P-Cop: Securing PaaS Against Cloud Administration Threats

Bruno Miguel Cruz Braga

Thesis to obtain the Master of Science Degree in
Telecommunications and Informatics Engineering

Supervisor: Nuno Miguel Carvalho dos Santos

Examination Committee

Chairperson:	Prof. Paulo Jorge Pires Ferreira
Supervisor:	Prof. Nuno Miguel Carvalho dos Santos
Member of the Committee:	Prof. Henrique João Lopes Domingos

May 2016

Acknowledgements

I would like to thank my parents for their patience and support through the easiest and hardest moments of this thesis. Without them, I would most likely give up on it during those stressful times. I would also like to thank my thesis advisor, Nuno Santos, for his good advice, patience and support. Without him, P-Cop would not have been possible. And since only sufferers understand suffering, I would like to thank my big friend and colleague Sérgio Alves for the help and suggestions, as well as the long conversations about the accomplishments and setbacks of our days of work.

Lisboa, May 2016

Bruno Miguel Cruz Braga

For the ones on the verge of giving up, carry on because you are almost through

Resumo

Os serviços de cloud tornaram-se cada vez mais populares. Programadores e fornecedores de aplicações procuram infraestruturas de Plataform-as-a-Service (PaaS) para correr e escalar as suas aplicações com custo e esforço relativamente baixos. No entanto, o modelo de PaaS acarreta riscos de segurança para os seus consumidores. Fornecedores de PaaS são confiados com aplicações e dados associados potencialmente confidenciais, que podem ser expostos ao público ou alterados por administradores. Para proteger a infraestrutura de cloud de ameaças administrativas, alguns sistemas baseados em tecnologias de computação confiável foram propostos em trabalhos de investigação. No entanto, apesar da sua eficácia, estes sistemas focam-se em Infrastructure-as-a-Service (IaaS), na qual a pilha de software a ser mantida é mais simples do que a de PaaS. Por outro lado, há técnicas interessantes para tornar segura a gestão de infraestruturas de PaaS, mas assumem que o fornecedor de cloud é confiável. Este trabalho propõe P-Cop: uma solução que utiliza hardware confiável (nomeadamente Trusted Platform Module or TPM), presente em hardware comum, combinado com mecanismos de controle de acesso, segregação de competências e auditorias às ações de administradores de sistemas para fornecer uma arquitetura segura para infraestruturas de PaaS. Este trabalho apresenta a investigação relacionada com este tema, descreve a concepção e implementação da um protótipo da solução, assim como a avaliação deste último.

Abstract

Cloud services are increasingly popular. Developers and application providers seek Platform-as-a-Service (PaaS) infrastructures to deploy and scale their applications at relatively low cost and effort. However, the PaaS cloud model entails security risks for its customers. PaaS providers are entrusted with applications as well as potentially confidential data processed by them, which may be leaked or tampered as a result of ill-configured cloud infrastructure software. To secure the cloud infrastructure from cloud administration threats, some systems have been proposed based on trusted computing technology. However, despite their effectiveness, these systems focus on Infrastructure-as-a-Service (IaaS), in which the software stack to be maintained is considerably simpler than in PaaS. On the other hand, there are interesting techniques for securing the management of PaaS infrastructures, but they assume that the cloud administrator is fully trusted. This work proposes P-Cop: a solution that leverages trusted computing hardware (namely Trusted Platform Module or TPM), present on commodity hardware, combined with access control mechanisms, segregation of administrative domains and auditing of administrators actions to provide a secure architecture for PaaS infrastructures. This work surveys and discusses previous research related to our work, describes the design of our solution, its implementation as well as its evaluation.

Palavras Chave
Keywords

Palavras Chave

Administradores de sistema

Auditores

Auditorias

Computação confiável

Computação na nuvem

Plataforma-como-Serviço

Segurança

Keywords

System administrators

Auditors

Audits

Trusted computing

Cloud computing

Platform-as-a-Service

Security

Contents

List of Figures

List of Tables

Acronyms

IaaS: Infrastructure-as-a-Service

PaaS: Platform-as-a-Service

RBAC: Role-Based Access Control

TC: Trusted Computing

TPM: Trusted Platform Module

1 Introduction

Cloud services bring the promise of nearly unlimited computing resources for customers who entrust such infrastructures with their workloads. Although it is foreseeable that cloud services will increasingly prevail in the future, security concerns of different sort are still a major deterrent for potential customers (29; 15). While some security threats are external, i.e., driven by outsider agents (e.g., hackers, misbehaved tenants), others are internally caused by some agent operating from within the cloud (15).

A particularly concerning internal security threat involves cloud administrators. Cloud administrators are responsible for managing the cloud infrastructure. To perform their jobs they require privileged access to the cloud nodes where the customers' computations take place. However, there is the risk that a cloud administrator with unlimited privileges (in the worst case) can control the entirety of the cloud node state, which includes the customers' data being processed on the node. Any mistake performed by the administrator under such privileges (let alone a deliberate action) can have catastrophic consequences as far as compromising the confidentiality and integrity of customers' data is concerned.

