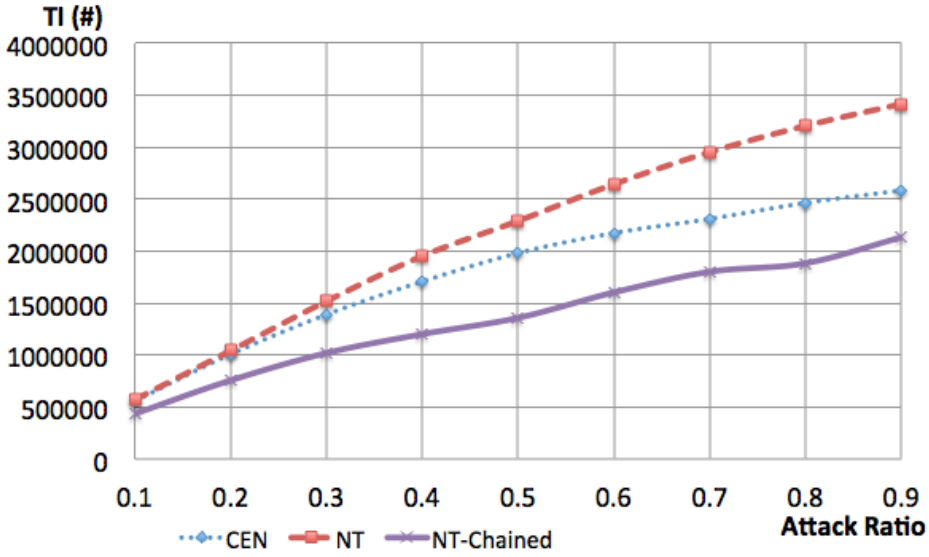


Figure 5.7: Independent Attacks to NeuronVisor. (Total interactions counts are the same for different attestation schemes under a same simulation configuration.)

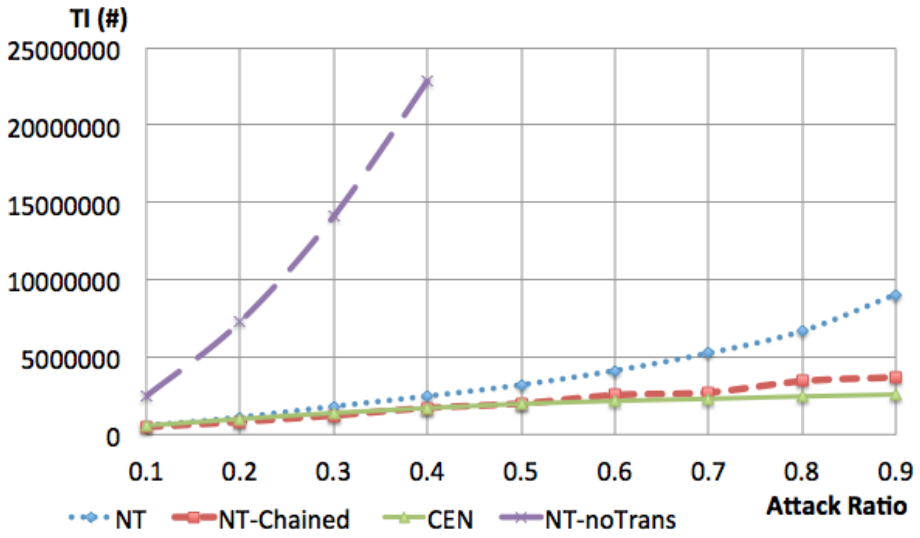


Figure 5.8: Malicious Collusive Attacks to NeuronVisor. (Total interactions counts are the same for different attestation schemes under a same simulation configuration.)

scheme, through it achieves lower TI count, we did not simulate the attacks to the centralized delegate. In this scheme, the delegate becomes the target of a single-point-of-failure. Any single attack to the delegate will disable the entire CEN, and result in all subsequent attacks to CEN nodes not being detected.

On the other hand, we present an enhancement to the NeuronVisor protocol, the NT-chained. When discovering a tampered node (N_a), NT-chained performs additional attestations to all the node's neighbour, as these neighbours' were less likely to be attested to, since N_a were tampered and were unable to perform attestations. Hence they are more likely to be tampered as well, especially under high attack ratios. As depicted in the figure, NT-chained achieves even better TI detection performance than the CEN scheme.

5.5.2.4 Collusive Attacks

In this simulation, tampered Neurons maliciously collides with each other. They do not perform attestations, but disseminate to all others high direct trust values for the tampered ones. Thus by tampering the trust evaluations, the malicious collusive decreases the probability for the "healthy" Neurons to attest to their members. In Figure 5.8, NeuronVisor incurs a higher TI counts difference to the CEN scheme. When 90% Neurons are tampered and maliciously collaborate with each other, more than one time of tampered interaction counts are incurred. However, as the Transitive Trust is calculated as the strength value, the damage of the attacks is still controlled. NeuronVisor still discovered all tampered nodes and carried on the simulation to the end. On the other hand, for the NT-noTrans scheme,

which sets the strength value to a target Neuron as the largest direct trust value any other Neuron has calculated against the target, the TI counts raise very fast. And all the Neuron will be tampered before the end of a simulation when the attack ratio is above 0.4. Besides, the NT-Chained enhancement still exhibit good TI detection rate, though its TI counts are slightly higher than the CEN counterparts when the attack ratios are high. As it is not common for a real cloud system to regularly have a very high attack ratio (e.g. $> 50\%$), we believe that the NT-Chained enhancement is suitable for most scenarios.

5.5.2.5 Overhead

In our simulations, to achieve the same TI level NeuronVisor incurs 4 to 10 times of attestation counts than the CEN scheme. The overhead varies as different simulation configurations are changed, e.g. the average application size, the interaction factors or the cloud scheduling policy. The overhead is mainly caused by the Transitive Trust calculation, as it only reuses the attestation results from a trustworthy Neuron. However, as the attestations are performed by all the nodes altogether, NeuronVisor distributed the attestation burden from a single centralized delegate. It reduces the risk for a single-point-of-failure, while increasing the stability and computing capabilities for implementing attestations. Moreover, as fine-grained cloud attestations are supported, we argue that the overhead is well compensated by the gained benefits. Besides, the autonomous and decentralized nature of NeuronVisor significantly reduces the management overhead for maintaining a trusted computing infrastructure.

5.6 Summary

In this chapter we proposed the NeuronVisor framework to define a fine-grained *Cloud Root-of-Trust* for effectively attesting to the *cTCB* of a target application. NeuronVisor adapts the *Decentralized Attestation* proposed in Chapter 4, but only uses it to attests to a simple property: whether a Neuron is capable of genuinely attesting to the upper-layer services. This simplicity eases the trust implication modelling. We therefore designed the *Connection Strength* model to represent the trustworthiness of a target Neuron. Experiments showed that the *Connection Strength* model helps to efficiently discover malicious behaviours, and resists to classic attacks to reputation-based systems. In the next chapter, we will present how the *cRoT* models helps implement the *Cloud Chain-of-Trust* abstractions.

Chapter 6

Cloud Chain-of-Trust

6.1 Overview

In Chapter 4, we presented the concepts of the *Cloud Trusted Computing Base (cTCB)*. The *cTCB* facilitates fine-grained dependency discovery and attestations for cloud applications. With the *Decentralized Attestation (DA)*, the centralised attestation delegates are able to identify the cloud services that have supported or affected the genuine behaviours of the target cloud applications. These delegates will then gather sufficient trust evidence (the attestation tickets for all the corresponding cloud nodes) to attest to the cloud customers the trustworthiness of their cloud applications and the correspondingly dependency components in the cloud, i.e. the *cTCB*. However, as these delegates are implemented and managed by the CSPs themselves, their trustworthiness and effectiveness are relatively complicated to verify.

To overcome this issue, in Chapter 5, we presented the concept of the *Cloud Root-of-Trust (cRoT)*[1]. The *cRoT* sits at the lowest layer of a cloud application's *cTCB*. It only represents the *cTCB*'s one simple property: whether it can genuinely report the trust evidence of the upper-layer services. With the *NeuronVisor* system, the *DA* is also employed to identify the cloud nodes, hosting the *cTCB* of the target cloud application. But it is only enforced to attest the integrity of these nodes' lowest TPM managing layer, i.e. the *NeuronVisor*, instead of to the nodes' *Core Services* when using the *RepVisor*. Therefore, the purpose of the *cRoT* is to allow the customers to implement their own attestation delegates inside the cloud. These delegates will identify the nodes hosting the target cloud applications' *cTCB*, and examine whether the nodes will reliably report the genuine behaviours of their upper-layer services, without the need to obtain the nodes' hardware identities, such as their TPMs' AIK (Section 2.1).

[1] As discussed, in this work, we only consider the Root-of-Trust for Measurement.

In this chapter, we will discuss how the trust evidence of these upper-layer services are collected and exported to the customers with the help of the *cRoT* abstractions. Typically, we will present the method to measure the properties of a cloud application and its *cTCB* to a *Chain-of-Trust (CoT)* (Section 2.1). We denote this *CoT* as the *Cloud Chain-of-Trust (cCoT)*. In particular, we consider the following issues:

- *Multiple-Roots Issues.* A *cCoT* is effectively a set of *CoTs* measuring the properties of all the nodes hosting the cloud application's *cTCB*. However, if the *cCoT* is constructed as an array of individual *CoTs*, it will have an array of measurement values. In this case, the attesters will have to verify the trustworthiness of each corresponding *RoT* for examining the trustworthiness of each value. This will significantly increase the *AIK* management complexity. It also reveals the underlying hardware's identities.

- *Redundancy.* A *cCoT* contains redundant *CoTs* representing a same set of properties. This is because the homogeneity still exists in a cloud infrastructure. Nodes implementing the same services may have a same *CoT*. For example, considering the OpenStack infrastructure, a *Scheduler* service communicates with multiple *Compute* services which have identical platform configurations [88]. The *CoTs* of all these *Compute* services are counted as part of the *Scheduler* service's *cCoT*, as the *Compute* services are recognized as *Scheduler*'s the dynamic dependency. Redundancy adds to the complexity for maintaining and attesting the *cCoT*.

- *Anonymity Violations.* From each *CoT*, the attester is able to differentiate a service provider entity (i.e. a *resource*) from the others. For example, the attester can determine whether two services are implemented by one node, or whether two VMs have a same host. This contradicts the goal of the cloud abstration, which is to hide the underlying detail and expose a unified logical resource presentation. Violations in anonymity also facilitates targeted attacks [76].

To overcome the above problems, our goal is to measure a *cCoT* as a single-rooted tree, with redundant properties reduced and individual resources' identities hidden. A *cCoT* thus represents the combined properties for a single *logical* resource, which provides all the corresponding cloud services to support a VM's lifecycle inside the cloud. In this chapter, we discuss how to achieve this goal with help from the *cRoT* abstration.

In the rest of this chapter, we first present the concepts of the *Compositional Chain-of-Trust (comCoT)* in Section 6.2. A *comCoT* combines multiple *CoTs* into a single chain with the help of the *Neuron Connections*. It is the foundation for constructing a *cCoT* abstraction. We further define the mathematical models for the *comCoT* and the *cCoT*

respectively. In Section 6.3, we present the *NeuronTPM* framework. The NeuronTPM is an extension to the NeuronVisor. It manages and reports the trust evidence for the upper-layer services managed by the NeuronVisor. The NeuronTPM helps measuring the *cCoT* of each VM, and exports the related measurement values and measurement logs through its *vTPM* interfaces. We will explain how cloud attestations with the *cCoT* can be achieved with our case study in the next chapter.

6.2 Cloud Chain-of-Trust Definitions

We identify two types of *CoTs* in a cloud environment: a *Resource Chain-of-Trust (rCoT)* and a *comCoT*. An *rCoT* is the *CoT* for a single resource. A *resource* is a conceptual entity, which provides services to other entities. In our context, we identify three kinds of resource: the *NeuronVisor*, the *Core Services* of a node and the VM. We will examine these resource in the next section.

A *comCoT* is the composition of multiple *rCoTs*. It represents multiple resource with a single abstract *rCoT*. Evaluating a *comCoT* is a verifier's main focus, as understanding the detailed construction of each individual *rCoT* will incur large complexity, due to the dynamism of a cloud system.

Two types of *comCoT* are considered, a *Domain Chain-of-Trust (dCoT)* and a *Collaboration Domain Chain-of-Trust (cdCoT)*. A *dCoT* reduces redundancy by combining multiple identical *CoTs* a single one. It also hides the individual identity of the resource that provides the same service. On the other hand, a *cdCoT* combines different *CoTs* into a logical one. It thus represents a logical resource which possesses the aggregated properties represented by combined *CoT*. It further hides the identities of a set of collaborating resource.

Based on these abstractions, we define the *Cloud Chain-of-Trust (cCoT)* for a VM as a *cdCoT*, which is comprised of all the collaborating resource in the cloud to support the VM. Therefore, the *cCoT* for a VM is built by iteratively identifying and constructing *dCoTs* and *cdCoTs* from all the components of the VM's *cTCB*. This section defines the *rCoT* and *comCoT* models based on the *Cloud Root-of-Trust* abstraction. These definitions further provide the building blocks for defining the *cCoT*, which is presented afterwards.

6.2.1 Resource Chain-of-Trust

We first define the *CoT* for a single Resource (*rCoT*) as a triple comprising an initial trust function (itf), a set of trust function (stf) and a sequence of elements in the chain (*sq*. $\langle x_0, x_1, \cdots, x_n \rangle$). x is an element representing any software or hardware component that composes the chain of the loading sequence, which is responsible for: 1) bootstrapping the corresponding resource components to an expected state, and 2) implementing their functionalities. We define a *trust function* as an operation that is used by a trusted software component to genuinely measure and verify the integrity of another software component that it will load. The *itf* takes one parameter, the x_0. It will be carried out by an assumed trusted component to measure and verify the x_0. The *stf* takes two parameters, with the first one enforcing the *stf* to measure and verify the second one. We will explain the *itf* and *stf* later.

A *trustworthy rCoT* requires: i) the initial trust function (*itf*) evaluates the first element of the sequence as *trusted* or *assumed trusted*; and ii) every function in the set of trust functions (*stf*) returns *trusted* when applied to any two consecutive elements in the sequence (*sq.*). This is formally defined as follows:

$$\begin{aligned} \textit{rCoT} = (\textit{itf, stf, sq}\langle x_0, x_1, \cdots, x_n \rangle \mid \textit{itf}(x_0) \in \{\textit{trusted,assumed trusted}\}, \\ \forall i : [1..n] \bullet \forall f : \textit{stf} \bullet f(x_{i-1}, x_i) \quad == \quad \textit{trusted}) \end{aligned} \tag{6.1}$$

The *RoT* (i.e. the first element in the sequence, x_0) is defined based on the type of the entity and its location within the cloud's different layers. In the context of TCG specifications, the *rCoT* should start from a CRTM (Core Root of Trust for Measurement), which should be stored in protected location (currently it is protected by the BIOS). Once the CRTM measures the platform initial environment state, it stores the result in the protected registers inside the TPM (i.e. the *PCRs*). The CRTM represents the *RoT*, x_0, and the *stf* contains the TCG root of trust functions (that is RTM, RTS and RTR) and other functions, such as the measurement methods used by the IMA [74]. The initial trust function (*itf*) is the one that measures the CRTM itself and stores the result inside the TPM's *PCRs*.

We define the resource having CRTM as its x_0 as a *physical resource*. Any other resource is defined as a *virtual resource*. In general, a virtual resource is a set of software, which has its *itf* and *stf* clearly defined. Determining the trustworthiness of the *rCoT* for a virtual resource requires its x_0 be ultimately verified by a hardware *itf*, as we only assume the tamper-proofness of a hardware *RoT*, i.e. the TPM. This is achieved by linking the *itf* to the *stf* of its underlying hosting resource's *rCoT*. This hosting resource can either be a

physical resource, or a virtual resource. In the later case, iterative *CoT* linking is expected until a physical resource is presented. Here we define the *extend* operations over *rCoTs* as:

$$\begin{aligned} \mathit{extend}(\mathit{rCoT}_1, \mathit{rCoT}_2) = (\langle \mathit{itf}_1, \langle \mathit{stf}_1, \mathit{stf}_2 \rangle, \langle \mathit{sq}_1, \mathit{sq}_2 \rangle \rangle \\ \mid \mathit{stf}_1(x'_0) \quad = \quad \mathit{itf}_2(x'_0)) \end{aligned} \tag{6.2}$$

This operation denotes that the two *rCoTs* are linked into a single one, with the *itf* of the second one implemented by the last *stf* of the previous one. Therefore, this operation effectively *extends rCoT* from the elements in rCoT_1 to elements in rCoT_2. Attesting to the extended *rCoT* verifies the properties of both *rCoTs* altogether. For example, in order to extend the *rCoT* from a VM's host ($\mathit{rCoT}_{\mathit{Host}}$) to the VM ($\mathit{rCoT}_{\mathit{VM}}$), the host should measure x_0 of the $\mathit{rCoT}_{\mathit{VM}}$, which is usually first component of the VM's bootstrapping procedure. For example, the SeaBIOS in the Qemu VM loading sequence.

6.2.2 Compositional Chain-of-Trust

We now define the *Compositional Chain-of-Trust* (*comCoT*). A *comCoT* represents a group of entities with a single Chain-of-Trust. This is important in a cloud environment, as many services exist as a composition of multiple entities, e.g. a cluster of physical servers. Members of such groupings may have identical or different CoTs. However, to an entity depending on this grouping, it should see a single logical *CoT* representing the trust it can have in the grouping. In other words, relying entities will see a single entity abstraction, even though that entity will be a grouping representing multiple entities. This abstraction will help to define the *Cloud Chain-of-Trust*.