To mitigate this problem, researchers (29; 30) have proposed security extensions for the cloud based on trusted computing. One of such representative proposals is the Trusted Cloud Computing Platform (TCCP) (29). Essentially, the idea of TCCP is to prevent cloud administrators from accessing raw customers' computation state by leveraging a hardened hypervisor and a piece of trusted hardware. The hardened hypervisor sits in the most privileged domain of the system and confines the administrators' operations to a sandboxed container isolated from guests virtual machines (VMs), where sensitive users' data is held. From his container, the administrator is allowed to control only the life cycle and resources of guest VMs; he cannot read or write the VMs raw memory and storage. The trusted hardware, typically a TPM chip (11) deployed on each cloud node, provides the root of trust for a remote attestation protocol that allows customers to remotely verify that the cloud nodes are running a current instance of

the hardened hypervisor and therefore that it is safe to entrust their data to the cloud service. Thus, assuming that the hardware is secured by complementary physical protection techniques and that the hypervisor code is correct, TCCP delivers end-to-end security in the presence of a rogue software cloud administrator. However, new challenges arise for cloud services that are delivered at a higher layers in the cloud stack.

1.1 Motivation

Although existing trusted cloud architectures like TCCP are well suited for Infrastructure-as-a-Service (IaaS) clouds, new challenges arise in Platform-as-a-Service (PaaS). In IaaS cloud services, e.g., Amazon EC2, the heavy-lifting work on the compute nodes is performed by a hypervisor. The hypervisor exposes to customers a virtual machine abstraction and requires a few simple management operations that can be cleanly isolated from the customers' VMs. As a result, TCCP can be directly applied to IaaS without hindering the maintainability of the cloud service. PaaS cloud services entail more complex management operations. PaaS offers to users a compute hosting service at higher level abstraction than VMs, typically under the form of Web applications (8) or Docker containers (6). Hence, the responsibility for maintaining the OS and the containers' runtime is pushed in the cloud stack from the customers down to the cloud administrators. On the positive side, this shift relieves PaaS customers from the burden of maintaining the guest VMs' OSes, which explains why PaaS is becoming increasingly attractive over IaaS. Side-effect, however, is that cloud administrators need to perform more potentially invasive operations to the customers' computations. In particular, they require fine-grained control over the underlying OS and container runtime in order to install, update, and configure the software, diagnose and fix potential problems, etc. Consequently, the clean separation between administrator and user-domain offered by TCCP can no longer be easily enforced, at least without a significant loss in the maintainability of the PaaS service. The problem is then how can PaaS services be secured without hindering the maintainability of the infrastructure.

1.2 Goals

This work attempts to address the threat represented by cloud administrators' on PaaS environments, while introducing little or no restrictions on their administrative tasks. To address

this trade-off between security and maintainability in PaaS services, we propose an alternative approach to TCCP called P-Cop (PaaS-Cop). The key insight is that, instead of never granting full administrative privileges on the compute nodes —the TCCP approach— we allow the administrator to acquire such privileges while protecting cloud customers workloads. To guarantee the security of customers' computations, P-Cop solution incorporates three main ideas:

- *Independent operational and auditing teams:* Operational team manages nodes hosting customers computations while the auditing team is responsible for auditing their actions. Both teams are responsible for choosing the software supporting the PaaS infrastructure. The operational team is hired by the cloud provider while the auditing team may have members hired by the customers.
- *Operational states:* Compute nodes (host customers' applications) can be in one of two states: verified and unverified. Verified nodes run customers applications while unverified nodes don't. A node switches from verified to unverified as soon as an operational administrator enters it (e.g. through SSH), and remains as such while the auditing team scrutinizes the actions of the latter.
- *Distributed remote attestation:* Developers assess the level of security of nodes hosting applications, based on TPM's remote attestation.

The focus of this project was to design and prototype a system to secure PaaS platforms called P-Cop. A basic PaaS software, that uses Docker to run developers' applications, was developed and we added P-Cop's security-related components: an auditing hub, node guards for computing nodes (hereby called minions) and mutually authenticated SSL and SSH communication channels. The auditing hub logs administrative sessions for auditing purposes and interacts with node guards (software running on nodes running customers applications) to protect the platform before the administrator enters. Both components cooperate to switch nodes states.

1.3 Contributions

This work, P-Cop gives the following contributions:

- Methodology for segregation of administrators in non-collusive teams

- Mechanism to audit administrators' actions
- Distributed attestation protocols based on trusted computing

The main results produced by this thesis may be enumerated as follows:

- Logging system for SSH sessions
- A set of distributed protocols for attestation and secure communication between PaaS nodes and entities.
- P-Cop prototype

The implemented prototype works with Docker, and can be integrated on typical PaaS environments, without major changes. Its interoperability stems from the fact that it only requires the deployment of one or more logging nodes, and the integration with the remaining components through SSH and simple P-Cop services running on the monitored nodes. To the best of our knowledge, P-Cop is the first solution presented to address administrative threats on PaaS environments.

1.4 Research History

Since the beginning, P-Cop has been aimed at being easily integrated and interoperable with typical PaaS infrastructures. Our first PaaS base environment was RedHat's OpenShift which is open source and could be modified to our purposes. Yet, we faced difficulties on getting OpenShift running and, since we could not find support to our issues, we decided to develop a basic PaaS environment that allows the deployment and removal of applications inside Docker containers. Over that basic PaaS, we built the remaining components which provided the security guarantees P-Cop provides.

1.5 Structure of the Document

Chapter 2 gives a background of the current PaaS panorama in terms of offered services, as well as current architectures. Trusted computing is also discussed since it will take part in

this thesis. Chapter 3 presents research on PaaS security, as well as, fields that we consider to be of great value to build a secure PaaS environment: trusted execution of software, secure cloud administration, logging and information tracking, as well as, remote attestation. Chapter 4 introduces P-Cop, a distributed architecture to protect PaaS customers' applications from administrative threats. In this chapter we provide an overall description of the system, as well as a detailed description of each of its components and protocols. Chapter 5 describes the implementation of a P-Cop prototype, including the environment and technologies used to support it. Chapter 6 presents the results of the experimental evaluation of our prototype on Emulab nodes, as well as, a discussion of the final results. Chapter 7 concludes this document by summarizing its main points and presenting future work that could enhance this thesis.

2 Background

This chapter aims at clarifying key concepts necessary to understand the next sections. First, we describe the current panorama in PaaS cloud services. We cover the PaaS players, main features and typical architecture (Section 2.1). Then, we introduce trusted computing and make a survey of the security capabilities provided by trusted computing hardware (Section 2.2).

2.1 PaaS Background and Current Panorama

Cloud services are typically provided in three forms: Infrastructure-as-a-Service(IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS) where:

- IaaS: translated into VMs that cloud customers may deploy and manage as they please, as well as storage facilities (namely storage volumes backed by hardware disks). This was the service offered initially by cloud providers such as Amazon Web Services (4).

- PaaS: less low-level, when compared with IaaS and has a model of deployment focused on the application insted of the virtual machine (developers upload application packages to the cloud). PaaS infrastructures are typically leveraged by developers or application providers seeking to deploy their applications in a simple and fast way to scale to potentially large crowds of application end-users.

- SaaS: model where applications hosted by application or cloud providers are made available for use by end-users. PaaS applications or applications hosted on IaaS VMs and accessible through the internet may become SaaS. SaaS takes the form of services (e.g. Goolge Gmail, Youtube, MicroSoft Office 365, Dropbox) provided to end-users and typically supported by cloud infrastructures.

Both IaaS and PaaS providers offer extra services to be used by cloud customers such as databases (e.g. MySQL, Amazon RDS, Google DataStore), caching (e.g. Redis) or queues (e.g. AWS SQS).

There are currently two types of PaaS, referred to in this thesis by: *classic PaaS* and *containerized PaaS*. Classic PaaS was the service initially offered by Google App Engine (GAE) (8), Amazon Web Services Elastic Beanstalk (AWS EB) (3) or Microsoft Azure (5) where users had a limited set of languages and language runtimes. Also, typical classic PaaS providers such as Google, Microsoft or Amazon do not allow customization of the environment where applications run, beyond the hardware features. For that reason, third-party libraries must be included on packages submitted to the PaaS infrastructure. Assuming an application requiring a front-end and a backend, classic PaaS offers a limited set of technologies (e.g. MySQL, Google DataStore) supporting this kind of demand, leading users to be stuck with the technologies offered. Yet, assuming that users' applications do not require complex dependencies or customized applicational stacks, classic PaaS offers a relatively simple deployment environment out-of-the-box.

Containerized PaaS, supported by Docker (6), brings the solution to all previously referred problems, by allowing developers to customize the environments where they deploy their application. Developers may now use any language, runtime, applicational stack or dependencies they need. Furthemore, in the limit, anything can be an applicational container, which means that a user may deploy any frontend and backend technologies, as long as the PaaS provider supports containers technology. Containerized PaaS may require more configurations than classic PaaS which are the cost of flexibility. Currenty, Docker is supported by Azure, AWS, AppEngine and other opens ource PaaS platforms such as Red Hat's OpenShift or Apache Stratos (2). In practical terms, a Docker container is a small and portable file containing developers applications, configurations, libraries and dependencies. It is similar to a small filesystem image.

Regardless of the specific PaaS type, PaaS services tend to adopt a similar high-level architecture depected on Figure 2.1 shows a typical PaaS architecture as well as the referred actors and their interactions with it.Typical PaaS environments encompass four sets of actors: *developers* or *cloud customers*, *application end-users or end-users* (for simplicity), PaaS cloud provider, and PaaS cloud administrators. Developers seek PaaS providers to avoid managing their own cloud and scale their applications easily, so they may focus on developing. They are responsible for uploading their applications to the cloud, manage application versions, and pay for resource