We first identify two types of cloud service configurations: homogeneous and heterogeneous. In a homogeneous setting, all resources have the same *rCoT*. They are configured uniformly to work together to provide one same service. We define this set of resources as a *Domain*. An example of this is the resource providing the Infrastructural Services (IS). Members of a *Domain* (e.g. all the *Compute* nodes in an OpenStack cloud cluster) are identical and carefully selected, interconnected and positioned to achieve the domain properties. Supporting Services and Business Logic Services, on the other hand, are heterogeneous, as they are composed of resources providing diverse services, which have different *rCoTs*. When these services collaborate together to achieve a common goal, they form a *Collaboration Domain*.

Two types of compositional *CoT* are hence defined according to the structural differences of the cloud services, namely: the *Domains Chain-of-Trust (dCoT)* and the *Collaboration Domains Chain-of-Trust (cdCoT)*. In order to achieve the presentation of a single

abstract *rCoT*, the *dCoT* or the *cdCoT* should also contain two components as an *rCoT*: i) a Root-of-Trust and ii) a sequence of entities, supporting or implementing the services provided by the corresponding Domains or Collaboration Domains. This requires a *dCoT* or a *cdCoT* to have a unique *itf* for defining the *RoT*, and a set of *stf* to extend the trust from the *RoT* all its entities.

However, unlike *rCoT*, the Root-of-Trust of *dCoT* or *cdCoT* attests to the trustworthiness of the way the Domain or Collaboration Domain is managed and operated. We need a *RoT* to satisfy two properties: i) its trustworthiness can be measured and assessed at all times, and ii) it can provide strong assurance about the trustworthiness of the way Domains and Collaboration Domains are managed and operated.

A straightforward way of combining multiple *CoTs* to a logical one is to attest to them separately and return *trusted* only when all of them are *trusted*. In this case the *itf* and the *stf* for the combined *CoT* are defined as wrapper functions, which invoke the *itf* and *stf* of each *rCoT* respectively. The *combine* operation for building a *dCoT* and *cdCoT* from separate *rCoTs* is defined as follows. For the ease of representation, we denote *combined CoT* as the set of all *rCoTs*. In this context, the combine operation is idempotent, commutative, and associative, so we will use terms such as $combine(X, Y, Z)$ since these are unambiguous.

$$\begin{aligned} combine(rCoT_1, rCoT_2) &= \langle\langle itf_1, itf_2\rangle, \langle stf_1, stf_2\rangle, \langle sq_1, sq_2\rangle\rangle \\ &= \langle rCoT_1, rCoT_2\rangle \end{aligned} \tag{6.3}$$

6.2.2.1 Domain Chain-of-Trust

We now discuss how to construct a single-rooted *CoT* from the combined *rCoTs*, in the context of our *Cloud Root-of-Trust* abstractions. In the last chapter, we defined the *Connection Strength* model, which represents how the Neuron of a node is attested to by another Neuron. Neuron Connections are formed according to upper layer communications. Therefore, as long as the resource has communicated with each other, their underlying Neurons will connect to form a Neuron Web, which represents an abstract *Cloud Root-of-Trust (cRoT)* for all the resources. This *cRoT* abstraction is represented by the *RoT* of any single Neuron on the web, which is identified as the *Entrance* of the web. The other Neuron's *RoT* is deduced by the connections strength values rooted from the entrance Neuron.

In order to implement *combine* with the help of the connection strength, we first define the *connection trust function (ctf)*. *ctf* evaluates whether the C_{ij} of a target Neuron N_j satisfies certain criteria. This evaluation can be implemented by comparing C_{ij} with a reference value. Here we do not restrict the formula of this evaluation.

We therefore define the *merge* operation as a special implementation of *combine*: it *combines* two identical *rCoTs* ($rCoT_1$ and $rCoT_2$) to a single one, when C_{12} satisfies the *ctf*. The result is a modification to $rCoT_1$: *ctf* is added to the end of its *stf*, and C_{12} is added to its sequence. We denote this modified $rCoT_1$ as $rCoT'_1$.

$$\begin{aligned} merge(rCoT_1, rCoT_2) &= (rCoT'_1 \mid rCoT_1 = rCoT_2) \\ &= \langle itf_1, \langle stf_1, ctf \rangle, \langle sq_1, C_{12} \rangle \rangle \end{aligned} \tag{6.4}$$

With this operation, the *dCoT* is defined by iteratively examining the C_{ik} for each N_k in the domain, and merging their *rCoTs*. The result is the *rCoT* of any chosen node in the domain, e.g. $rCoT_1$. Moreover, when iteratively merging a list of *rCoTs*, only the smallest connection strength value is added to the *sq*, instead of the list of different values.

$$\begin{aligned} dCoT(rCoT_1, rCoT_2, \cdots, rCoT_n) &= merge(rCoT_1, merge(\cdots, rCoT_n)) \\ &= rCoT'_1 \end{aligned} \tag{6.5}$$

On the other hand, when *merge* is applied to two different *rCoTs*, it returns the *combined rCoT* of these two:

$$\begin{aligned} merge(rCoT_1, rCoT_2) &= (combine(rCoT_1, rCoT_2) \mid rCoT_1 \neq rCoT_2) \\ &= \langle rCoT_1, rCoT_2 \rangle \end{aligned} \tag{6.6}$$

We further extend this definition to the case when *merge* is applied to a list of *rCoTs*. It returns a *combined* list of *rCoTs*, with each *rCoT* representing a distinct *dCoT* formed by all the identical *rCoTs* in the original list:

$$\begin{aligned} merge(rCoT_1, rCoT_2, \cdots, rCoT_n) &= combine(dCoT_i, dCoT_j, \cdots, dCoT_k) \\ &= \langle rCoT'_i, rCoT'_j, \cdots, rCoT'_k \rangle \end{aligned} \tag{6.7}$$

As in Figure 6.1, we assume that N_1 is depending on $N_{[2-6]}$. N_3 and N_4 have identical *rCoTs*, while N_5 and N_6's *rCoTs* are also identical. By applying the *merge* operation to $rCoT_{[N_2-N_6]}$, N_1 is now depending on the *rCoTs* of N_2, N'_3 and N'_4. We now propose how to combine $rCoT_{[N_1,N_2,N'_3,N'_4]}$ as one logical CoT.

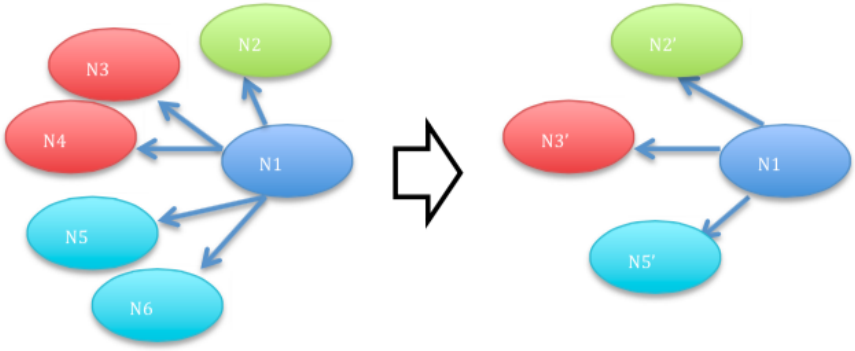


Figure 6.1: Domain Chain-of-Trust.

6.2.2.2 Collaboration Domain Chain-of-Trust

We now define the *Collaboration Domain Chain-of-Trust (cdCoT)* to create a single *CoT* to represent multiple different *rCoTs*. To achieve this, we also use the connection strength value as the *RoT* factor, as verifying C has the same effectiveness of examining the *RoT* a target node at an implied interval. Therefore, according to the definition of the *extend* operation, when the *RoT* of a second resource is examined by the first one, its *rCoT* can be *extended* to the first one's *rCoT*. In this case, C represents the x_0 of the second *rCoT*, and the *ctf* is added to the first *rCoT*'s *stf* for verifying C.

As *ctf* and C are inserted into the *stf* and the sequence of the resulting *rCoT* respectively, the *extend* operation cannot apply directly. We therefore define the *connect* operation, which is a special implementation of *extend*: it *extends* an *rCoT* to another when the C to the connected one satisfies certain criteria.

$$connect(rCoT_1, rCoT_2) = (\langle itf_1, \langle stf_1, ctf, stf_2 \rangle, \langle sq_1, C_{12}, sq_2 \rangle \rangle \tag{6.8}$$

Given this definition, *cdCoT* for a Collaboration Domain is defined by choosing an initial node (N_1) in the domain, and iteratively *connecting* the *rCoT* of the other nodes to $rCoT_1$. To reduce redundancy and hide resource identity, *dCoTs* are firstly identified with the *merge* operation.

$$\begin{aligned} cdCoT(rCoT_1, rCoT_2, \cdots, rCoT_n) & \qquad (6.9) \\ &= connect(merge(rCoT_1, rCoT_2, \cdots, rCoT_n)) \\ &= connect(dCoT_i, dCoT_j, \cdots, dCoT_k) \\ &= connect(rCoT_i, connect(\cdots, rCoT_k)) \end{aligned}$$

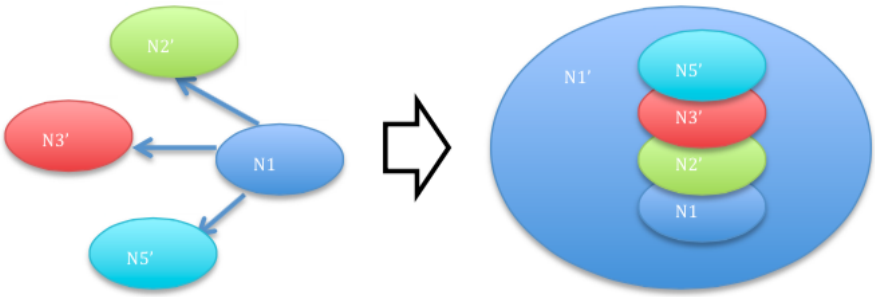


Figure 6.2: Collaboration Domain Chain-of-Trust.

Therefore, in Figure 6.2, to a VM hosted on N_1, it sees a collaboration domain providing the aggregated services implemented by N_1, N_2, N_3', and N_5'. The *cdCoT* for this collaboration domain is defined by chaining $\langle rCoT_{N_1}, rCoT_{N_2}, rCoT_{N_3'}, rCoT_{N_5'} \rangle$ iteratively, as if the later *rCoT* represents a virtual resource hosted on the previous one.

In summary, in order to combine multiple *CoTs* into a single one, we define three operations. 1) The *extend* operation combines the *rCoT* of a virtual resource to its hosting physical resource's *rCoT*. This operation links an *rCoT* ultimately to a physical *RoT*, i.e. the TPM. 2) The *merge* operation combines multiple identical *rCoTs* with satisfactory *Connection Strength* as a single one: the *dCoT*. A *dCoT* represents a Domain containing identical resource providing a same service. 3) The *connect* operation combines multiple collaborating *rCoTs* with satisfactory *Connection Strength* as a single abstract one: the *cdCoT*. A *cdCoT* represents a Collaboration Domain containing different services to serve a same purpose. We now discuss how to use the *cdCoT* concept to construct a *Cloud Chain-of-Trust*.

6.2.3 Cloud Chain-of-Trust

Cloud Chain-of-Trust (cCoT) organizes the trust evidence to represent *cTCB*'s properties. As *cTCB* is defined in a hierarchical manner, *cCoT* is constructed for each level accordingly. We first define the *CoT* of the *Core TCB* of a node as the *coreCoT*.

$$coreCoT_N = extend(rCoT_{Neuron}, rCoT_{CoreServ}) \tag{6.10}$$

Core TCB represents a node's basic functionalities inside a cloud. As the NeuronVisor manages the TPM for each node, $rCoT_{Neuron}$ represents the node's basic hardware *RoT*. *Core Services* mainly include the *Infrastructure Services* implemented by the ISPs (*Infrastructure Service Providers*), e.g. for an *OpenStack* installation, the *Compute* and *Schedule*

etc. Moreover, ISPs usually serve as SSPs (*Supporting Service Providers*) by implementing additional supporting services, which are tightly integrated with their cloud infrastructure to enrich the cloud service catalogues. Therefore, these *Supporting Services* are also included as the *Core Services*. As these services are all deployed on top of the NeuronVisor, the *extend* operation is used to bind these two *rCoTs* together.

The *Cloud TCB* of a node further includes the *Core TCB* of all its *dynamic* and *static* dependencies. *Dynamic Dependency* is the dependency perceived by the *inspectors*, i.e. the NeuronVisors. It includes all the services that have interacted with the target node's Core Services. We define the *dynCloudCoT* of N_i as the *cdCoT* containing all the Core Services (*coreCoT*) in N_i's dynamic dependency set (*DynDep*).

$$\mathit{dynCloudCoT}_{N_i} = \mathit{cdCoT}(\mathit{coreCoT}_{N_j}) \mid N_j \in \mathit{DynDep}(N_i) \tag{6.11}$$

Static Dependency is defined by cloud administrators. It provides services to support the functionalities of N_i's Core Services. We define the set of static dependency nodes for a node N_i as $\mathit{StaDep}(N_i)$. On the other hand, as we assume that services are only enforced when direct interactions occur, the node implementing depending services can only be counted in the *CoT* when it is also a dynamic dependency. This definition eliminates the *inactive* static dependency from the target node's cloud CoT.

$$\mathit{staCloudCoT}_{N_i} = \mathit{cdCoT}(\mathit{cCoT}_{N_j}) \mid N_j \in \mathit{StaDep}(N_i) \wedge N_j \in \mathit{DynDep}(N_i) \tag{6.12}$$

In this definition, the *cCoT* of each node in $\mathit{StaDep}(N_i)$ is used instead of the *coreCoT*. This is because static dependency in fact represents parts of the target node N_i's Core Services' functionalities. Therefore, their *dynamic* and *static* dependency should also be considered as part of N_i's *cCoT*. This recursive definition will cover all the nodes hosting the services to support the target node's Core Services. It also includes all those nodes that have the potential to implement malicious behaviours against the target node, and the nodes hosting its dependent services.

The *cTCB* for node is thus defined as the Collaboration Domain to combine core TCB of the node, along with the node's dynamic and static dependency. According to the definition, the *cCoT* for a node is built by firstly identifying *dCoTs* from both the node's *dynamic* and *static* dependency set, and then iteratively *connecting* these *dCoTs* with the node's coreCoT. With this method, a single *CoT* is formed to represent a *cCoT*. It ultimately starts with the node's NeuronVisor's *RoT*, i.e. the CRTM, and is *extended* to include all the Core Services on the node, and all the coreCoTs of its dynamic and static dependency inside the cloud.

$$cCoT_{N_i} = cdCoT(coreCoT_{N_i}, dynCloudCoT_{N_i}, staCloudCoT_{N_i}) \tag{6.13}$$

We now define the Cloud *CoT* for a VM, $cCoT_{VM}$. On a single virtualization platform, the TCB for a VM contains all the supporting software facilities locally deployed, e.g. the hypervisor and some other management consoles. However, in a cloud infrastructure, to support a VM, a set of core cloud services must exist, which is usually deployed on different nodes. These services are identified by the recursive chain of *Static Dependency* of the VM's host. For example, the *Compute* hosting the VM, and the *Schedule* controlling the *Compute*, and the *Storage* offering the image storage service for the VM. Therefore, the *cCoT* of a VM should include the *cCoT* of the VM's host: $Host(VM)$.

$$cCoT_{VM} = cCoT_{Host(VM)} \tag{6.14}$$

On the other hand, as a VM can be migrated within a cloud, its host changes from time to time. In order to capture the entire dependency of a VM throughout its lifecycle, the *cCoT* of all it hosts should be recorded. As we will present in the next section, this is achieved by aggregating the $cCoT_{VM}$ for the target VM at its critical phases during its lifecycle, i.e. deployment, host updates, and migration.

Finally, we define the *cCoT* for an application. We first define a *Cloud Application* as a set of collaborating customer VMs and an optional set of VMs implementing Supplemental Services, e.g. the *Supporting Services* or the *Business Logic Services*. We denote *App* and *Supplement(App)* as the sets containing the customer owned VMs and all other third-party VMs respectively. The Cloud *CoT* for the application, *cCoT(App)*, is therefore defined as the *cdCoT* encompassing the cloud *CoTs* all VMs.

$$cCoT_{App} = cdCoT(cCoT_{VM_i}) \mid VM_i \in (App \cup Supplement(App)) \tag{6.15}$$

6.3 NeuronTPM Framework

In this section, we present the NeuronTPM framework for building the *cCoT* for each VM. NeuronTPM collects and organizes the trust evidence for *cCoTs*. It measures the evidence and returns the measurements to the attester through the vTPM interfaces. A measurement log is also returned, which records how the *cCoT* measurement is constructed. NeuronTPM

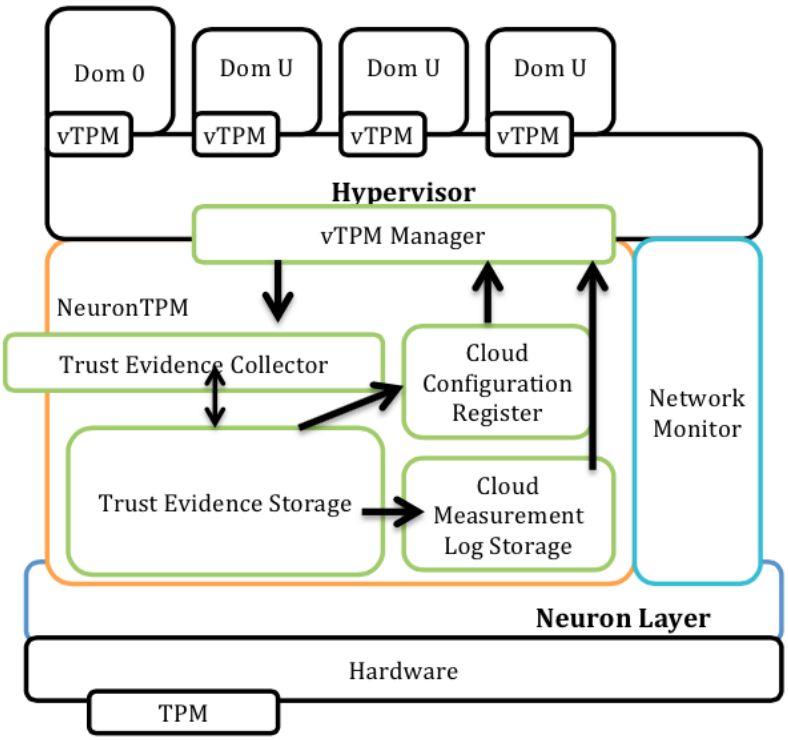


Figure 6.3: NeuronTPM Structure.

creates an illusion that a VM is hosted on a logical server, which provides the combined cloud services, and at the same time, possesses all the potentials in the cloud to perform malicious behaviours. NeuronTPM facilitates effective cloud attestations by hiding the complexity, heterogeneity, and dynamism of the cloud.