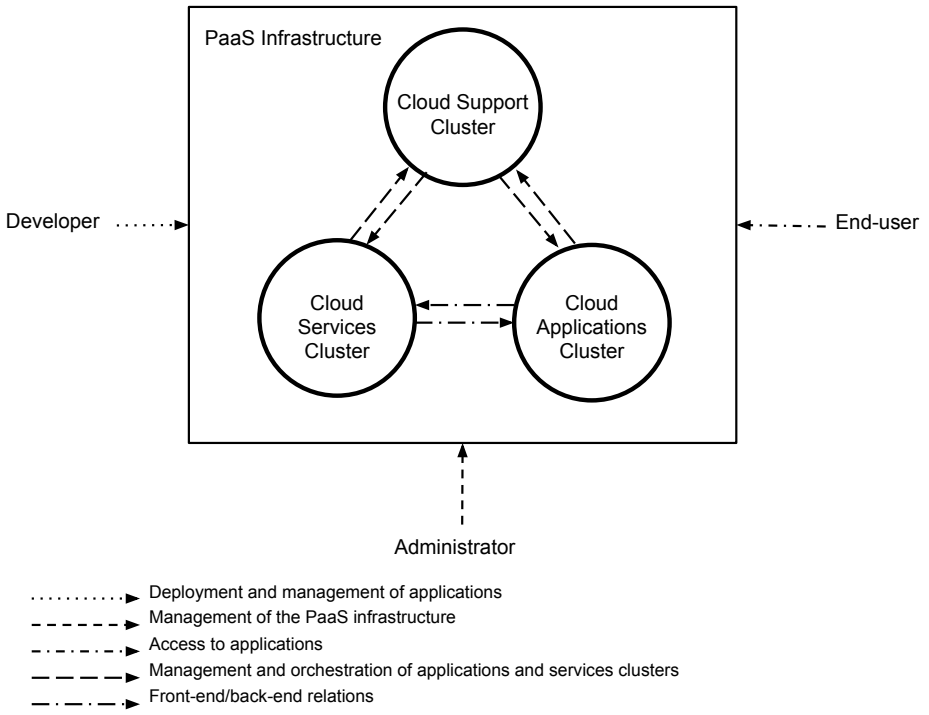

Figure 2.1: Typical PaaS Architecture

usage. End-users access developers' applications running on the cloud and may need to provide them with personal information (e.g. name, age, lifestyle, banking credentials). End-users, typically, have little knowledge of the technologies and resources running behind the scenes when they access cloud applications. The PaaS provider represents the PaaS infrastructure which is managed by teams of administrators responsible for managing cloud nodes and make sure they (cloud nodes) provide the services required by developers and end-users.

In a typical PaaS infrastructure, three clusters are typically present: *applications*, *support*, and *services*. The applications cluster run customers' applications. Since PaaS environments rarely enable customers the possibility of persisting data locally (e.g. create files on the local filesystem), they often rely on backend services (part of the services cluster) offered by PaaS providers (e.g. databases). The cloud support cluster agglomerates all the nodes responsible for: scheduling applications to run on applications or nodes, support infrastructure administration interfaces, repositories of trusted software and stock images for PaaS nodes, as well as PaaS APIs to be leveraged by cloud customers.

Even though developers and end-users play different roles when using cloud resources, they share similar concerns. Both expect:

- **Availability:** application and data should be always available or having as little downtime as possible.
- **Consistency:** new versions of software deployed by the developer should be equally available to all end-users and, if it's not possible, end-users should access the same versions. End-users should be able to access the most updated versions of application-related data or, at least, the same versions.
- **Privacy:** applications as well as data associated with them should not be disclosed to unauthorized entities (e.g. general public, cloud administrators).
- **Integrity:** applications as well as data associated with them should not be tampered with by unauthorized entities (e.g. public, cloud administrators).

Failing to fulfill these requirements can result in leakage, loss, or tampering with of end-users' data. Both cloud provider and cloud administrators, in general, expect these conditions to be met as frequently as possible for the sake of their public image. Overall, cloud customers require a means to assess the security of the PaaS infrastructure they interact with (namely running software, configurations and security policies). Software-level protocols, by themselves, are not enough to provide these data in a reliable manner which justifies the creation of trusted computing hardware and software, which we now introduce.

2.2 Trusted Computing and Trusted Platform Module

In order to ensure a correct behaviour of computing systems, it is fundamental to guarantee that their execution state is well-known. Yet, this is difficult to achieve, specially in complex distributed systems such as PaaS clouds. For instance, when a computing platform boots, it's difficult to assess the security level of it since it may be running, either a secure hypervisor or malicious software (e.g. malware). Trusted computing, a concept proposed by the Trusted Computing Group (11), aims at addressing these concerns by combining hardware and software technologies to enable the construction of trusted platforms, verifiable by external entities.

The security provided by trusted computing is rooted on a trusted hardware component which cannot be tampered with by software running on the host computer. Through a set of

low-level primitives implemented by such hardware, it is possible to build trust on the platform hosting it. The Trusted Platform Module (TPM), a secure co-processor widely available on commodity hardware and servers, is an instance of trusted computing hardware which, at a high level, offers four main primitives that help making a platform trusted: *platform identity*, *measured boot*, *remote attestation* and *sealed storage*.

Platform identity provides a means to identify unequivocally the TPM module and the platform. Each TPM has a (hardware embedded) key pair called Endorsement Key (EK), as well as a set of unmodifiable cryptographic functions. The public part of the EK is signed by the manufacturer as a means to guarantee correctness of the chip and validity of the key.

Measured boot plays an important role on providing a means for external entities to attest the security of the platform since they may learn of the software supporting the service he is accessing. Upon system boot, the first code to be executed (typically pre-BIOS firmware), measures the BIOS. This measurement operation typically involves hashing the BIOS code and extending it to one of TPM's registers (PCRs), where extending means concatenating the BIOS hash with the PCR default value, hash both values and overwrite the previous PCR value. Then, the BIOS performs the same steps on the next software component to be loaded (typically the bootloader) and the process continues until the boot is complete. The final hash is typically called platform configuration and identifies the software running on the platform until the former is booted once more. Measured boot works similarly when UEFI is used instead of BIOS.

Remote attestation aims to provide a remote entity with a means to assess the security of the platform he is contacting, remote attestation allows him to retrieve from the TPM a signature called quote that tells him the resulting platform configuration, built through the attested platform boot, by the loaded firmware and software. Since the EK is unique to a TPM, using its private part for signatures and disclosing the public one for verification, violates the privacy of the platform. Currently, the TPM uses Attestation Identity Keys (AIKs) signed by a Trusted Third Party (TTP) as identities, instead of using the EK. Performing remote attestation requires the challenger to generate a nonce and send it to the attested platform. The TPM creates a quote containing the platform configuration hash and the nonce, signed with the private part of the AIK. Upon reception of the quote, the challenger may verify (using the TPMs public part of the AIK) the signed nonce and platform configuration hash. If the hash is trusted and the nonce is the originally sent, the platform may be trusted. Such facility allows a

challenger to, for instance, learn of the hypervisor's version loaded during boot (assuming the challenger knows the hash of the hypervisor beforehand) and make a more informed decision regarding the platform security (e.g. the remote challenger may trust a given hypervisor with a given hash to be secure). A TPM quote is typically accompanied by a list of hashes that lead to the final configuration.