6.3.1 General Framework

Figure 6.3 depicts the structure of the NeuronTPM. NeuronTPM implements NeuronVisor's vTPM manager component. In order to construct the *cCoT*, the *Trust Evidence Collector*, or *Collector* for short, queries the trust evidence of the dynamic and static dependency nodes of the VM's host. These dependency nodes are identified by NeuronVisor's *Network Monitor*, according to the communication traffic to the Core Services of the host. Therefore, two kinds network inspections are enforced by the *Network Monitor*. 1) It monitors all the traffic to its host, which include the traffic to the Core Services and to all the hosted VMs. This monitoring allows the local Neuron to identify its neighbours for constructing the Neuron Web model. 2) It distinguishes the traffic only to the Core Services. This discovers the functional dependency of a node.

As the Neurons of the dependency nodes are connected to the querying node, the *Collector* simply examines whether the connection strength value satisfies certain criteria, before requesting its peer *Collector* for trust evidence. When the target node is only a dynamic dependency, the *Collector* returns the measurement value of its Core Services as the trust evidence. This is achieved by allowing the *Collector* to query the vTPM Manager for the specified vPCR value. This returned evidence is stored inside the *Trust Evidence Storage* (*Storage* for short). When the target node is the querying node's static dependency, the target node's static dependency is iteratively queried. To facilitate this querying and the trust evidence caching, RepCloud's trust dissemination mechanisms are applied. The difference is that with NeuronVisor, the *RoT* any target node is assumed. Therefore, the trust evidence is transmitted without the need to verify the TPMs' signatures.

The *cCoT* for a node is constructed from all the collected trust evidence, according to the dependency relationship. The measurement value for the *cCoT* of the node is stored in the *Cloud Configuration Register (CCR)*, which will be exported through the vTPM interface to the VMs. The log for how this measurement value is constructed is managed by the *Cloud Measurement Log Storage*. In the following text, we will examine how these components work together to construct the *rCoT*, *cCoT* for a node, and *cCoT* for a VM respectively. Finally, we present the vTPM interface extensions for reporting the *cCoT's* measurement values.

6.3.2 Measuring Resource Chain-of-Trust

Resource *CoT* (*rCoT*) is the basic element for building and measuring a Cloud *CoT* (*cCoT*). Three kinds of *rCoT* are measured in NeuronTPM: *rCoT* for the *NeuronVisor* ($rCoT_{Neuron}$), *rCoT* for Core Services ($rCoT_{CoreServ}$) and *rCoT* for the VM ($rCoT_{VM}$).

$rCoT_{Neuron}$ records the basic measurement matrix for the NeuronVisor. Its component sequence, i.e. the *sq*, contains the software component in the Neuron's boot sequence, as a Neuron is deployed at the lowest layer of a node. In particular, this sequence includes CRTM, BIOS, Bootloader, and NeuronVisor. Their measurement values are extended to the hardware *PCRs*.

According to the TCG specifications, this measurement, $M(rCoT_{Neuron})$, is calculated by iteratively applying the *TPM_Extend* operation on $rCoT_{Neuron}$'s sequence. Here we allow this operation to take the whole sequence as a parameter to represent this iterative process. PCR_{init} is the initial value of the *PCRs* after the platform's fresh boot. It is generally set to 0.

$$M(\textit{rCoT}_{\textit{Neuron}}) = \textit{TPM\_Extend}(\textit{PCR}_{\textit{init}}, \textit{sq}(\textit{rCoT}_{\textit{Neuron}})) \tag{6.16}$$

However, with NeuronVisor, $\textit{rCoT}_{\textit{Neuron}}$ is examined by each interacting Neuron with the Decentralized Attestation. It is not revealed to the upper layer services. During Neuron attestations, each Neuron attests to its peer against the pre-loaded property certificates. The attestation results are disseminated and aggregated to calculate a *Connection Strength* value for each Neuron to represent its trustworthiness. Accordingly, we propose to use the Neuron's Connection Strength value to represent the measurement of an $\textit{rCoT}_{\textit{Neuron}_j}$ in the view of the attesting Neuron N_i:

$$M(\textit{rCoT}_{\textit{Neuron}_j}) = C_{ij} \tag{6.17}$$

$\textit{rCoT}_{\textit{CoreServ}}$ contains the software components to implement and support the *Core Services*. Different hypervisors and infrastructure services may exhibit different properties. To hide the implementation detail and reduce the attack surface, property-based attestations are enforced at this level. In order to confirm to the *Restriction Model with Multiple Entities*, this measurement supports different *SPDs* to install *translators* to perform property translation for a same $\textit{rCoT}_{\textit{CoreServ}}$ sequence. Therefore, the function $\textit{Trans}_{\textit{SID}}$ by the *SPD* identified by *SID* is applied to each entry in the sequence, before the *TPM_Extend* operation is applied. M_0 denotes the initial value for calculating the measurement. It is usually set to 0.

$$M(\textit{rCoT}_{\textit{CoreServ}}, \textit{SID}) = \textit{TPM\_Extend}(M_0, \textit{Trans}_{\textit{SID}}(\textit{sq}(\textit{rCoT}_{\textit{CoreServ}}))) \tag{6.18}$$

Accordingly, for each $\textit{rCoT}_{\textit{CoreServ}}$, a list of measurement values is maintained, recording the different versions translated by different *SPDs*. We denote this list as:

$$M(\textit{rCoT}_{\textit{CoreServ}}) = \{M(\textit{rCoT}_{\textit{CoreServ}}, \textit{SID}_i) \mid \textit{SID}_i \in \{\textit{SIDs for Supported SPDs}\}\} \tag{6.19}$$

$\textit{rCoT}_{\textit{VM}}$ records the measurements for all the software components in a VM. It usually contains the virtualized BIOS, and other software components to boot a system on a virtualization platform, e.g. the patched bootloader and kernel. All the software components loaded inside the VM are extended to this *rCoT*. $\textit{rCoT}_{\textit{VM}}$ represents the property of a VM as a whole. Similarly, different versions are maintained by the supported *SPDs*. We also use M_0 to denote the initial value for the measurement.

$$M(rCoT_{VM}, SID) = TPM\_Extend(M_0, Trans_{SID}(sq(rCoT_{VM}))) \quad (6.20)$$

Similarly, the list of translated measurements is denoted as:

$$M(rCoT_{VM}) = \{M(rCoT_{VM}, SID_i) \mid SID_i \in \{IDs\ for\ Supported\ SPDs\}\} \quad (6.21)$$

The measurement log for each $M(rCoT)$ is maintained by the resource's provider, i.e. the CSEs (*Cloud Service Enforcers*). This log records the detailed information of the CoT, i.e. the *itf*, *stf*, and the *sq*. It serves as the *Manifest* for each resource. When deploying each resource, the CSEs record and store the *Manifest* for each expected state of the resource. They also implement querying services to return the correct *Manifest* given a measurement value.

6.3.3 Measuring Cloud Chain-of-Trust for a Node

The measurement value of a node N's *cCoT* is stored in the *Cloud Configuration Register (CCR)*. We denote the information for constructing the *cCoT* as the *Cloud Measurement Log (CML)*, which is stored in the *Cloud Measurement Log Storage (CMLS)*. By definition, constructing N's *cCoT* requires its *Collector* to obtain the followings:

- The measurement of its own *Core Chain-of-Trust (coreCoT)*, i.e. $M(rCoT_{CoreServ_N})$. This is achieved by querying the vPCR values from the vTPMs of the node's *Core Services*.
- The measurements of all its dynamic dependency nodes' *coreCoT*, i.e. $M(dynCloudCoT_N)$. These are gathered by requesting all the node's *Core Services*'s communication peers to return their $M(rCoT_{CoreServ})$.
- All its active static dependency node's *cCoT*, i.e. the $M(staCloudCoT_N)$. These measurements are obtained by querying the *CCR* and the *CMLS* of the dynamic dependency nodes, when they are marked as N's static dependency.

Collector stores all the obtained measurement values to the *Storage*, which is implemented as a *sorted set*. The *set* implementation eliminates identical measurement values so that *dCoTs* are identified. The *sorted* property produces a definite *sequence*, so that a same list of measurement values will generate the same *cCoT* measurement. We denote this sequence as $sq(Storage)$.

Every time when *sq*(*Storage*) is changed, the measurement value for N's *cCoT* is calculated, according to Equation 6.13. This is implemented by firstly *extending* the measurement value in the *sequence* to the related *Connection Strength* value C_{ij}, and then iteratively *extending* the result value to the node's $M(\textit{rCoT}_{\textit{CoreServ}_N})$. Here we use the *TPM_Extend* operation to implement the *extending*. The final result is stored in the *CCR* to represent the measurement value of N's current *cCoT*.

$$\textit{CCR} = \textit{TPM\_Extend}(M(\textit{rCoT}_{\textit{CoreServ}}), \{M_i \mid M_i \in \textit{sq}(\textit{Storage})\}) \tag{6.22}$$

CML records all the measurement values extended to the *CCR*. Moreover, for each $M(\textit{cCoT})$ value for N's active static dependency node, its *CML* is copied. Therefore, a *CML* recursively records the $M(\textit{rCoT})$ for all N's static and dynamic dependency nodes. From each $M(\textit{rCoT})$, the *Manifest* can be queried from the related *CSEs* to interpret the detailed properties for each resource.

$$\begin{aligned} \textit{CML} = \{M(\textit{rCoT}_{\textit{CoreServ}}) \quad &, \quad \textit{sq}(\textit{Storage}) \\ &, \quad \{\textit{CML}_i \mid N_i \in \textit{StaDep}(N) \wedge N_i \in \textit{DynDep}(N)\}\} \end{aligned} \tag{6.23}$$

CCR and *CML* only record the *current cCoT* of a node. This means that every time when Equation 6.22 is calculated, the original *CCR* and *CML* value is overwritten, instead of iteratively measured. Iteratively measuring the new *CCR* value with the existing value will record the history of all different *cCoT* for a node. This history is only concerned when measuring the *cCoT* for a VM.

6.3.4 Measuring Cloud Chain-of-Trust for a VM

The goal for measuring the *cCoT* of a VM is to record the *cCoT* for all the nodes that have hosted the VM during its lifecycle inside the cloud. The changed *cCoTs* of a node during its hosting period are also be recorded. According to Equation 6.14, this is implemented by iteratively *extending* the VM's host's new *cCoT* measurement to VM's current *cCoT* measurement. Typically, three critical phases are considered:

1. the VM's *cCoT* is initialized as $\textit{vPCR}_{\textit{Init}}$, which is generally defined as 0;
2. when the VM is deployed or migrated to a host, its $M(\textit{cCoT}_{\textit{VM}})$ is *extended* with the host's $M(\textit{cCoT}_{\textit{Host}})$;
3. when the host's *cCoT* is updated, the new $M(\textit{cCoT}_{\textit{Host}})$ is *extended* to $M(\textit{cCoT}_{\textit{VM}})$.

NeuronTPM manages this measurement with the vTPM manager module. vTPM manager maintains a *vCCR* for each vTPM. Whenever the *CCR* of a node is changed, the *vCCR* for every hosted VM is updated by *extending* the new *CCR* to its exist value. Accordingly, each new *CML* is appended to the previous one, so that a *vCML* is constructed, which records all the measurement logs for calculating the $M(rCoT_{VM})$.

$$vCCR = TPM\_Extend(vCCR, CCR) \tag{6.24}$$

Accordingly, *vCCR* records all the occurred *cCoT* properties for the VM during its lifecycle inside the cloud. By examining each *CCR* value from the *vCML*, the attester will determine: 1) what kinds of services the cloud has provided for supporting the VM, and 2) what kinds of potential malicious behaviours have been launched against the VM. We now discuss how *vCCR* and *vCML* are exported to VM to achieve cloud attestations.

6.3.5 vTPM Interfaces

Several previous research have has proposed the vPCR-PCR mapping schemes to bind the vTPM with the hardware Root-of-Trust. We propose the similar structure, though the mapping can be adjusted according to implementation needs. *vPCR*$[0]$ - *vPCR*$[7]$ are mapped to the hardware *PCRs* to implement *Deep Quote* [94]. In particular, they map to the *PCR*$[0]$ - *PCR*$[7]$, which record the measurement values of the $rCoT_{Neuron}$. However, as we propose to hide the implementation detail of the NeuronVisor, and more importantly, to hide the host and its TPM's identity, we preserve the hardware *PCR* mappings only for compatibility.

vPCR$[9]$ records the measurements of all the necessary software components for booting the VM, i.e. the SeaBIOS, Trust Grub, Linux Kernel, etc. *vPCR*$[10]$ is reserved for measuring all the software loaded by the Linux Kernel IMA [74]. As we require the NeuronVisor to obtain the trust evidence for the *Core Services*, which are implemented as VMs as well, we extend the vTPM implementation to allow the vTPM Manager to securely query the *vPCR*$[9]$ and *vPCR*$[10]$.

Finally, we assign *vPCR*$[11]$ to store the VM's *vCCR*. As this measurement is calculated and maintained by the underlying NeuronVisor, its value is writeable to the NeuronVisor, but read only to the VM. To return the related measurement log, we propose to extend the vTPM interface with the *vTPM_MLQuote* instruction. By initiating this instruction, the vTPM return the *vCML* to the VM's user space.

6.4 Summary

In this chapter we designed the *Cloud Chain-of-Trust* model, based on the definitions of the *Cloud Trusted Computing Base* and the *Cloud Root-of-Trust* model. *cCoT* effectively constructs a single *CoT* to link the trust evidence for all the trust dependencies of an application inside the cloud. With the *vCCR* interface, a single *TPM_Quote* will query and attest to this entire *cCoT* for a VM. We leave the NeuronTPM implementation to our future work, as understanding and modifying the vTPM manager and vTPM instance implementations are out-of-scope of this thesis. In the next chapter, we will examine how to implement cloud attestation services with a large-scale distributed cloud application based on the *vCCR* interface.

Chapter 7

Case Study

7.1 Overview

In this chapter, we propose a Trusted MapReduce (TMR) framework, which integrates MapReduce systems with the TCG Trusted Computing infrastructure. TMR demonstrates how a large-scale distributed application effectively employs remote attestations to achieve efficient and deterministic integrity verification. Based on TMR, we further discuss how these attestations are implemented in the cloud environment with the help of our *SoP* models and the NeuronVisor framework.

MapReduce [104] has been developed by Google to simplify parallel data processing on large clusters. It has been widely adopted by both industry and academic organizations for solving complicated problems involving large-scale data processing, such as high end computing [105] and large scale semantic annotation [106]. Various data intensive scientific computations [107, 108] also benefit from this model. Meanwhile, the rapid emergence of open infrastructure, such as Service-Oriented Architecture (SOA) and Cloud Computing, provides public data processing services to researchers, data analysts, and developers to process vast amounts of data efficiently and cost-effectively (e.g. the Amazon MapReduce [109] services). Difficulties in gaining access to large-scale computing and storage resource for implementing MapReduce applications have been significantly reduced. As a result, MapReduce systems deployed over open infrastructure are becoming attractive solutions for large-scale cost-effective data processing services.

However, in this model, as the infrastructure no longer belongs to customers, its *integrity* must be protected and verified. Computing nodes (machines) may be tampered with or ill-configured to return wrong results for assigned MapReduce tasks, which will in turn generate incorrect results. Considering the scale of the data, this error is very hard to identify. It may result in significant damage, especially for scientific [110] and financial computations [111]. Moreover, the *confidentiality* and *privacy* of customer data in the

public MapReduce systems are also at risk. Even if customers encrypt all their data stored inside the cloud, they still need to be sure that the MapReduce systems will not leak the data when decrypting them for processing.

Concerning integrity, recent research mainly focuses on replication-based verification schemas [112, 113]. The same tasks are deployed simultaneously on different nodes. Their outputs are compared to identify inconsistency, which indicates malfunctions in one or more nodes. However, deficiencies still exist:

1) *Large overheads.* Inevitable overheads are introduced as duplicated computations are performed. One time replication incurs 100 percent performance overheads. For protections against conspiring malicious nodes (i.e. all those tampered nodes returning the same result in order to be regarded as producing the right results) even more replications will be needed.

2) *Probability-based fault discovery.* To reduce the replication rate, probability models [112] are used to duplicate selectively part of the overall workloads, in pursuit of a certain fault-discovery rate. However, for integrity-critical applications (e.g. scientific and financial computing) deterministic accuracy on the results must generally be achieved.

3) *Incapable of faulty-node identification.* Inconsistency in outputs may indicate the existence of faulty nodes, but these nodes are not effectively identified. Faulty-nodes identification is especially difficult when wrong output from the conspiring nodes constitutes the larger portion in the total replications. In this case, all the nodes producing inconsistent results will be examined, introducing large service disruption overheads.

4) *Vulnerable to customer-based DoS attack.* Malicious customers can deliberately submit tasks, which generate random outputs. In this situation, good nodes are treated as tampered ones since they are assigned to execute the same malicious task but return different output. This attack forces good nodes to be regularly re-examined or re-initialized. It will decrease the overall throughput of the MapReduce systems. Higher fault-discovery rate targeted by the replication-based scheme requires increasing the replication rate, which will intensify the impact of this attack.

Security enhancements to the MapReduce infrastructure have also been proposed. Authentication and authorization mechanisms have been integrated into Hadoop MapReduce systems [114, 115] for preventing unauthorized access to customer data and credentials. IDSs and firewalls are also deployed inside the open infrastructure to mitigate different attacks [109]. Mandatory access control and differential privacy have also been used [116] for avoiding information leakage beyond the data provider's policy. However, these security mechanisms, including the integrity verification mechanisms discussed above, are also implemented inside the open infrastructure — that is, they are still services provided by

the infrastructure. Hence, their trustworthiness (i.e. whether they are the controls are enforced as expected) should still be verified. This recursive dependency of trust leads to the demand for a definition of Root of Trust (RoT) in this infrastructure. Moreover, a Trusted Third Party (TTP) to provide assurance for the trustworthiness of this RoT [29] should also be introduced.

As discussed, the tamper-proof nature and the implanted cryptographic protocols empower a TPM to act as a TTP dedicated to a platform. Therefore, with remote attestation, customers can make deterministic assumptions on the security properties of a MapReduce infrastructure, including the reliable enforcement of all the security protection mechanisms. More importantly, the integrity and genuineness of MapReduce tasks' results can be verified. As no replication is introduced, the deficiencies discussed above are prevented.

In this chapter, we propose a practical design and implementation of a Trusted MapReduce (TMR) infrastructure. We show that Trusted Computing will significantly improve the overall efficiency for fault discoveries. Meanwhile, by carefully integrating remote attestation with MapReduce communication protocols, the impacts on overheads and scalability issues introduced by the Trusted Computing can be well controlled. In particular, we made the following contributions:

1) ***Deterministic MapReduce tasks integrity verification scheme.*** With remote attestations to verify the trustworthiness of every computing node, *deterministic* statements of the integrity of MapReduce tasks and their results can be made. Tampered or ill-configured nodes can be *accurately identified*, and wrong results can be effectively filtered. As *no replication* on task execution is needed, the overheads are significantly reduced. The customer-based DoS attacks are also avoided. TMR enables security-critical MapReduce applications to be implemented in publicly-available open infrastructure. It hence encourages a huge amounts of large-scale distributed computations.

2) ***Practical and efficient MapReduce infrastructure attestation***. In TMR, we integrate the Timestamped Hashchain-based attestation schema [33], and propose a parallel attestation implementation. We also propose to split the attestation session, further reducing the overheads by eliminating time-consuming but unnecessary computations. Moreover, as the attestations are only performed inside the infrastructure, the white-lists are easy to manage and privacy is protected. Our scheme explores a practical way to implement trusted computing in large-scale distributed applications. It can easily be adapted to implement a wide range of trusted applications on open infrastructure, e.g. trusted PaaS.

3) ***Trusted Hadoop MapReduce implementation.*** We implemented TMR by extending the Hadoop MapReduce system [114]. We also presented the deployment and modifications of the underlying supporting Trusted Computing infrastructure. The widely used

Hadoop MapReduce enables TMR to be quickly adapted for industry-strength applications. Our experiments and analysis showed that with only negligible overheads, our Trusted Hadoop MapReduce implementation achieves a strong integrity assurance.

4) ***Trusted Cloud Integration.*** We showed that TMR implementation can be migrated to the cloud infrastructure with only minor adaptations. With the *Cloud Root-of-Trust* and *Cloud Chain-of-Trust* abstractions implemented by the NeuronVisor, the *Cloud Trusted Computing Base* of TMR can be effectively identified and attested to. This allows the customers to further determine the trustworthiness of the entire MapReduce infrastructure.

The next section presents the general framework of the Trusted MapReduce infrastructure. Section 7.3 describes its implementation, and presents the performance evaluations and security analysis. In Section 7.4, we further discuss the NeuronVisor integration, which support the TMR in the trustworthy cloud environment. Related research on securing the MapReduce systems is presented in Section 7.5.

7.2 Trusted MapReduce Framework

7.2.1 MapReduce Model

MapReduce [104] is a framework for processing large amount of data on certain kinds of distributable problems using a large number of computers (nodes). Computational processing can occur on data stored either in a file-system (unstructured) or within a database (structured). Two fundamental computing steps exist in the MapReduce computation model. In the "Map" step, the *Master* (i.e. the management node) takes the input, partitions it up into smaller sub-problems, and distributes those to the *Workers* (i.e. the computing nodes). A Worker may do this again in turn, leading to a multi-level tree structure. The Worker processes that smaller problem, and passes the answer back to the Master. In the "Reduce" step, the Master then takes the answers to all the sub-problems and combines them in some way to get the output the answer to the problem it was originally trying to solve.

During the Map phase, the Master assigns Map tasks to Workers. A Worker then reads a data block from the DFS (Distributed File System), processes it and writes its intermediate result to its local storage. The intermediate results generated by each Worker is divided into r partitions $P1, P2, \ldots, Pr$ using a partitioning function. The number of partitions is the same with the number of reduce tasks r. During the Reduce phase, the Master assigns Reduce tasks to Workers. Each Reduce task specifies which partition the Worker should process. After a Worker receives a Reduce task, it waits for notifications of Map task

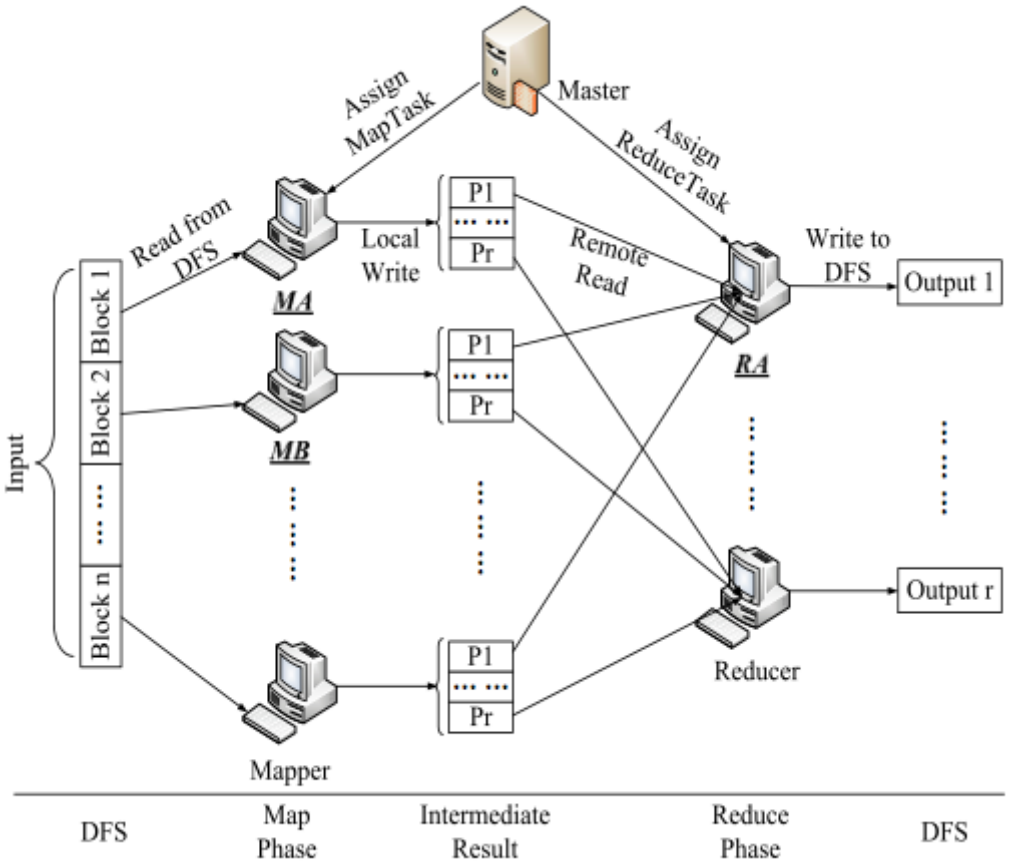


Figure 7.1: MapReduce Computing Model

completion events from the Master. Upon notified, the Worker reads its partition from the intermediate result of each Worker who finishes its Map task. For example, in Figure 7.1 (from [114]), *RA* (the Worker running a Reduce tasks) reads $P1$ from *MA*, *MB* (the Workers running Map tasks) and other Workers. After the Worker reads all the partitions it needs, it starts to process them, and finally outputs its result to the DFS.

7.2.2 Threat Model

In TMR, we consider the threats from malfunctioning Workers who attack directly the MapReduce tasks. We further examine the indirect threats aiming at attacking the underlying infrastructure. Moreover, attacks against our trusted enhancements are also reviewed, including circumventing or abusing the TMR mechanisms.

Malfunctioning Workers. In a public MapReduce infrastructure, malfunctioning Workers may violate customers' security requirements. They may be tampered with to return forged outputs, which will violate the *integrity* of customers' results. They may also be

modified to leak customers' *confidential* data or profile customers' behaviours or preferences for *privacy* mining. On the other hand, software flaws and bugs or inappropriate policy configurations can also lead to incorrect results or unintended information leakage.

Infrastructure attacks. Attacks to the open infrastructure also exist. Tampered nodes may *eavesdrop* the traffic among other Workers and the Master, in order to leak confidential data, enforce malicious privacy profiling or perform *replay* or *Man-In-the-Middle* attacks to the MapReduce protocols [112]. They may also *impersonate* the Master to steal other Workers' data, or vice versa. Moreover, malicious Workers can launch DoS attacks against other good Workers. For example, they may keep sending requests to a good Worker and asking for intermediate results or they may impersonate the master to send fake task assignments to Workers.

Trusted Computing attacks. Various attacks exist for circumventing or abusing the Trusted Computing mechanisms. Malicious Workers can perform *replay attacks* to reuse past attestation tickets, representing trustworthy states, to fake its behaviour. A *Man-In-the-Middle attack* enables malicious Workers to forward the attestation request from the Master to a good Worker, and redirect the returned ticket to the Master to also pretend to be in the states of the good one. Moreover, *runtime attacks* can be performed to inject malicious codes directly into memory and without being measured, e.g. stack-overflow attacks. Finally, remote attestations can be maliciously performed to retrieve the detailed configurations of other Worker for system profiling.

Assumptions. In TMR, we do not consider hardware attacks. We hence regard the TPM as tamper-proof, i.e. its instructions cannot be tampered with and its protected storage cannot be exposed. Moreover, as the TPM's design goal is to resist all software-based attacks, we assume that all software loaded on a trusted platform will be genuinely measured to the TPM. We also assume the deterministic behaviours of a software component, i.e. in an unchanged environment, the same binary code will produce the same output for the same input.

7.2.3 General Framework

The general Trusted MapReduce (TMR) framework is shown in Figure 7.2. The *Task Scheduler* in every Worker manages and executes tasks deployed on them. *Trust Collector* is added to the Worker to generate TCG trust evidence (i.e. the attestation tickets) and implement the remote attestation service. In the Master, the *Job Scheduler* deploys jobs to Workers and collects their results. Worker information and job information for scheduling are stored and managed by the *Worker Manager* and the *Job Manager* respectively. In addition, TMR adds the *Trust Manager* to manage the trust information of Workers, and the

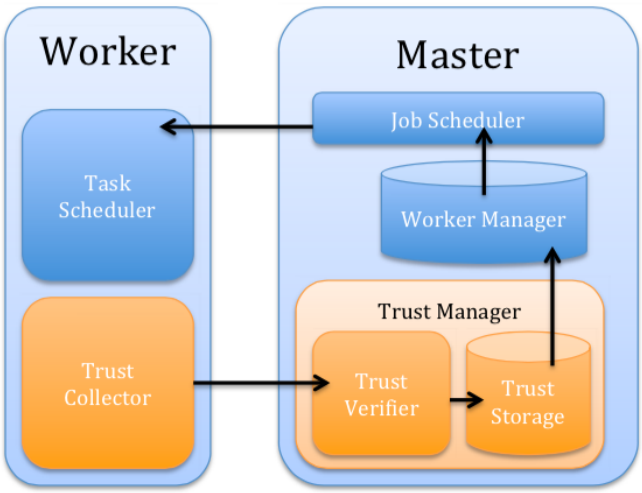


Figure 7.2: Trusted MapReduce Framework

Trust Verifier, which connects to the *Trust Collectors* on Workers and performs attestations to them. The gathered security properties are stored in the *Trust Storage*. Two procedures are introduced:

i) ***Initial trust establishment.*** Each time a Worker sends a connecting request to the Master, an initial attestation will be performed from the Master to the Worker. The *Trust Verifier* verifies the properties of the Worker and stores them inside the *Trust Storage*. Only expected Workers with expected properties will be added to the schedule list in the *Worker Manager* for executing MapReduce tasks. Initial credentials and necessary shared session keys are also set up.

ii) ***Periodical trust update.*** As the properties of a platform is liable to change after being attested to, attestations to it should be performed regularly. The security properties of each Worker are evaluated periodically. Any inconsistency with predefined security policies will result in the Worker being removed from the schedule list immediately. The tasks, assigned to the Worker after its last trustworthy state, will be re-scheduled.

7.2.4 Trust Management

As depicted in Figure 7.3(a), the major latency introduced by a remote attestation session is composed of the ticket generation (mainly the *TPM_Quote* instruction), and the *Storage Measurement Log (SML)* generation and verification [117]. As we will introduce shortly, TMR uses the *active attestation* [33] to generate tickets, in parallel with the attestation sessions (Figure 7.3(b)). Hence, for each attestation, only negligible overhead for fetching and sending the latest ticket, together with verifying it, is incurred (Figure 7.3(c)). Active

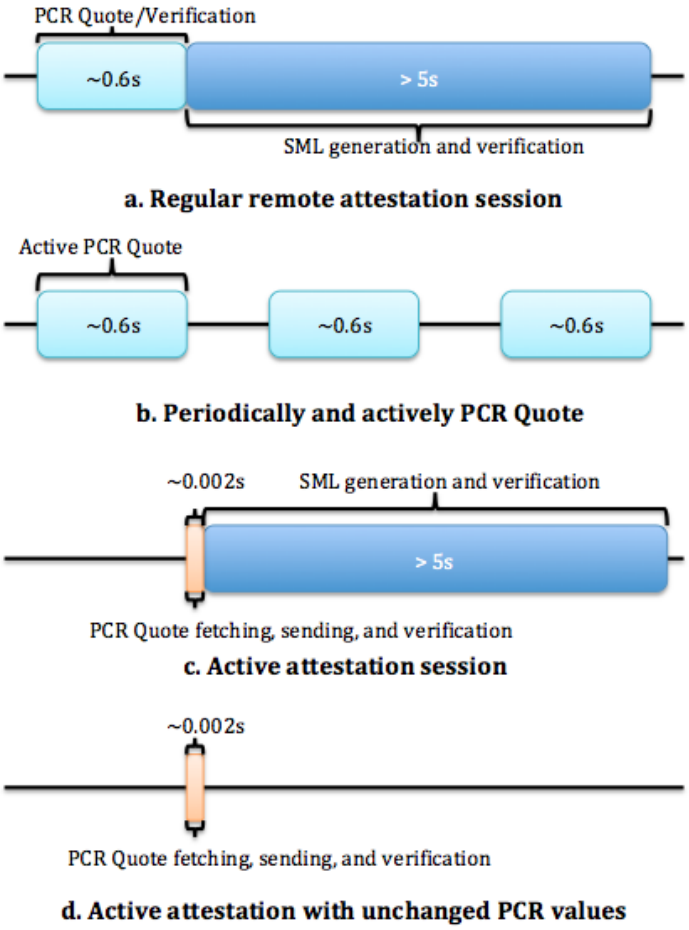


Figure 7.3: Attestation response time constitution

attestation adds some latency for fault detections and may result in more computations to be revoked, as the tickets are previously generated. However, as we will show in our evaluations, this latency can be controlled to a very low level. Moreover, considering the significantly improved overall throughput, this overhead is negligible.

On the other hand, in a stable production infrastructure, the platforms' states seldom change [118]. For example, the Workers' configurations only changes when they load new privileged executables, e.g. loading new security modules, applying patches or launching malicious attacks. The attestations to a Worker in TMR are hence further optimized by first comparing the *PCR* values with the previous ones. The time-consuming *SML* generation and verification procedures are only initiated when the *PCR* values change. Therefore, the overheads in this part is further reduced for only comparing two hash values (Figure 7.3(d)), which is also negligible. The following sections present the related protocols, and we will discuss the implementation and detailed performance overheads in Section 7.3.