Sealed storage is a mechanism to help a platform user (either remote or local) that may need a piece of data to be only accessible on a platform with a trusted configuration. Assuming a user encrypts data on a trusted platform running a trusted OS, if the latter is compromised and tampered with, upon reboot, previously encrypted data should not be accessible since the platform is no longer secure. TPM sealing primitives enable data encryption to be linked to a given platform configuration, preventing an untrusted platform from retrieving sensitive data. Seal encrypts data and binds the unsealing capability to the PCR values upon sealing, i.e., in order to unseal data, PCRs must be congruent to the ones used to seal it.

Sealed storage, along with all other TPM primitives are resilient to software attacks performed by the system administrator, who owns root privileges on the computer. As a result, TPMs can constitute a powerful building block for making PaaS infrastructures more resilient to security threats.

2.3 Summary

PaaS environments have been increasingly used by developers and software providers as low-cost platforms to run and scale their applications. In this chapter we started by surveying the current PaaS panorama in terms of: contrast with IaaS, main PaaS players (e.g. Google, Microsoft, Amazon), the differences between classic PaaS and the relatively-recent containerized PaaS (with Docker) and, typical PaaS architectures and actors. This survey will be most helpful on subsequent chapters since the proposed solution, P-Cop aims at seamless integration with the current PaaS panorama. We also provided a brief introduction to trusted computing hardware, in particular TPM technology, as well as its capabilities since the proposed solution relies on such technologies to provide meaningful guarantees to cloud customers. On the next chapter, we survey research proposals that we consider relevant to build a trusted PaaS architecture.

3 Related Work

This section discusses the related work. Section 3.1 surveys previous research on PaaS security. Since such research fails to address the concerns of our work: administrative threats, we then focus on enumerating and describing some solutions, considered relevant as potential building blocks for a secure PaaS environment:

- **Trusted execution environments**: manage to protect PaaS applications from potentially untrusted malicious administrators or compromised OSes (Section 3.2).
- **Secure cloud administration**: limit administrators privileges and actions on cloud resources and customers workloads (Section 3.3).
- **Logging and information tracking**: log administrators' actions on cloud nodes for auditing purposes (Section 3.4).
- **Remote attestation**: provide meaningful attestation to remote challengers (e.g. developers and end-users) (Section 3.5).

3.1 Secure PaaS Platforms

Since our work focuses on delivering a trusted PaaS environment for developers and end-users, we now cover two solutions oriented to securing PaaS, but focused on an alternative potential attacker: the developer. One of the solutions is more oriented to MapReduce computations while the other focuses on any PaaS application.

Airavat (26) is concerned with protecting MapReduce related data (i.e. inputs and outputs) from malicious developers. The threat model addresses situations where a given party (e.g. company) hires the services of developers to process company's data without leaking it or having access to it. MapReduce requires at least two functions: a mapper and a reducer. A mapper

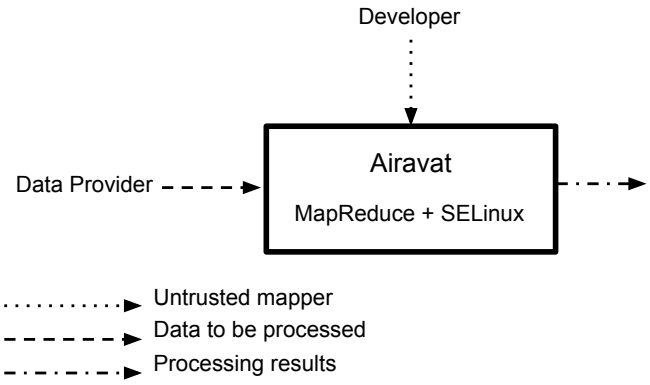

Figure 3.1: Airavat Overall Architecture

receives strings and outputs key-value pairs which, in turn, are provided to the reducer that aggregates those pairs by key. A malicious developer may manipulate the computation and leak the original data directly (e.g. on reducer outputs) or indirectly (e.g. by manipulating the order of reducer outputs according to the occurrence of a given value on the original input). Airavat prevents such leaks by enforcing a strict order on outputs and by limiting accesses to them. Limiting accesses is achieved by having the developer query the results using concrete keys instead of having the system supply the whole reduce output to him, preventing encoding of sensitive data on keys. Airavat forces developers to use a limited set of reduce functions instead of custom ones and introduces noise on reduce outputs to prevent a malicious developer from outputting more or less data according to the existence of some value on the original input. Figure 3.1 depicts Airavat's overall architecture. Airavat uses SELinux to confine the computation and prevent data from being leaked through the filesystem (e.g. prevent a malicious developer from creating files and store computation values there).

Brown and Chase (14) address the lack of transparency for application end-users, which typically trust cloud applications with their (potentially private) data. Malicious developers may deploy applications which misuse end-users' data or provide a service which is different from the advertised. The authors propose transparency through source code disclosure to application end-users. The deployed application is fingerprinted (hash of the source code is computed) and, both this hash and the source code are made available to the application end-user. Once the application is deployed, the developer cannot change the code (except by uploading a new version), assuring end-users that they may inspect the source code of the application they are accessing.