7.2.4.1 Trust Establishment

In TMR, every Worker is identified with its AIK (Attestation Identity Key), and the Master serves as the Privacy-CA [31, 100] for certifying and managing all these AIKs. When a new Worker is added to the MapReduce infrastructure, it is first registered with the Master and assigned with an AIK. As an AIK cannot be forged and can only be used inside a specific genuine TPM, in TMR, only expected Workers can connect to the Master. The registration protocol initiated by a Worker (W_i) to the Master (M) is listed in Protocol 1. Every time when a Worker requests to connect to the Master, e.g. after the registration or a fresh reboot, the trust establishment protocol depicted in Protocol 2 is invoked. This protocol is adapted from the attestation protocols in [33].

Protocol 1 Worker Registration.

---

a) *Registration request by W_i.* W_i creates an AIK key pair by invoking the TPM_CreateAIK instruction of its TPM. The private key is never exposed to outside of the TPM, and its public part is signed by the Endorsement Key (EK) of the TPM [31], which is also protected by the TPM. W_i then initiates a registration request and sends its EK certificates [1] and the AIK public key to M.

$$W_i \rightarrow M : Cert(K_{EK_i}), \{K_{AIK_i}\}_{K_{EK_i}^{-1}} \tag{1.1}$$

b) *Assign AIK_i by M.* M verifies $Cert(EK_i)$ and generates the AIK certificate for W_i. A unique ID is also assigned to W_i.

$$M \rightarrow W_i : \{Cert(K_{AIK_i}), WID_i\}_{K_M^{-1}}, Cert(K_M) \tag{1.2}$$

7.2.4.2 Trust Gathering

With the limited computation capability of a TPM chip, a complete attestation procedure between two nodes may require several seconds [117], the latency of which is unacceptable especially when attestations are performed regularly. Stumpf et al. [33] proposed the Timestamped Hashchain-based Attestation (or the *active attestation*) to compensate this deficiency. To avoid performing expensive TPM operations for every attestation request, the server (the node to be attested to) deduces a nonce from a Trusted Third Party (TTP) certified timestamp-nonce binding. This nonce is then used for performing the *TPM_Quote* instruction, and generating attestation tickets. As the nonce is no longer a shared secret between the server and a particular challenger, the ticket can be reused by a third party. Nevertheless, as it is bound to a global time, the freshness of the ticket can still be deduced.

Protocol 2 Trusted Establishment.

a) *Shared key pre-computation by W_i.* W_i selects an appropriate prime p and generator g of $Z_p^*(2 \leq g \leq p-2)$. W_i chooses a random secret $s, (2 \leq s \leq p-2)$, and computes $g^s \mod p$. W_i transmits p and g to M.

b) *Shared key pre-computation by M.* M chooses a random secret $c_m, 2 \leq c_m \leq p-2$, and computes $g^{c_m} \mod p$.

c) *Attestation challenge by M.* M initiates an attestation session to W_i.

$$M \rightarrow W_i : \{g^{c_m} \mod p\}_{K_M^{-1}} \tag{2.1}$$

A nonce is then generated and sent to W_i.

$$M \rightarrow W_i : N_a \tag{2.2}$$

d) *Report attestation ticket by W_i.* W_i sends its PCR values back to M, together with the nonce and the pre-computation. They are signed by W_i's AIK.

$$W_i \rightarrow M : \{N_a, \{PCR\}, g^s \mod p\}_{K_{AIK_i}^{-1}} \tag{2.3}$$

e) *Key confirmation.* M computes the shared session key by computing $K_{SC} = (g^s)^{c_m} \mod p$. M then generates a second non-predictable nonce (N_b) and transfers message to W_i.

$$M \rightarrow W_i : \{N_b, g^{c_m} \mod p\}_{K_{SC}} \tag{2.4}$$

f) W_i computes the shared session key by computing $K_{SC} = (g^{c_m})^s \mod p$ and decrypts the received message with K_{SC}. W_i then transfers the SML to M.

$$W_i \rightarrow M : \{N_a, N_b, SML, g^s \mod p\}_{K_{SC}} \tag{2.5}$$

g) *Verify W_i by M.* M verifies the AIK_i signature, examine consistency of PCR and SML, and then determine the security properties of W_i from SML. The $\langle PCR, Properties \rangle$ binding is updated to the *Properties Storage*, and the $\langle WID_i, PCR \rangle$) binding is updated to the *Worker Manager*, which can later be used for making job schedule decision and implement provenance binding. We will describe the provenance management in detail in the next section.

In TMR, we adapt this attestation schema to implement efficient periodical attestations. The Master acts as the TTP for issuing the initial timestamp-nonce binding. After every time interval v, a new nonce is generated by hashing the previous one. The *Trust Collector* implements the active attestation and act as both the server and the challenger. It continuously deduces the new nonce and generates attestation tickets to itself. It sends the newest ticket to the Master, who then determine the most recent state of the Worker and validate the integrity of the results. As we will discuss in detail later, the ticket generation and reporting are implemented in parallel, hence only negligible latency is incurred. On the discovery of any validation failure, the Master re-schedule the tasks finished after the last success attestation time, which can easily be deduced from the nonce used for the ticket.

To implement active attestation, the time of W_i and M should firstly be synchronized. This can be done with the initialization protocol (Protocol (2.2)). Here we omit the detailed synchronization protocol, which can be achieved by simply using the NTP (Network Time Protocol). The nonce certificate is also generated accordingly:

Initial attestation request and nonce certificate by M. M generates and signs with its public key K_M^{-1} the nonce certificate for the binding of the initial nonce with the time-stamp of the current time. The increment time interval v is also bound. The initial nonce is then used for the trust establishment attestation, and used as the seed for generating new nonces further on. Protocol (2.2) is modified to include the initial nonce deployment procedure:

$$M \rightarrow W_i : \{N_0, t_0, v\}_{K_M^{-1}}$$

In every pre-defined interval, an active attestation is performed by a Worker. The nonce is updated, and a new attestation ticket is generated. The trust update protocol is depicted as Protocol 3. The nonce update and verification algorithms in [33] are adapted.

7.3 Hadoop MapReduce Implementation

7.3.1 Implementation

In this section, we first present our modification to the most prevalent MapReduce system, the Hadoop MapReduce [114], to implement our TMR framework. We then describe the implementation of the underlying trusted computing infrastructure for supporting TMR.

Protocol 3 Trust Update.

a) *Updating attestation nonce by W_i.* After every time interval, W_i generates a new attestation ticket. It first generates the nonce by hashing the last nonce.

$$N_k = h(N_{k-1})\,(k \geq 1) \tag{3.1}$$

b) *Reporting attestation ticket by W_i.* W_i sends its PCR values and N_k back to M.

$$W_i \rightarrow M : \{N_k, k, \{PCR\}\}_{K^{-1}_{AIK_i}} \tag{3.2}$$

c) *Verifying attestation ticket by M.* On receiving the attestation ticket, M first validates the AIK_i signature, and then check the freshness of the nonce by examining whether it is generated within an acceptable time period:

$$t_0 + v \times k < t_{now} < t_0 + v \times (k+1) \tag{3.3}$$

And whether the nonce is valid:

$$N_k = h^k(N_0) \tag{3.4}$$

d) *Update W_i state by M.* M determines the state change of W_i by comparing the PCR_k to PCR_{k-1}. If they are different, it fetches the SML_k from M, validates its consistency with PCR_k and update the security properties of W_i. For applications with more strict security requirements, Protocol 2 can be initiated.

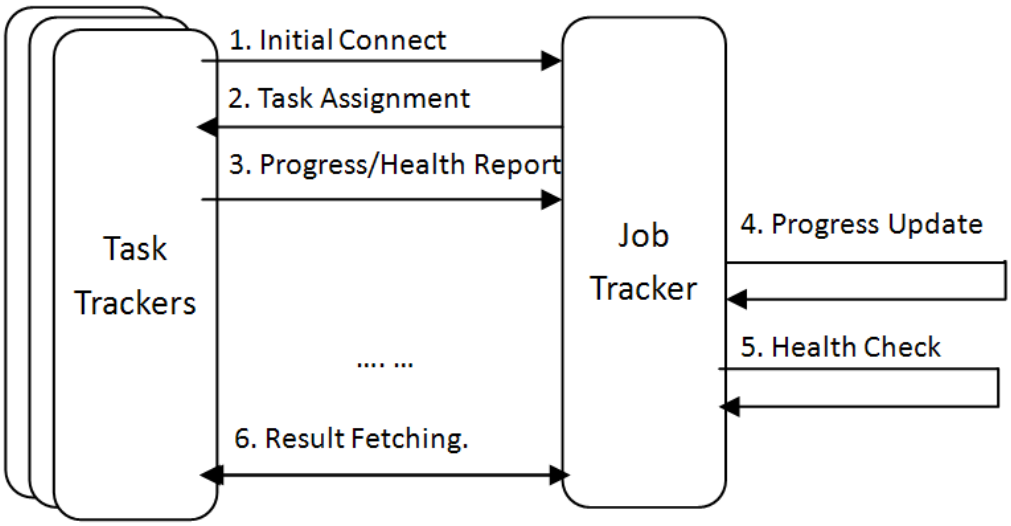


Figure 7.4: The heartbeat protocol in Hadoop MapReduce

7.3.1.1 Hadoop MapReduce Model

The JobTracker and TaskTracker are the Hadoop counterparts of Master and Worker respectively. JobTracker schedules Map and Reduce tasks and manages TaskTrackers. A *job* (the Hadoop terminology for referring to a customer's task) is split by the JobTracker into *tasks* (Hadoop's schedule entities) to operate on a subset of the customer data. A TaskTracker executes the assigned tasks, and reports the progress periodically. As depicted in Figure 7.4, the *heartbeat protocol* enforces the communications between the JobTracker and TaskTrackers, including:

i) *TaskTracker Initialization.* When a TaskTracker first connects to the JobTracker, an *initialConnect* bit is set in the *heartbeat request* (i.e. the heartbeat package sent by the TaskTracker to the JobTracker). The JobTracker then updates the TaskTracker's information in its database for task scheduling. No security check is performed at this stage.

ii) *Task assignment and results collection.* TaskTracker monitors its own load. It sets the *acceptNewTasks* bit in the heartbeat request to ask for new tasks from the JobTracker when it has capacities to execute more. The executable and the range of data to operate is sent back from the JobTracker as the *heartbeat response*. After finishing a task, TaskTracker also sends the storage address of the results to the JobTracker.

iii) *Task executing progress reporting.* In each heartbeat request, the progress of the executing tasks is sent to the JobTracker. JobTracker updates this progress data, and makes scheduling decisions correspondingly.

iv) *Health status reporting.* Scripts can be run on each TaskTracker for performing administrator-defined health checking. Each heartbeat request quotes the health report

generated by the scripts and sends it in the *healthStatus* field. JobTracker then invokes the *healthCheck* interface for examining this report, and decides whether to keep on using this TaskTracker or add it to a blacklist and stop communicating with it.

7.3.1.2 Trust Manager

In our Trusted Hadoop MapReduce implementation, the *Attester* and the *Verifier* are added to TaskTracker and JobTracker respectively. Their communications are integrated into the heartbeat protocol. We simplify our prototype by using a manually issued AIK certificate and nonce certificate for each TaskTracker. Attester implements the Trust Collector interface in our TMR model. It runs as a thread inside the TaskTracker process, and is initialized when the TaskTracker starts. It first initializes the TPM on the TaskTracker, and then reads and loads the locally stored private AIK into the TPM. The nonce certificate specifies the initial nonce value, the time bound to the nonce and the nonce update interval. It is also loaded by the Attester at the initialization time. Afterwards, the Attester updates the nonce, and generates the attestation ticket repeatedly by invoking the *TPM_Quote* instruction with the latest nonce. The attestation timestamp and the nonce counts are also recorded within the ticket. The latest generated ticket is added to the *healthStatus* field of the heartbeat request and sent to the JobTracker.

The Verifier implements the Trusted Verifier interface. It maintains the public AIKs of all TaskTrackers. When a heartbeat request with the *initialConnect* bit is received, it searches and loads the AIK public key into memory. Verifier then invokes the *openpts* command (describe in the next section) to attest to the TaskTracker. It verifies the return PCR values with the AIK, checks the compliance of the PCR values and the SML entries, and then deduces the security properties from the SML. The properties are examined with predefined security policies, and only the trusted TaskTracker can be added for future task scheduling.

Every time a heartbeat request with an attestation ticket arrives at the JobTracker, the Verifier is invoked from the *healthCheck* interface in the heartbeat processing procedure. As shown in Algorithm 1, Verifier maintains a nonce cache (the *nonce_cache*), recording a number of the most recent nonce (*cache_size*). It updates the cache by iteratively performing a number of times the *SHA-1 hash* function to the newest nonce. Each new nonce is pushed in to the cache, and with the oldest removed. The number of times for performing this function is determined by the number of time intervals (δ) passed from the time bound to the latest nonce (*time_last*) to the current time (*time_curr*). The length of time interval and the size of the cache together determine the oldest time for a ticket to be valid.

Algorithm 1 Verifying Attestation Tickets

$nonce\_new \leftarrow nonce\_last$
$nonce\_updates\_times \leftarrow ((time\_curr - time\_last)/\delta)$
for $i = 1 \rightarrow nonce\_updates\_times$ **do**
    $nonce\_new \leftarrow hash(nonce\_new)$
    add $nonce\_new$ into $nonce\_cache$
    while $nonce\_cache.size \geq cache\_size$ **do**
        remove oldest $nonce$ from $nonce\_cache$
    end while
end for
if $ticket.nonce \notin nonce\_cache$ **then return** false (Ticket Obsolete)
end if
if VERIFY_SIGNATURE$(nonce, AIK) = false$ **then return** false (Ticket Invalid)
end if
if $hash(PCRs) \notin tt\_pcrhash\_map$ **then**
    attest to TaskTracker with OpenPTS
    if Attestation Succeeded **then**
        add IR to $pcrhash\_IR\_map$
        add $\langle time\_curr, pcrhash \rangle$ to $tt\_pcrhash\_map$ **return** true (Success)
    elsereturn false (TaskTracker Failed)
    end if
else
    add $\langle time\_ticket, pcrhash \rangle$ to $tt\_pcrhash\_map$ **return** true (Success)
end if

Verifier searches the nonce used by the ticket (*ticket.nonce*) in the cache. A search miss indicates that the ticket is obsolete, and will cause the discard of this heartbeat request. Only when the TaskTracker sends back another heartbeat with a valid ticket can it continue to communicate with the JobTracker. Verifier then validates the quote signature with both the nonce and the public AIK of the TaskTracker by the *VERIFY_SIGNATURE* instruction of TPM. Failure in the verification will cause the return of *false* from the *healthCheck* procedure, which will result in the reinitialization of the TaskTracker and rescheduling of the tasks deployed to the TaskTracker under the current Hadoop MapReduce implementation.

After a successful verification, Verifier compares the PCR values with the former ones of the TaskTracker. The mappings of TaskTrackers to the PCR values (and the timestamp of their generation) are maintained in a hashtable (*tt_pcrhash_map*). Any change in the PCR values indicates changes in the TaskTracker's states, and a new attestation session will be performed to it by invoking the openpts. Failure in attestation will also result in the healthCheck returning false. On success of the attestation, the new TaskTracker configuration, i.e. the Integrity Report (*IR*, the openpts's terminology for SML), will be added to the *pcrhash_IR_map*. The *tt_pcrhash_map* will also be updated. If the sates are not changed, only the timestamp deduced from the nonce of the ticket (*time_ticket*) will be updated to the *pcrhash* entry in the *tt_pcrhach_map*.

7.3.2 Evaluation

Our prototype is implemented by modifying Hadoop MapReduce 0.21. We deployed 3 TaskTrackers and a JobTracker on 4 HP Compaq nc6320 laptops respectively. Each machine is equipped with 1 GB memory and a dual-core Intel Centrino Duo CPU running at 1.67GHz. An Infenion TPM v1.2 is attached to each of the machines. Ubuntu 11.04 is deployed as the base system, running Linux Kernel version 2.6.38-8-generic with IMA supports turned on. The supporting trusted computing facilities are also deployed. Java HotSpot(TM) Client VM v1.6.0_26 is used as the JVM for TMR. We deliberately choose these low capacity machines to amplify the performance impacts of the trusted enhancement, and we will show that TMR still achieves satisfying results.

7.3.2.1 Macro benchmark

Our evaluations are based on the comparison of the time differences for the Hadoop MapReduce 0.21 (HMR) and our TMR prototype to implement the WordCount [119] on 3.5 giga-

Table 7.1: TMR Configurations

No.	Heartbeat	Quote	State-change
1	3	3	-
2	3	3	300-350
3	0.1	0	300-350

bytes of text data. Three important parameters are considered in our experiments:

a) *Heartbeat Interval.* The attestation tickets are sent in every heartbeat request to reflect the newest states of the TaskTracker. The verifications also took place during the processing of the heartbeat request. Hence the smaller the heartbeat interval is, the more tickets are sent and verified, hence the more latency is imposed. On the other hand, heartbeats update the progress of the TaskTrackers with the JobTracker, hence a smaller heartbeat reduces the time a TaskTracker waiting for the assignment of new tasks, increasing overall throughput.

b) *Quote Interval.* This interval determines the frequency the attestation tickets are generated. A larger interval indicates a longer time the TaskTracker's states are updated. This results in a longer intermediate state, giving rise to the risk of more tasks to be rescheduled. However, lower attestation frequency no doubt incurs less overheads.

c) *State-change Interval.* When a state-change of a TaskTracker is discovered, a new attestation will be performed to it. As this involves the regeneration and revalidation of the SML, much more overheads will be incurred. Though the states of TaskTrackers are seldom changed in an industry-strength system [118], we deliberately increase this change frequency to test its impacts. In our experiments, in every state-change interval, we load a new script, which will result in the change of the loaded-software-list maintained by Kernel-IMA of the TaskTracker and hence changes the value of $PCR10$.

Table 7.1 illustrates our three different sets of evaluation configurations. First two configurations are set with the default Hadoop MapReduce *Heartbeat Interval*: 3 seconds. The *Quote Intervals* are hence set to be compliant with the heartbeat, in order to report one new attestation ticket in every heartbeat request. In test case #2 (TMR-Att), we deliberately changed every TaskTracker's state in an interval no less than 300 seconds: this interval is refreshed each time with a random number between 300 and 350 seconds.

With each configuration, the WordCount runs 15 times, and the average of the time (seconds) to finish each round is presented, together with the standard deviations. As shown in Table 7.2, TMR only incurs 1.24% overheads which is far less than existing proposals [112, 113]. For TMR-Att, even under our simulated extensively unstable environment (as

Table 7.2: Response time under default Heartbeat Interval

	Avg Resp. Time	StDv	Overheads
HMR	2107	34	-
TMR	2134	40	1.24%
TMR-Att	2175	29	3.10%

Table 7.3: Response time under tight Heartbeat Interval

	Avg Resp. Time	StDv	Overheads
HMR-Tight	1862	23	-
TMR-Tight-Att	1956	45	4.79%

oppose to the average state change intervals of an stable industry-strength platform), the incurred overheads is still acceptable.

For the test case #3 (Table 7.1), the *Heartbeat Interval* for both HMR and TMR are set to 0.1 second, for simulating an intensively tight scheduling environment. They are referred to as HMR-Tight and TMR-Tight respectively. The *Quote Interval* for TMR-Tight is set to 0 to force the active attestations to perform with no interval, i.e. new tickets will be generated immediately one after another. As shown in Table 7.3, TMR-Tight-Att still exhibits good results.

As we tested TMR under the constrained environment, in real systems, the overheads percentage can be further reduced. The overheads for an attestation is fixed for a TaskTracker when its platform configurations (i.e. the list of loaded software components) are not changed. Hence by increasing the throughput of the TaskTracker, e.g. by using more CPUs (cores), which is a common case in industry-strength systems, the overheads percentage reduces proportionally and can easily be controlled to below 1%.

7.3.2.2 Micro benchmark

We record the time (seconds) for performing critical operations introduced by TMR as the micro benchmark. Each result is the average of 15 times of execution. The standard deviations are also presented. As shown in Table 7.3, a full attestation needs around 5 seconds, which includes quoting and verifying the PCR values, and generating and analysing the Integrity Reports (IRs). Significant delays will be imposed if the attestation is performed with every heartbeat, i.e. 5 seconds' delay for every 3 seconds for each TaskTracker. The JobTracker is also easily overwhelmed when the scale of the MapReduce system grows.

Table 7.4: Trusted computing operations overheads

	Avg Time	StDv
Full Attestation	5	0.3
PCR Quote	0.659	0.006
Verify Quote	0.0016	0.0008
Trusted Boot	191	4
Regular Boot	52	1

However, in TMR, PCR values are examined separately, and the full attestations are performed much less frequently. Moreover, as the active attestation schema is used, the heartbeat delay is further reduced. For the TaskTracker, each heartbeat request simply sends the already generated the last ticket. Moreover, for the JobTracker, the *Quote Verification* overheads is also negligible (0.0016 seconds). As TMR only introduces overheads to the heartbeat protocol, which is independent of different type of MapReduce applications, the overheads is also general to the applications. From the micro benchmark we can further deduce TMR overheads in a large-scale system, which we argue can still be managed to be in a low level.

The parallel execution of active attestation also helps increasing the respond time: though the *PCR Quote* still needs 0.659 seconds, as it is mostly carried out by the TPM, the CPU time for this operation is still negligible. Hence only little impact is imposed on the Workers for executing MapReduce tasks.

On the other hand, TMR incurs an inevitable bootstrapping delay. During its bootstrapping, all software components should be measured and extended to the TPM. As in Table 7.3, this process is 3 times longer than a general system booting. However, in real systems, nodes (TaskTrackers) are not rebooted frequently. Hence, as opposed to the average up-time of a commercial MapReduce server, the bootstrapping delay is still negligible.

7.3.2.3 Security Analysis

Tampered workers. In TMR, any program loaded on a Worker will be reliably measured, with the measurement value irreversibly stored inside the tamper-proof TPM. The malicious behaviours can hence be undeniably detected. Collusion attacks are also avoided by default. Faults are also faithfully recorded and can later be traced. For example, the hash value of our modified Hadoop MapReduce implementation, the *hadoop-TMR-all.jar*, is recorded in the measurement log and extended to *PCR[10]*. Any change to the binary of the program will result in a change in its hash value, and, in turn, a change to the value

of *PCR[10]*. The Master can then check this log and determine whether the right program has been loaded. Manipulating the measurement log will result in inconsistency when it is validating against *PCR[10]*, and will also be detected. All malicious programs will be measured and examined in the same way.

Infrastructure attacks. In TMR, a Worker's identity is represented by its AIK, which cannot be abused for *impersonation*, even if the malicious Worker successfully obtains the AIK files of a good Worker. The Master itself acts as the Privacy-CA [31, 100] for certifying the AIKs for Workers. It possesses all the information necessary for validating AIKs. Hence it also cannot be cheated with a forged AIK.

The public key of the Master is deployed to every Worker. It can be used to verify the signature of the instructions the Master sends to the Workers. The shared session key established during the initial attestations can also be used to encrypt the communication traffic between the Master and a Worker. The *DoS attack* and *eavesdropping* are hence avoided. Moreover, TPM's features can further be used to provide stronger security properties for these protocols, e.g. trusted channel [78, 120] and non-migratable keys.

Trusted Computing attacks. A DRTM (Dynamic Root of Trust for Measurement) [31, 91] can be used to establish a runtime chain-of-trust, to protect critical components on the system against *runtime attacks*. Every attestation ticket in TMR is bound to a nonce, from which the time of the ticket generation can be deduced. Hence, past tickets can be effectively detected and discarded, avoiding *replay attacks*. A shared communication secret between a Worker and the Master is also used to encrypt the attestation session, avoiding the *Man-In-the-Middle attacks*.

In TMR, attestations are performed within the MapReduce infrastructure: from the Master to the Workers. The detailed configurations are translated into security properties to satisfy customers' provenance queries. Hence they are not directly exposed to customers. Moreover, the SML transmission traffics are encrypted by the shared session keys, and the initial protocol for setting up these keys is signed by the Master. Hence, in TMR, only the Master can fetch the SMLs of Workers. Malicious Workers are avoided from performing attestations for *malicious configurations profiling*. The trustworthiness of the Master can be easily verified by a Trusted Third Party, as with most trusted cloud proposals [58, 61]. We will examine this verification with NeuronVisor in the next section.

The security properties of a piece of software or service may change over time. For example, after the discovery of a vulnerability, a program's original property of *data encryption* is changed. However, if the trusted provenance is bound to platform properties instead of detailed platform configurations, this change cannot be traced. In this case, we propose the property bindings to be temporary. Users can update this binding with the

Master and re-evaluate the security properties of the enforcement environments of their MapReduce tasks.

7.4 NeuronVisor Integration

Hadoop MapReduce systems deployed as open infrastructures allow customers to access large-scale computing and storage services with low costs. However, customers may still desire to only expose their privately data to their private owned MapReduce system, while avoiding the costs of purchasing and maintaining the large-scale physical servers. This can be achieved by deploying the MapReduce system as an IaaS cloud application. Typically, customers acquire VMs from the cloud, and deploy Master and Worker services respectively. The virtualization technologies enforce strong isolation, so that the MapReduce system is under the customers' full control.

As we discussed with our trusted cloud expectations, remote attestations in cloud should consider the cloud's complexity, heterogeneity, and dynamism. NeuronVisor framework facilitates these attestations by implementing the *Cloud Root-of-Trust* and *Cloud Chain-of-Trust* abstractions, which allow customers to regard the cloud as a single logical resource. Based on these abstractions, two trusted cloud services can be implemented:

- *Trusted Communication.* This service allows applications to integrate attestations with their business logic. Critical operations among application VMs can be preceded by immediate attestations to ensure that the communication targets have the expected properties. In this case, attestations are enforced to VMs, instead of to the entire application. For example, when initiating critical traffic to a target VM, the initiator VM can first examine the target VM's *cTCB*.

- *Cloud Attestation.* Customers attest to their cloud applications and the *cTCB* by interrogating the *cRoT*, which is maintained and testified by the *Trust Evidence Reporters (TERs)*. According to the *Multiple Instances* restriction, customers are free to employ different TERs to gather trust evidence. They can contact TERs periodically to attest to their applications. TERs can also provide services to enforce periodical attestations and inform the customers for any SLA violations. This helps them to narrow the *fault-discovery-window*, which is defined as the time gap between the actual misbehaviours' occurrence and particular counteractions' enforcement.

In this section, we discuss how these two services help to implement TMR in the cloud environment.

7.4.1 Trusted Communication

When TMR is deployed as a cloud application, the Master attests to the Worker VMs periodically to identify security property violations. However, two major problems arise. The first is that the Worker VM's entire *cTCB* should be attested to. This is because, in a cloud environment, the entire hosting infrastructure determines the genuine behaviours of a VM. Misbehaviours in any supporting part will alter the target VM's behaviour. For example, in the OpenStack cloud system, the permanent storage of a VM is managed by the service hosted on the *Storage* node, while the VM's networking capabilities may be managed by the *Neutron* service hosted on another node. Attesting only to a Worker VM and its host cannot identify the faults in one Worker's storage and networking capabilities, etc.

The second problem is that the integrity of the Master VM's *cTCB* is also crucial for verifying the properties of a Worker. This is because, inside the cloud, the attester VM's behaviours still depend on its hosting infrastructure. Misbehaviours in this infrastructure will affect the attester's genuine judgements. For example, the attestation results are subject to change when the Master VM's host is tampered with. Moreover, even when faults in the Worker VM are identified, the counter-measurements enforced by the Master VM may also be tampered with. As a tampered Master VM cannot effectively verify its own trustworthiness, we delegate this verification to a third party: the TERs or the customers themselves. This is implemented by the *Cloud Attestation* presented in the next section. We now discuss how the Master VM attests to the *cTCB* of the Worker VMs.

In NeuronCloud, the *cTCB* of a VM is attested to by examining its *cCoT*. This examination is achieved by examining the *vCCR* of the target VM. Quoting the *vCCR* returns the measurement value of the entire *cCoT* of the VM's hosting nodes. This value verifies the trustworthiness of the corresponding *vCML*, which records how the *cCoT* is constructed. By examining this log, the attester will obtain the measurement values for: 1) $rCoT_{Core}$ of the nodes that have hosted the target VM; 2) $rCoT_{Core}$ of all the static dependency nodes, which implement the services to support the VM's host's functionalities; and 3) $rCoT_{Core}$ of all the nodes that have interacted with the host and its static dependency. These nodes may provide extra services that the host or its static dependency need. They may also possess security risks, which facilitate malicious behaviours.

Integrating trusted communication with TMR requires modifications to its attestation procedures. On the Worker side, the OpenPTS client should *quote* the *vPCRs* recording the measurement values for the VM's *cCoT*. This includes: 1) $vPCR[9, 10]$, representing all software components loaded inside the VM; and 2) $vPCR[11]$, recording the *vCCR*. Moreover, the OpenPTS client should be modified to fetch the corresponding *vCML*, in addition

to the measurement log maintained by the IMA [74]. However, the Worker implementation is not changed, as it simply sends the *vPCR* values and measurement logs produced by the OpenPTS client to the Master.

On the Master side, when receiving the attestation ticket through the *Heartbeat* protocol, it firstly examines the *Connection Strength* value of the target Worker's host. This represents the integrity strength of the Worker's RoT, i.e. the NeuronVisor layer. As only the integer value is presented, this underlying cloud infrastructure detail is hidden from the Master. Moreover, the Master avoids obtaining and examining the signature of the Worker's underlying TPM for trusting the vTPM's attestation ticket. This further hides the infrastructure detail, and reduces the management complexity.

For the values in $vPCR[9, 10]$, the active attestation scheme is still effective, as these values seldom change for a stable running system. Therefore, Algorithm 1 is employed for their verification. Verifying these values, the Master will determine the integrity of the Worker VM.

$vPCR[11]$ is verified by iteratively extending the measurement values listed in *vCML* according to Equation 6.14, and comparing the final result with its value. However, this measurement may change frequently, as it represents the properties of the Worker VM's entire *cCoT*. Therefore, we envisage non-negligible computing overhead, as for every new *vCCR* value, its *vCML* will be examined. Meanwhile, we argue that this overheads is still controllable when the caching mechanism is properly designed and implemented. This is because each entry in the *vCML* represents the aggregated measurement values of a node in the cloud, which seldom changes based our assumption. Moreover, in one general case, most collaborating VMs will be deployed in one same compartment in the cloud for reducing communication overheads. Therefore, most part of the Worker VMs' *vCMLs* is identical, as they share the same part of the supporting infrastructure.

To further improve performance, the *Master* can skip verifying the *cCoT* of each Worker VM, but examining only each Worker VM's *rCoT* instead, i.e. $vPCR[9, 10]$. As the TMR application is attested to regularly by customers, the *cCoT* of the entire application is still examined frequently. Cloud attestation is presented next.

7.4.2 Cloud Attestation

The integrity of the *Master* needs further examination. This is because, in the cloud environment, misbehaviours in the *Master*'s hosting infrastructure may alter the *Master*'s capabilities in perceiving or counter-acting the misbehaviours in the *Workers*. With NeuronVisor, this examination is achieved by the Cloud Attestation service. With support from the

Separation-of-Powers model, customers are able to make informative decisions on whether the CSEs are providing the declared services for hosting their MapReduce applications.

To attest to the TMR from outside of the cloud boundary, customers contact the attestation Proxy of the TER they have acquired. They specify the metadata of the TMR application with the Proxy. This usually includes the identities of all the VMs in the applications, e.g. the Master and Worker VMs. When the property-based attestation is employed, the *SID* is also specified to identify the SPD the customers have employed. With this information, the Proxy instructs the Delegate, which locates each specified VM inside the cloud and quotes its *vCCR* and *vCML*. Obtaining the *vCCRs* and *vCMLs*, the Delegate first examines whether *vCMLs* conforms to the *vCCRs* by iteratively computing the hash value for the measurement values in the *vCML* and comparing the final value the *vCCR*. When all the *vCMLs* are verified, they are *merged* to represent the $cCoT_{App}$ according to Equation 6.15. This final *vCML* is returned to the customer as the *Digest* generated by the TER.

The verification procedure is similar to the $cCoT_{VM}$ attestation. After obtaining the *vCML*, customers further require the *CSEs* to provide the *Manifests*, which contain the list of hash values which are used for calculating the measurement for each *vCML* entry. This list represents the identity of all the loaded software components composing each chain. As the property-based attestation is used, each entry on the list is the *translated* value of the measurement value of the binary code of the target components. With these values, customers query their SPDs for the property certificates. To support *Multiple Instances*, customers can employ different TERs. This decision can be helped with the reputation models or the risk models.

7.5 Related Work

SecureMR [112] proposes to reduce the replication rate based on probability models. Secure protocols are used for encrypting the task information, and signing the outputs. PKI facilities are also introduced accordingly. It achieves a fault-detection rate of 90% while reducing the replication rate to 40%. Its overheads under this setting with 60 map tasks ranges from 5% to 12%. VIAF (Verification-based Integrity Assurance Framework) [113] is based on the idea of replication-based and quiz-based methods, and it can detect both colluding and non-colluding mappers. VIAF adds a limited number of trusted computation nodes called verifier to verify a small portion of consistent results in a random manner and thereby detect collusive mappers. However, as the quiz threshold changes, verification overhead can raise to 23.58%, which is still very high. Moreover, as with the nature of a

replication-based solution, critical deficiencies besides performance overheads still exist as we discussed in Section 7.1.

Airavat [116] provides strong security and privacy guarantees for distributed computations on sensitive data and integrates mandatory access control and differential privacy. It balances the competing goals of a permissive programming model and the need to prevent information leaks. IDS and firewall are also deployed inside the open infrastructure, with secure internal communications enforced [109].

In TMR, as remote attestations are performed, deterministic judgements on Workers' integrity can be made. Meanwhile, as we discussed in Section 7.1, the integrity of the security services should still be attested to. With TMR, these components can be measured as the security properties of Workers. As shown in our experiments, the overheads can be easily managed to below 1%.

7.6 Summary

In this chapter, we presented a Trusted MapReduce (TMR) framework. With remote attestations, a Master in TMR can determine the exact properties of its Workers, and also make deterministic statements on the integrity of the result generated by them. As no task replication is needed, overheads are reduced significantly. We implemented the TMR on the Hadoop MapReduce system. Our experiments showed that the introduced overheads can easily be managed to within only 1%. TMR demonstrates that with the capabilities of identifying the genuine configurations of a remote platform, Trusted Computing (TC) can help effectively establish and maintain trust in a large and complex system, without involving complicated algorithms and protocols. By carefully organizing the TC components, the TC deficiencies can be well tackled. TMR sets an exemplar for implementing large-scale trusted systems.