Discussion: Both solutions, alone, cannot address the main issue addressed in this thesis: mitigating administration threats in PaaS environments. Airavat is strictly oriented to MapReduce, providing no usable ideas to secure a generic PaaS infrastructure. While we could propose the disclosure of PaaS-supporting software and source code to the public (i.e. developers and end-users), as proposed by Brown and Chase (14), we find such solution intrusive and weak in terms of security since providers may not agree to disclose their software stacks to competitors and hackers may learn of and attack vulnerable ones, while developers and end-users wouldn't be willing to scrutinize complex software. Since research on PaaS security is limited, the following subsections describe cornerstones to build secure PaaS environments.

3.2 Trusted Execution Environments

This section focus on research on isolation of applications running in potentially untrusted environments. Cloud infrastructures support workloads from multiple customers and encompass potentially large software stacks to support them. Large software stacks are prone to bugs which undermine the security provided to customers' applications. Also, administrators may leverage their privileges and tamper with or inspect customers workloads on cloud nodes under their rightful administrative duties.

Research on isolated execution of applications relies on a combination of hardware and software, and offers remote challengers the possibility of attesting it (the isolated execution). Isolated execution is typically achieved using virtual machine monitors to isolate applications or, running security critical applications or code segments under protection mechanisms provided by hardware (e.g. AMD SVM (1), Intel TXT (9), Intel SGX (9), customized CPU). As for attestation, most solutions rely on TPM while others use other mechanisms (e.g. attestation provided by Intel SGX or attestation using a customized CPU).

At a high level, all architectures share a similar architecture as depicted in Figure 3.2. The trusted computing base component may be either a piece of software (e.g. trusted hypervisor, trusted library), or hardware (e.g. TPM, Intel SVM), or a combination of both. It supports the isolated execution that protects security-critical code running on the trusted environment from the untrusted software running on the same platform (e.g. OS, applications).

Terra (18) leverages a virtual machine monitor (trusted virtual machine monitor) to divide

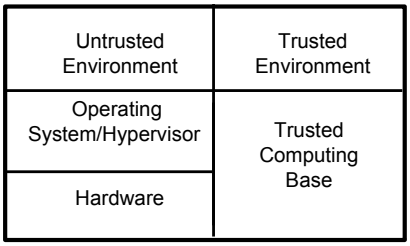

Figure 3.2: Trusted Execution Common Architecture

compute resources into isolated VMs protected even against node administrators. Terra also relies on tamper resistant hardware with properties similar to the TPM (e.g. unique hardware-embedded cryptographic identity signed by the manufacturer) to provide a way for a remote challenger (e.g. cloud customers) to attest the loaded software and the integrity of his VMs. Software and firmware measurements as well as remote verification by challengers are slightly different from TPM's model: in Terra every piece of software and firmware loaded must request the previously loaded one to sign a public key associated to a certificate representing the current component (the certificate attests its hash). The certification chain is rooted in the hardware component key. A remote challenger must verify the whole chain and the associated and certified component hashes. This solution was one of the first oriented to trusted computing and implements a measurement and verification model slightly different from the TPM while providing the same guarantees.

Instead of running applications in VMs to enforce isolation, Flicker (21) relies on hardware facilities to run pieces of code in isolation, which fulfills its main objective: reduction of the trusted computing base. The solution is aimed at running a piece of code in a confined environment provided by AMD's SVM or Intel's TXT instructions extensions, typically used to run VMMs or secure kernels on highly privileged layers (more privileged than the host OS kernel) with protection against software-based attacks. The attestation capability provided by Flicker is similar to a typical TPM attestation mechanism (challenger sends nonce and TPM signs nonce, together with PCRs values). Yet, while a typical TPM PCR state value is a chained hash of the multiple hashes of software loaded by a platform upon boot, Flicker supplies a chained hash of

code, inputs and outputs. Flicker also offers the possibility of saving state across Flicker executions by leveraging TPM's seal facilities, under the hash of the code running. Replay attacks by the untrusted operating system are avoided using TPM's non-volative memory to store counters (state versions) and by sealing data, together with those counters.

The successor of Flicker, TrustVisor (20) addresses its performance limitations imposed on multi-tenant environments (i.e. one TPM to attest multiple software instances and execution delays introduced by having a single isolated slot for every running application) and usability limitations (i.e. developers required to change code and link their applications with Flicker modules). As opposed to Flicker, TrustVisor relaxes the restrictions on TCB's size by using a special-purpose hypervisor responsible for isolating code from the rest of the running software. Application code to be executed must be registered through a hypercall (developers must provide blocks of code and know what pieces of their applications are security-critical) and then, TrustVisor provides an isolated environment hosting the application code. TPM's slowness is addressed by emulating the hardware TPM using software with micro-TPMs associated to isolated executions (one for each executing piece of code). Micro-TPMs measure the code execution (code, inputs and outputs) and the hardware TPM measures TrustVisor. Remote challengers must attest both the micro-TPM and the hardware TPM.

CloudVisor (32) takes advantage of hardware support (e.g. Intel TXT, AMD SVM) to run a security monitor on highly privileged layers (similarly to TrustVisor). It is designed to isolate VMs and protect them from a potentially compromised virtual machine monitor or from malicious administrators leveraging management VMs to access cloud customers workloads (similarly to Terra). CloudVisor protects the privacy and integrity of resources owned by VMs (e.g. CPU registers, disk storage) while keeping the VMM responsible for allocating and managing resources used by VMs. The VMM is responsible for initializing the platform, passing then the execution to the security monitor. Upon booting, the processor measures the security monitor to prevent the VMM from booting a tampered version of it and stores the measurements on TPM for attestation purposes.