We further presented how to migrate TMR to a cloud environment. Two critical problems are concerned: 1) effectively examining the trustworthiness of the hardware RoT for a target VM; and 2) verifying the properties of the entire *cTCB* for the target. NeuronVisor mitigates these problems by implementing the *Cloud Root-of-Trust* and the *Cloud Chain-of-Trust* abstractions. As discussed previously, *cRoT* effectively hides the underlying TPMs, while enabling the Master to trust the attestation tickets generated by the Worker VM. Moreover, the *cCoT* discovers and organizes all the necessary trust evidence for attesting the cloud services that have supported the target VM. The services have the potential to tamper with the target VM will also be identified. Based on these two abstractions,

the TMR's attestation protocols can be well supported, when it is migrated to the cloud environment.

Chapter 8

Conclusion and Future Directions

Verifying the trustworthiness of a cloud infrastructure is an important and hard problem. We observe that a critical cause for the hesitation of the wide-scale adoption of the cloud computing paradigm is the lack of a well-defined trust in the cloud. Customers need to know whether their purchased cloud services have been genuinely executed. They may also concern whether unauthorized behaviors are enforced on their private data or applications inside the cloud. Consequently, without the design and implementation of effective and practical mechanisms to verify the genuine behaviors of a cloud infrastructure, the risks for employing the cloud computing paradigm arise.

The currently widely applied cloud auditing schemes are only able to examine and certify the trustworthiness of a cloud infrastructure at a very coarse grained level . They cannot help customers to determine the satisfaction of their specific Service Level Agreements (SLAs). They also cannot adapt to the fast development of a cloud infrastructure, as they are usually enforced in a one-off pattern, with a relatively long interval. On the other hand, the cloud's characteristics of complexity, heterogeneity, and dynamism hamper the effective enforcement of the TCG's remote attestation technology. Existing cloud attestation schemes are unable to determine a fine-grained Trusted Computing Base (TCB) for a cloud application, and apply targeted attestations.

My DPhil study has been dedicated to discover a new direction for solving this problem. In Chapter 3, we defined the *Separation-of-Powers (SoP)* models, which restricts the authority of the currently overpowered Cloud Service Providers. The key to implementing *SoP* is a cloud infrastructure that allows third-parties to effectively inspecting and defining its internal behaviors. From Chapter 4 to Chapter 6, we defined three important concepts respectively for implementing this cloud infrastructure, namely the *Cloud Trusted Computing Base (cTCB)*, the *Cloud Root-of-Trust (cRoT)* and the *Cloud Chain-of-Trust (cCoT)*. In Chapter 7, we further demonstrated how this cloud infrastructure fulfills the trusted ser-

vice requirements from a large-scale distributed cloud application, the Trusted MapReduce system.

Separating the Powers. We argue that the root cause for the trust issues in the cloud comes come from the overpowered *Cloud Service Providers (CSPs)*. Currently, the CSPs are endowed with the powers for *defining*, *executing*, and *inspecting* the cloud's activities. When these powers are concentrated in one authority, customers lose capabilities for verifying the genuine behaviors of the cloud. Accordingly, we designed the *Separation-of-Powers* model by referencing the political philosophy concepts.

We defined three roles to carry the responsibilities for defining, executing, and inspecting the cloud computing services respectively. With the *Collaboration Model*, the behaviors of a *Cloud Service Enforcer (CSE)*, the reduced-power CSP, are *inspected* by multiple third-party *Trust Evidence Reporters (TERs)*. By employing the Trusted Computing technologies, *TERs* are able to record the genuine behaviors of the cloud infrastructure. When these behaviors are interpreted by the *Software Property Definers (SPDs)*, customers will be able to determine the satisfaction of their SLAs, or identify unauthorized access.

As we designed that the *TERs* can only obtain the *meshed* behavior digests, they are not able to discern the detailed behaviors represented by the evidence they have gained, without the malicious collaborations from the other roles. This increases the difficulties for malicious TERs to secretly tampering with the behavior evidence. It also limits malicious or unintended information leakage. Meanwhile, as the *SPDs* are unaware of the detailed trust evidence, they are unable to secretly falsify the property definitions, without malicious collaborations, to tamper with the evidence interpretations.

To prevent malicious collaborations or unintended faults, we further defined the *Restriction Models* to allow each role to restrict the behaviors of the others, so that the *Balance-of-Power* is achieved. The *Mutual-Inspection* model enforces restrictions between *TERs* and *SPDs*. The *Multiple-Parties* model allows *TERs* to inspect the other *TERs*, and *SPDs* to define the other *SPDs*' behaviors.

With the *Collaborations* and *Restrictions*, we hope to achieve a trustworthy and open cloud ecosystem by endowing customers with the *Freedom-of-Choices*, and allowing service providers to equally participate in the ecosystem, regardless of their scales and established credibility. In this ecosystem, as the genuine behaviors of each parties can be verified, they are all able to gain equal trust. As the powers are separated to multiple roles and parties, customers are able to choose the *CSE-TER-SPD* combination for providing cloud services, based on what they are *willing* to trust, instead of what they are*forced* to trust. We wish the *SoP* model to open an opportunity for achieving a more flourish cloud computing ecosystem.

Defining the Trusted Computing Base in the Cloud. Regarding implementing an *SoP*-compatible cloud infrastructure, my research focuses on the *TER* side: how to effectively obtain the trust evidence for testifying the cloud services' genuine behaviors. The first problem is to identify the relevant supporting cloud services, and potential malicious entities, for a target cloud application: i.e. defining the application's *Cloud Trusted Computing Base (cTCB)*.

Based on our observations that both the supporting dependency and malicious behaviors require direct communications, we defined that the genuine behaviors of a node depend on the behaviors of its direct communication peers. Accordingly, we proposed to manage the trust evidence inside the cloud based on the cloud nodes' communication patterns. We designed the *Decentralized Attestation* scheme, which distributes the attestation efforts from a centralized attestation delegate to every cloud computing nodes. As each node has the capabilities of collecting the trust evidence of its communicating peers, the *cTCB* of each node is easy to determine and attest to. By aggregating the *cTCB* of each node hosting the target application, the *cTCB* of the application is defined.

Moreover, the distributed enforced attestations increase the overall throughput and scalability, as no centralized controlling service is needed. Single-point-of-failure is also avoided. However, one issue of concern is the increased redundant attestation efforts, as distributed peers lack the shared information on how the others are attested to. Therefore, repeated attestations to a node will be performed frequently.

This is solved by integrating the reputation systems from the Peer-to-Peer community. We designed the RepCloud system, which employs the trust aggregation and dissemination mechanisms to manage the trust evidence gathered by each node. As shown from the simulations, this greatly reduces the redundant attestation efforts, as attestation tickets are effectively reused. More importantly, the reputation models further help to determine the hidden trust dynamics inside the cloud, which will be modelled for defining the *cRoT* and *cCoT* abstractions.

RepCloud further opens an opportunity of using the reputation systems for managing the trust relationship defined by the Trusted Computing Infrastructure (TCI). The *deterministic* property of the trust evidence generated by TCI facilitates more efficient trust aggregation and dissemination. Meanwhile, these trust management schemes reduce redundant attestation efforts, and direct the attestation focuses to the critical targets, according to nodes' interaction dynamics.

Rooting Trust into the Cloud. The major problem for attesting to a cloud application's *cTCB* comes from the lack of definitions for the *Cloud Root-of-Trust (cRoT)* abstraction.

The *cTCB* abstraction creates a logical cloud presentation, which only contains the necessary cloud components for supporting or tampering with the target cloud application. A *cRoT* should also be defined accordingly for the customers to securely obtain the trust evidence vouching for the *cTCB*'s genuine behaviors, without concerning the underlying details of the physical hardware Roots-of-Trust, i.e. the TPMs. Revealing each TPM's details to customers breaks the cloud's promises for hiding the physical resources' identities. It also enlarges the attack surfaces and increases management overheads.

We designed the *NeuronVisor* framework, which adapts the *Decentralized Attestations* only to manage a simple property for each node: whether it can genuinely attest to the behaviors of its upper-layer services. Accordingly, the formed Neuron Web will attest to the properties of all the hosted services. As the Neuron Web is formed based on the communication patterns, it covers the *cTCB* for a target application. A Neuron Web thus serves as the *cRoT* for attesting to the related *cTCB*. By attesting to any Neuron on the web, and examining its *Connection Strength* to the other Neurons, the trustworthiness of this *cRoT* will be verified. Based on this verified *cRoT*, the attesters can securely fetch the trust evidence maintained by each Neuron to attest to the represented *cTCB*. As a Neuron Web only reveals a single TPM's information for attesting to the initial Neuron, the infrastructure detail is hidden.

Chaining Trust from the Root. Concerning organizing the evidence for achieving effective cloud attestation based on the *cRoT*, we design the *Cloud Chain-of-Trust (cCoT)* abstraction. Our goal is to implement the *cCoT* as a single-root linear sequence, with redundant trust evidence reduced, and individual service identity hidden.

To achieve this goal, we first defined the concepts of *Domain Chain-of-Trust (dCoT)* and *Collaboration Domain Chain-of-Trust (cdCoT)*. *dCoT* and *cdCoT* combine multiple separated *CoTs* to a single one by *merging* the identical ones, and *connecting* different ones based on the *Connection Strength* defined by the *cRoT*. Based on these two definitions, the *cCoT* is defined, according to the multi-level *cTCB* definitions.

To construct this *cCoT*, we further design the NeuronTPM framework. NeuronTPM records the measurement values for all the cloud nodes that have supported or affected a VM during its lifecycle inside the cloud. It constructs a single measurement value from all these measurements to represent the entire *cCoT* for the VM. Accordingly, with the help of the *cRoT* and *cCoT* abstractions, the attester will determine the properties for the entire *cTCB* of a VM by verifying one single *vPCR* value and examining the related measurement log.

Integrating Trust Services. We built a *Trusted MapReduce (TMR)* framework to demonstrate how Trusted Computing helps effectively establish and maintain trust in a

large and complex distributed system. Our experiment showed that, by carefully integrating remote attestations with the application workflows, the complexity and overheads introduced by the Trusted Computing Infrastructure will be greatly reduced.

We further presented how to implement TMR in a cloud environment, with support from the *trusted communication* and *cloud attestation* services implemented by the NeuronVisor and NeuronTPM frameworks. As we aimed to design the *cRoT* and *cCoT* abstractions with the similar presentations of the TCG's *RoT* and *CoT*, attestations to a cloud application can be implemented as attesting to a single-platform application: by verifying a single TPM's signature, the *cRoT* is examined; by examining a single vPCR value, the *cCoT*'s properties are determined. This property greatly reduces the implementation or adaptation efforts for trusted cloud applications.

We believe that the ease of implementation will further facilitate cloud customers to secure their cloud applications by integrating the trusted cloud services, which will in turn ease the application of the *SoP* model and encourage a wider range of customers to employ the cloud computing paradigm.

Future Directions. We admit that limitations still exist, and we envisage a future research direction for each topic discussed throughout the dissertation.

- For the SoP models, *collaboration* and *restriction* models only define the basic relationship among the different roles. A wider range of *extension* models (Section 3.4) can be designed to further enrich the SoP model. These models may help the cloud customers to make more informated decisions when choosing among different service providing parties.

- Identifying the *cTCB* based on the direct communication is still not accurate. We argue that precise *cTCB* identification requires a deeper understanding to the communication semantics, which will only be achieved with the collaboration from the cloud services. We propose that more interfaces be implemented by the NeuronVisor to allow cloud services to actively and genuinely register their dependency inside the cloud.

- A wider range of reputation systems from the P2P community can be studied and utilized in RepCloud or NeuronVisor for better modelling the trust dynamic inside the cloud. Various reputation attack defensing systems may also be studied. Moreover, we find it particularly interesting to apply the Big Data technologies on analyzing the trust relationship gathered by the RepCloud or NeuronVisor framework. This may help to identify hidden threats inside the cloud, or determine security critical nodes for more targeted attestations.

- Property-based attestation mechanisms have been proposed. However, effective implementations are still missing. Integrating these systems with the NeuronVisor and NeuronTPM framework will complete the implementation of a full-supported *SoP*-compatible cloud infrastructure.

- The TMR system further leads to another direction of trustworthy big-data processing. Trusted provenance systems can be designed to bind the data processing infrastructure's properties with the data processing lineage, so that customers will able to understand how each piece of their data is processed in detail.

References

[1] "2015 state of the cloud report from rightscale." http://www.rightscale.com/lp/2015-state-of-the-cloud-report.

[2] "The state of the cloud 2015 bessemer venture partners." www.bvp.com/blog/state-cloud-2015.

[3] "Cloudcamp: Five key concerns raised about cloud computing." http://www.cloudcamp.org/.

[4] ENISA, "Cloud computing - SME survery." http://www.enisa.europa.eu/act/rm/files/deliverables/cloud-computing-sme-survey/, 2009.

[5] E. Keller, J. Szefer, J. Rexford, and R. B. Lee, "NoHype: virtualized cloud infrastructure without the virtualization," *SIGARCH Comput. Archit. News*, vol. 38, pp. 350–361, June 2010.

[6] Ponem Institute, "Insecurity of privileged users: Global survey of IT practitioners." http://h30507.www3.hp.com/hpblogs/attachments/hpblogs/666/62/1/HP%20Privileged%20User%20Study%20FINAL%20December%202011.pdf., 2011.

[7] M. Keeney, "Insider threat study: Computer system sabotage in critical infrastructure sectors." http://www.secretservice.gov/ntac/its_report_050516.pdf, 2005.

[8] E. Kowalski, "Insider threat study: Illicit cyber activity in the information technology and telecommunications sector." http://www.secretservice.gov/ntac/final_it_sector_2008_0109.pdf, 2008.

[9] "Cloud security market." http://www.transparencymarketresearch.com/pressrelease/cloud-security-market.html.

[10] MarketsandMarkets, "Cloud security market worth." http://www.marketsandmarkets.com/PressReleases/cloud-security.asp.

[11] J. M. A. Calero, N. Edwards, J. Kirschnick, L. Wilcock, and M. Wray, "Toward a multi-tenancy authorization system for cloud services," *IEEE Security and Privacy*, vol. 8, pp. 48–55, November 2010.

[12] R. Chow, M. Jakobsson, R. Masuoka, J. Molina, Y. Niu, E. Shi, and Z. Song, "Authentication in the clouds: a framework and its application to mobile users," in *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, CCSW '10, (New York, NY, USA), pp. 1–6, ACM, 2010.

[13] S. Bleikertz, M. Schunter, C. W. Probst, D. Pendarakis, and K. Eriksson, "Security audits of multi-tier virtual infrastructures in public infrastructure clouds," in *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, CCSW '10, (New York, NY, USA), pp. 101–112, ACM, 2010.

[14] G. Wang, Q. Liu, and J. Wu, "Hierarchical attribute-based encryption for fine-grained access control in cloud storage services," in *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, (New York, NY, USA), pp. 735–737, ACM, 2010.

[15] F. Zhang, J. Chen, H. Chen, and B. Zang, "CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 203–216, ACM, 2011.

[16] S. Butt, H. A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy, "Self-service cloud computing," in *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, (New York, NY, USA), pp. 253–264, ACM, 2012.

[17] A. Srivastava and V. Ganapathy, "Towards a richer model of cloud app markets," in *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*, CCSW '12, (New York, NY, USA), pp. 25–30, ACM, 2012.

[18] "Amazon web services: Overview of security processes." http://d36cz9buwru1tt.cloudfront.net/pdf/AWS_Security_Whitepaper.pdf, 2011.

[19] M. Christodorescu, R. Sailer, D. L. Schales, D. Sgandurra, and D. Zamboni, "Cloud security is not (just) virtualization security: a short paper," in *Proceedings of the 2009 ACM workshop on Cloud computing security*, CCSW '09, (New York, NY, USA), pp. 97–102, ACM, 2009.

[20] "Cloud security alliance." http://www.cloudsecurityalliance.org.

[21] "Kernel-based Virtual Machine." http://www.linux-kvm.org/page/Main_Page.

[22] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, (New York, NY, USA), pp. 164–177, ACM, 2003.

[23] "OpenStack model." http://vmartinezdelacruz.com/in-a-nutshell-how-openstack-works/.

[24] F. Azmandian, M. Moffie, M. Alshawabkeh, J. Dy, J. Aslam, and D. Kaeli, "Virtual machine monitor-based lightweight intrusion detection," *SIGOPS Oper. Syst. Rev.*, vol. 45, pp. 38–53, July 2011.

[25] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold-boot attacks on encryption keys," *Commun. ACM*, vol. 52, pp. 91–98, May 2009.

[26] P. Jonathan, S. Matthias, H. Els, Van, and W. Michael, "Property attestation – scalable and privacy-friendly security assessment of peer computers," in *Technical Report RZ 3548*, IBM Research, 2004.

[27] A.-R. Sadeghi and C. Stüble, "Property-based attestation for computing platforms: caring about properties, not mechanisms," in *Proceedings of the 2004 workshop on New security paradigms*, NSPW '04, (New York, NY, USA), pp. 67–77, ACM, 2004.

[28] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: a virtual machine-based platform for trusted computing," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 193–206, Oct. 2003.

[29] J. M. McCune, "Turtles all the way down: research challenges in user-based attestation," in *Proceedings of the 2nd workshop on Recent advances on intrusion-tolerant systems*, WRAITS '08, (New York, NY, USA), pp. 2:1–2:1, ACM, 2008.

[30] "Getting started with XenServer and OpenStack." https://wiki.openstack.org/wiki/XenServer.

[31] "Trusted Computing Group." http://www.trustedcomputinggroup.org.

[32] "Trusted platform module main specification." www.trustedcomputinggroup.org/resources/tpm_main_specification.

[33] F. Stumpf, A. Fuchs, S. Katzenbeisser, and C. Eckert, "Improving the scalability of platform attestation," in *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, STC '08, (New York, NY, USA), pp. 1–10, ACM, 2008.