While Intel TXT and AMD SVM technologies used by previous solutions protect code (i.e. applications, VMMs, security monitors) by running it in a protected environment enforced by hardware, they fail to address hardware attacks on memory (data stored on memory by the isolated code is in plaintext and has no integrity protection). Owusu et al. (23) propose the

isolation of applications as well as remote attestation based on a set of CPU instructions that extends the typical CPU instructions set. OASIS (the name of the solution), provides a more ambitious threat model where, besides protecting against a compromised OS, persistent threats (e.g. malware), compromised firmware, it also protects against maliciously attached peripherals and eavesdrops on memory or PCI buses, while assuming the CPU cannot compromised. Code is executed inside the CPU's SDRAM and any secure-critical data is cryptographically protected before leaving it. As some of the previous solutions, OASIS instructions support remote attestation of code, inputs and outputs, and provides state persistence and rollback attack prevention by sending to end-users the hash of the state which may be sent later to identify and resume the execution.

Haven (12), similarly to OASIS, protects security-critical applications from software and some hardware attacks. When compared to Flicker, CloudVisor and OASIS, Haven is a more flexible solution for process isolation since it is capable of running complex legacy applications (instead of pieces of code) as well as enabling them to interact with the untrusted environments (e.g. file-system). Haven leverages Intel SGX, used to run pieces of code in isolation using segments of encrypted and integrity-protected memory, to store application state when it leaves the CPU, as a basis for isolated execution. While Intel SGX allows interaction with components outside the isolated environment, secure code must be aware of potential attacks from the untrusted environment (e.g. OS system call may return malicious values to disrupt isolated applications). As such, Haven uses Drawbridge (7) and extends it to secure the interactions with the untrusted environment. Drawbridge is a system for sandboxing of Windows applications consisting of two core mechanisms used by Haven: picoprocess (protects host from shielded code in Haven) and library OS (a version of Windows 8 refactored to run as a set of libraries). Haven extensions to Drawbride perform checks to data returned by the untrusted environment to the isolated one and implement some OS facilities in order to rely less on the OS: threading, virtual memory and secure persistent storage.

Previous solutions rely on hardware to run pieces of software considered security-critical by developers or VMM's to provide secure execution of VMs. Some of the latter required developers to specify the pieces of security-sensitive code (e.g. TrustVisor). AppSec (25) requires no software changes and takes advantage of Intel TXT or AMD SVM to run a VMM that protects running applications against malicious operating systems by: verifying linked dynamic shared

objects (e.g. strcpy in a malicious loaded library may pass private data to a compromised kernel), enforcing addressing spaces' isolation (e.g. kernel inspecting applications' memory pages) and protecting human-machine interactions (e.g. kernel intercepting keyboard or mouse interactions). AppSec describes the security-critical applications as sensitive applications and relies on the VMM to intercept system calls and verify if the kernel is authorized to access the required memory ranges. If the ranges belong to sensitive applications the access is denied.

Discussion: PaaS environments are expected to host potentially complex software stacks (e.g. Ruby application running on Ruby Rails Framework inside a Docker container). Flicker and TrustVisor require the developer to choose pieces of code considered security-critical in his application which limits usability. While Haven solves the previous issue (by allowing full applications to run), it is incompatible with current PaaS and would require relatively deep changes to the current PaaS frameworks. While Haven relies on existing CPU architectures, OASIS proposes a modified one where the user must supply code and inputs and verify outputs. We consider this solution even more impractical since it assumes a model where developers request the execution of an application wich is expected to provide deterministic and verifiable outputs, which is very different from a PaaS model (application serving multiple requests from application's end-users). While AppSec does not require changes to applications and is more PaaS oriented, limiting the access to applications' addressing space may hinder debugging and troubleshooting on PaaS nodes: an infected PaaS node may require a full memory dump for further analysis, which is not allowed by AppSec since it would constitute a violation of PaaS applications' addressing space. Solutions relying on VMs and hypervisors (Terra and CloudVisor) introduce performance penalties when compared to running software directly on cloud nodes, being more suited for IaaS environments.

3.3 Secure Cloud Administration

The first line of defense against abusive administrators was introduced on the previous section. Such line of defense relies on isolation of applications using trusted software and hardware. Since administrators may access cloud nodes to perform administrative tasks and with potentially superuser privileges, a secure PaaS solution requires some degree of access control to limit administrators actions on developers' and end-users' data while allowing administrators to perform required maintenance tasks. Research on this matter is essentially focused on IaaS

environments and leverages trusted hypervisors or access control frameworks (e.g. SELinux) and data migration to protect cloud customers' workloads.

Butt et al. (27) propose a self-service cloud computing that addresses both lack of administration flexibility and security for cloud customers. Cloud customers cannot deploy software to analyse their VMs (e.g. search rootkits) or protect their storage (e.g. encrypt persistent data when it leaves the VMs), relying on cloud administrators to do so. Also, administrators have typically full control over VMs which may lead to data leaks. This solution proposes the idea of VMs domains: administrative domain and customers' meta-domains. The administrative domain (administrative VM) retains most privileges (e.g. start/stop meta-domains VMs, run drivers for virtualized devices, scheduling time-slices) but disallows the domain from inspecting the state of the clients VMs in their meta-domains. A customer meta-domain encompasses an administrative VM (equivalent to the node administrative VM but for clients meta-domains), typical VMs running customers workloads, service VMs (providing services for customers VMs in the same meta-domain such as transparent encryption for persistent storage) and mutually (by customer and cloud provider) trusted VMs used by providers to verify customers compliance.