[34] V. Haldar, D. Chandra, and M. Franz, "Semantic remote attestation: a virtual machine directed approach to trusted computing," in *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3*, (Berkeley, CA, USA), pp. 3–3, USENIX Association, 2004.

[35] E. Shi, A. Perrig, and L. V. Doorn, "BIND: A fine-grained attestation service for secure distributed systems," in *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, (Washington, DC, USA), pp. 154–168, IEEE Computer Society, 2005.

[36] T. Jaeger, R. Sailer, and U. Shankar, "PRIMA: policy-reduced integrity measurement architecture," in *Proceedings of the eleventh ACM symposium on Access control models and technologies*, SACMAT '06, (New York, NY, USA), pp. 19–28, ACM, 2006.

[37] L. Gu, X. Ding, R. H. Deng, B. Xie, and H. Mei, "Remote attestation on program execution," in *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, STC '08, (New York, NY, USA), pp. 11–20, ACM, 2008.

[38] X.-Y. Li, C.-X. Shen, and X.-D. Zuo, "An efficient attestation for trustworthiness of computing platform," in *Proceedings of the 2006 International Conference on Intelligent Information Hiding and Multimedia*, IIH-MSP '06, (Washington, DC, USA), pp. 625–630, IEEE Computer Society, 2006.

[39] M. Alam, X. Zhang, M. Nauman, T. Ali, and J.-P. Seifert, "Model-based behavioral attestation," in *Proceedings of the 13th ACM symposium on Access control models and technologies*, SACMAT '08, (New York, NY, USA), pp. 175–184, ACM, 2008.

[40] J. Park and R. Sandhu, "Towards usage control models: beyond traditional access control," in *Proceedings of the seventh ACM symposium on Access control models and technologies*, SACMAT '02, (New York, NY, USA), pp. 57–64, ACM, 2002.

[41] C. Gebhardt and C. Dalton, "LaLa: a late launch application," in *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, STC '09, (New York, NY, USA), pp. 1–8, ACM, 2009.

[42] B. Kauer, "OSLO: improving the security of trusted computing," in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, (Berkeley, CA, USA), pp. 16:1–16:9, USENIX Association, 2007.

[43] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: an execution infrastructure for tcb minimization," *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 315–328, April 2008.

[44] K. Hoffman, D. Zage, and C. Nita-Rotaru, "A survey of attack and defense techniques for reputation systems," *ACM Comput. Surv.*, vol. 42, pp. 1:1–1:31, December 2009.

[45] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina, "The eigentrust algorithm for reputation management in p2p networks," in *Proceedings of the 12th international conference on World Wide Web*, WWW '03, (New York, NY, USA), pp. 640–651, ACM, 2003.

[46] M. Srivatsa, L. Xiong, and L. Liu, "TrustGuard: countering vulnerabilities in reputation management for decentralized overlay networks," in *Proceedings of the 14th international conference on World Wide Web*, WWW '05, (New York, NY, USA), pp. 422–431, ACM, 2005.

[47] A. Nandi, T.-W. J. Ngan, A. Singh, P. Druschel, and D. S. Wallach, "Scrivener: providing incentives in cooperative content distribution systems," in *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, Middleware '05, (New York, NY, USA), pp. 270–291, Springer-Verlag New York, Inc., 2005.

[48] R. Aringhieri, E. Damiani, S. D. C. Di Vimercati, S. Paraboschi, and P. Samarati, "Fuzzy techniques for trust and reputation management in anonymous peer-to-peer systems: Special topic section on soft approaches to information retrieval and information access on the web," *J. Am. Soc. Inf. Sci. Technol.*, vol. 57, pp. 528–537, February 2006.

[49] K. Walsh and E. G. Sirer, "Experience with an object reputation system for peer-to-peer filesharing," in *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 2006.

[50] J. R. Douceur, "The sybil attack," in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, (London, UK, UK), pp. 251–260, Springer-Verlag, 2002.

[51] K. Lai, M. Feldman, I. Stoica, and J. Chuang, "Incentives for cooperation in peer-to-peer networks," 2003.

[52] E. J. Friedman* and P. Resnick, "The social cost of cheap pseudonyms," *Journal of Economics and Management Strategy*, vol. 10, no. 2, pp. 173–199, 2001.

[53] S. Dahan and M. Sato, "Survey of six myths and oversights about distributed hash tables' security," in *Proceedings of the 27th International Conference on Distributed Computing Systems Workshops*, (Washington, DC, USA), pp. 26–, IEEE Computer Society, 2007.

[54] J. Abawajy, "Determining service trustworthiness in intercloud computing environments," in *Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium on*, pp. 784–788, Dec 2009.

[55] S. M. Habib, S. Ries, and M. Muhlhauser, "Cloud computing landscape and research challenges regarding trust and reputation," in *Proceedings of the 2010 Symposia and Workshops on Ubiquitous, Autonomic and Trusted Computing*, UIC-ATC '10, (Washington, DC, USA), pp. 410–415, IEEE Computer Society, 2010.

[56] B. Hay, K. Nance, and M. Bishop, "Storm clouds rising: Security challenges for IaaS cloud computing," in *Proceedings of the 2011 44th Hawaii International Conference on System Sciences*, HICSS '11, (Washington, DC, USA), pp. 1–7, IEEE Computer Society, 2011.

[57] K. M. Khan and Q. Malluhi, "Establishing trust in cloud computing," *IT Professional*, vol. 12, pp. 20–27, Sept. 2010.

[58] J. Schiffman, T. Moyer, H. Vijayakumar, T. Jaeger, and P. McDaniel, "Seeding clouds with trust anchors," in *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, CCSW '10, (New York, NY, USA), pp. 43–46, ACM, 2010.

[59] S. Berger, R. Cáceres, D. Pendarakis, R. Sailer, E. Valdez, R. Perez, W. Schildhauer, and D. Srinivasan, "TVDc: managing security in the trusted virtual datacenter," *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 40–47, January 2008.

[60] N. Santos, K. P. Gummadi, and R. Rodrigues, "Towards trusted cloud computing," in *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, (Berkeley, CA, USA), USENIX Association, 2009.

[61] F. J. Krautheim, "Private virtual infrastructure for cloud computing," in *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, (Berkeley, CA, USA), USENIX Association, 2009.

[62] J. McCune, T. Jaeger, S. Berger, R. Caceres, and R. Sailer, "Shamon: A system for distributed mandatory access control," in *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, pp. 23–32, Dec 2006.

[63] B. Parno, J. M. McCune, and A. Perrig, "Bootstrapping trust in commodity computers," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, (Washington, DC, USA), pp. 414–429, IEEE Computer Society, 2010.

[64] "Greg papadopoulos's comments." http://news.cnet.com/2008-1011_3-6141598.html.

[65] "Cloudaudit." http://cloudaudit.org/CloudAudit/Home.html.

[66] "Iso27002." http://www.27000.org/iso-27002.htm.

[67] "Cobit." https://cobitonline.isaca.org/.

[68] "Hippa." http://www.hhs.gov/ocr/privacy/.

[69] "Trusted compute pool." https://wiki.openstack.org/wiki/TrustedComputingPools.

[70] J. Schiffman, Y. Sun, H. Vijayakumar, and T. Jaeger, "Cloud verifier: Verifiable auditing service for IaaS clouds," in *Services (SERVICES), 2013 IEEE Ninth World Congress on*, pp. 239–246, June 2013.

[71] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu, "Policy-sealed data: a new abstraction for building trusted cloud services," in *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, (Berkeley, CA, USA), USENIX Association, 2012.

[72] I. Abbadi and A. Ruan, "Towards trustworthy resource scheduling in clouds," *Information Forensics and Security, IEEE Transactions on*, vol. 8, pp. 973–984, June 2013.

[73] S. Berger, K. Goldman, D. Pendarakis, D. Safford, E. Valdez, and M. Zohar, "Scalable attestation: A step toward secure and trusted clouds," *Cloud Computing, IEEE*, vol. 2, pp. 10–18, Sept 2015.

[74] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, (Berkeley, CA, USA), pp. 16–16, USENIX Association, 2004.

[75] W. Arbaugh, D. Farber, and J. Smith, "A secure and reliable bootstrap architecture," in *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pp. 65–71, May 1997.

[76] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, (New York, NY, USA), pp. 199–212, ACM, 2009.

[77] A.-R. Sadeghi, C. Stüble, and M. Winandy, "Property-based TPM virtualization," in *Proceedings of the 11th international conference on Information Security*, ISC '08, (Berlin, Heidelberg), pp. 1–16, Springer-Verlag, 2008.

[78] Y. Gasmi, A.-R. Sadeghi, P. Stewin, M. Unger, and N. Asokan, "Beyond secure channels," in *Proceedings of the 2007 ACM workshop on Scalable trusted computing*, STC '07, (New York, NY, USA), pp. 30–40, ACM, 2007.

[79] J. Lyle and A. Martin, "Trusted computing and provenance: better together," in *Proceedings of the 2nd conference on Theory and practice of provenance*, TAPP'10, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 2010.

[80] A. Ruan and A. Martin, "Repcloud: achieving fine-grained cloud TCB attestation with reputation systems," in *Proceedings of the sixth ACM workshop on Scalable trusted computing*, STC '11, (New York, NY, USA), pp. 3–14, ACM, 2011.

[81] A. Ruan and A. Martin, "Neuronvisor: Defining a fine-grained cloud root-of-trust," in *Proceedings of the sixth International Conference on Trustworthy Systems*, InTrust '14, 2014.

[82] "Trusted computer system evaluation criteria." http://en.wikipedia.org/wiki/Trusted_Computer_System_Evaluation_Criteria.

[83] "Rainbow series." http://en.wikipedia.org/wiki/Rainbow_Series.

[84] "Common criteria for information technology security evaluation." http://en.wikipedia.org/wiki/Common_Criteria.

[85] L. Gu, Y. Guo, A. Ruan, Q. Shen, and H. Mei, "SCOBA: Source code based attestation on custom software," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, (New York, NY, USA), pp. 337–346, ACM, 2010.

[86] J. Cucurull and S. Gausch, "Virtual tpm for a secure cloud: fallacy or reality?," in *Proceedings of the 13th Spanish Meeting on Cryptology and Information Security.*, pp. 197–202, RECSI Press, 2014.

[87] M. Strasser and H. Stamer, "A software-based trusted platform module emulator," in *Proceedings of the 1st International Conference on Trusted Computing and Trust in Information Technologies: Trusted Computing - Challenges and Applications*, Trust '08, (Berlin, Heidelberg), pp. 33–47, Springer-Verlag, 2008.

[88] I. M. Abbadi, "Clouds trust anchors," in *Proceedings of the 2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, TRUSTCOM '12, (Washington, DC, USA), pp. 127–136, IEEE Computer Society, 2012.

[89] "Amazon cloud architecture." http://jineshvaria.s3.amazonaws.com/public/cloudarchitectures-varia.pdf, 2008.

[90] "OpenStack." http://www.openstack.org.

[91] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient TCB reduction and attestation," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, (Washington, DC, USA), pp. 143–158, IEEE Computer Society, 2010.

[92] "TrouSerS - the open-source TCG software stack." http://trousers.sourceforge.net/.

[93] "Open platform trusted service user's guide." http://iij.dl.sourceforge.jp/openpts/51879/userguide-0.2.4.pdf, 2011.

[94] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn, "vTPM: virtualizing the trusted platform module," in *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, (Berkeley, CA, USA), USENIX Association, 2006.

[95] A. Ruan and A. Martin, "TMR: Towards a trusted MapReduce infrastructure," in *Proceedings of the 2012 IEEE Eighth World Congress on Services*, SERVICES '12, (Washington, DC, USA), pp. 141–148, IEEE Computer Society, 2012.

[96] "Xtables-addons." http://xtables-addons.sourceforge.net.

[97] A. Montresor and M. Jelasity, "PeerSim: A scalable p2p simulator," in *Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on*, pp. 99–100, sept. 2009.

[98] "Eucalyptus." http://www.eucalyptus.com.

[99] M. Piatek, T. Isdal, A. Krishnamurthy, and T. Anderson, "One hop reputations for peer to peer file sharing workloads," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, (Berkeley, CA, USA), pp. 1–14, USENIX Association, 2008.

[100] "Privacy CA." http://www.privacyca.com.

[101] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project: Design and implementation of nested virtualization," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, (Berkeley, CA, USA), pp. 1–6, USENIX Association, 2010.

[102] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato, "BitVisor: a thin hypervisor for enforcing i/o device security," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, (New York, NY, USA), pp. 121–130, ACM, 2009.

[103] L. Xiong and L. Liu, "Peertrust: Supporting reputation-based trust for peer-to-peer electronic communities," vol. 16, (Piscataway, NJ, USA), pp. 843–857, IEEE Educational Activities Department, July 2004.

[104] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, (Berkeley, CA, USA), pp. 107–113, USENIX Association, 2004.

[105] G. Mackey, S. Sehrish, J. Lopez, J. Bent, S. Habib, and J. Wang, "Introducing MapReduce to High End Computing," in *Petascale Data Storage Workshop at SC08*, (Austin, Texas), pp. 1–6, Nov. 2008.

[106] M. Laclavík, M. Šeleng, and L. Hluchý, "Towards large scale semantic annotation built on MapReduce architecture," in *Proceedings of the 8th international conference on Computational Science, Part III*, ICCS '08, (Berlin, Heidelberg), pp. 331–338, Springer-Verlag, 2008.

[107] J. Ekanayake, S. Pallickara, and G. Fox, "MapReduce for data intensive scientific analyses," in *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, (Washington, DC, USA), pp. 277–284, IEEE Computer Society, 2008.

[108] R. Taylor, "An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics," 2010.

[109] "Amazon elastic mapreduce." http://docs.amazonwebservices.com/ElasticMapReduce/latest/DeveloperGuide/index.html.

[110] G. Miller, "A scientist's nightmare: Software problem leads to five retractions," *Science*, vol. 314, Dec. 2006.

[111] J. Cheney, S. Chong, N. Foster, M. Seltzer, and S. Vansummeren, "Provenance: a future history," in *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, (New York, NY, USA), pp. 957–964, ACM, 2009.

[112] W. Wei, J. Du, T. Yu, and X. Gu, "SecureMR: A service integrity assurance framework for MapReduce," in *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, (Washington, DC, USA), pp. 73–82, IEEE Computer Society, 2009.

[113] Y. Wang and J. Wei, "VIAF: Verification-based integrity assurance framework for MapReduce," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pp. 300–307, IEEE, July 2011.

[114] "Hadoop tutorial." http://public.yahoo.com/gogate/hadoop-tutorial/start-tutorial.html.

[115] O. OMalley, "Integrating Kerberos into Apache Hadoop." http://www.kerberos.org/events/2010conf/2010slides/2010kerberos_owen_omalley.pdf.

[116] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, "Airavat: security and privacy for MapReduce," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, (Berkeley, CA, USA), pp. 20–20, USENIX Association, 2010.

[117] A.-R. Sadeghi, M. Selhorst, C. Stüble, C. Wachsmann, and M. Winandy, "TCG inside?: a note on TPM specification compliance," in *Proceedings of the first ACM workshop on Scalable trusted computing*, STC '06, (New York, NY, USA), pp. 47–56, ACM, 2006.

[118] J. Lyle and A. Martin, "On the feasibility of remote attestation for web services," in *Computational Science and Engineering, 2009. CSE '09. International Conference on*, vol. 3, pp. 283–288, Aug 2009.

[119] "Hadoop MapReduce WordCount example." http://wiki.apache.org/hadoop/WordCount.

[120] F. Armknecht, Y. Gasmi, A.-R. Sadeghi, P. Stewin, M. Unger, G. Ramunno, and D. Vernizzi, "An efficient implementation of trusted channels based on OpenSSL," in *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, STC '08, (New York, NY, USA), pp. 41–50, ACM, 2008.

Appendix A

Abbreviations

Acronyms

AH Attestation History

AIK Attestation Identity Key

BLP Business Logic Provider

BLS Business Logic Services

cCoT Cloud Chain-of-Trust

CCR Cloud Configuration Register

cdCoT Collaboration Domain Chain-of-Trust

CEN Centralized Attestation Scheme

CML Cloud Measurement Log

CMLS Cloud Measurement Log Storage

comCoT Compositional Chain-of-Trust

CoT Chain-of-Trust

cRoT Cloud Root-of-Trust

CRTM Core Root of Trust for Measurement

CSE Cloud Services Enforcer

CSP Cloud Service Provider

cTCB Cloud Trusted Computing Base

ctf connection trust function

DA Decentralised Attestation

dCoT Domain Chain-of-Trust

DECEN fully Decentralized Attestation Scheme without reputation systems

GTM Global Trust Metric

IaaS Infrastructure as a Service

IS Infrastructure Services

ISP Infrastructure Service Provider

itf initial trust function

LTV Local Trust Vector

NT NeuronVisor with Transitive Trust

PaaS Platform as a Service

PCR Platform Configuration Register

rCoT Resource Chain-of-Trust

REP RepCloud Attestation Scheme

RoT Root-of-Trust

SaaS Software as a Service

SGTM Sliced Global Trust Metric

SLA Service Level Agreement

SML Storage Measurement Log

SoP Separation-of-Powers

SPD Software Property Definer

SS Supporting Services

SSP Supporting Services Provider

stf set of trust function

TCB Trusted Computing Base

TCG Trusted Computing Group

TER Trust Evidence Reporter

TI Tampered Interactions

TMR Trusted MapReduce

TPM Trusted Platform Module

TTP Trusted Third Party

VM Virtual Machine