Secure Cloud Maintenance (13) focus strictly on protecting customers workloads while allowing administrators to perform the tasks they need. In terms of access control, the architecture encompasses five levels of administrative privileges on compute nodes (enforced by SELinux) and segregation of administrative domains (hardware maintenance team, security team and remote maintainers). The threat model focuses on the remote maintainers, assuming trust in the other two. The security team must define stock images for compute nodes and for the cloud infrastructure software serving the customers.

Figure 3.3 shows the architecture of secure cloud maintenance. Stock software and OS images are stored in the repository managed by the security team (software repository). A maintenance agent runs on compute nodes and is responsible for interacting with administrators extended management interface (e.g. allows them to elevate privileges) and the trust database (e.g. to put the node in untrusted state). This database influences the placement of new VMs performed through the extended management interface since nodes under maintenance cannot host new VMs.

The authors define five levels of privilege on compute nodes hosting customers workloads: *no-privileges*, *read*, *white-listed write*, *black-listed write* and *unlimited*. While the first three levels

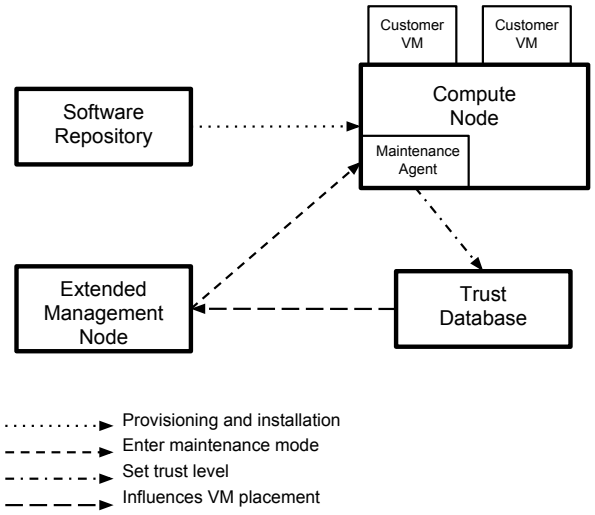

Figure 3.3: Secure Cloud Maintenance Architecture

protect cloud customers workloads, the others do not and require a request for approval sent to customers (through email) before entering the compute nodes. A more efficient approach provided by the authors involves labelling VMs according to workload criticality and make decisions based on those labels, requiring less feedback from customers (some labels still require their approval). If the customers do not approve the maintenance operation or they do not answer, their VMs are migrated from the node and associated data is erased. Nodes under administration are marked as untrusted while the changes to the nodes are evaluated by the security team. Unapproved changes are subject to roll-back based on pre-management snapshots. The solution assumes that there is, at most, one administrator performing maintenance on a node to avoid conflicts on operations and ease the task of the security team, which must evaluate changes.

Discussion: Developers seek PaaS environments to deploy their applications regardless of the underlying supporting stacks (e.g. hypervisor, kernel versions). As such self-service loud computing by Butt et al. adds useless flexibility. Secure Cloud Maintenance provides several ideas that can be applied to a PaaS environment, with no hypervisor support. Segregation of administrative domains (and assuming no collusion exists between teams) is applicable to PaaS cloud environments and offers strong security against administrators since a given administrator cannot control all cloud nodes (e.g. an administrator wouldn't be able to inspect encrypted storage volumes but could easily log directly on a compute node and inspect data directly).

Migrating customers workloads to other nodes when an administrator logs with high privileges is also applicable to PaaS environment since applications have typically no persistent state relying solely on memory state and could be simply killed after spawning new instances on trusted nodes. As for stateful applications such as databases, this solution wouldn't scale as well since cloud supported databases are typically shared by multiple cloud customers which lead to relatively big amounts of data. Relying on the security team to audit the remote maintainers allows the detection of misbehaving administrators. We consider comparing new images against previous snapshots to be relatively complex and unnecessary since it would be more reliable focusing on the interactions between the remote maintainers and the nodes, by scrutinizing inputs (commands) and outputs (extracted data). Also, the solution lacks attestation facilities for developers and end-users to assess the security of the cloud platforms hosting applications.

3.4 Logging and Information Tracking

In order to protect cloud customers workloads, the solutions discussed so far isolate processes, enforce access control policies or, leverage TPM to offer the guarantee that secure components are running on cloud nodes. We consider models where administrators have their privileges limited when accessing nodes remotely, to be cumbersome to implement and use: administrators must decide which OS resources and objects must be protected to design policies and, if the latter become too restrictive, they must be redesigned. Secure Cloud Maintenance addresses this issue to an extent by allowing nodes' administrators to raise privileges as needed. Changes made on nodes are evaluated by comparing VMs snapshots with pos-administration VMs which may introduce high latency and become unreliable. We advocate that administrators' actions on nodes should be assessed based on interactions and not on results. As such, in this section, we survey and discuss solutions that propose logging as a means to understand the changes introduced on nodes by administrators. Architectures that combine logging with information tracking mechanisms allow cloud customers to understand what cloud components and actors process their data. We also discuss complementary work done on processing and analysis of logs to extract malicious patterns in cloud environments.

H-one (17) extends the hypervisor and the management VM kernel to log administrators interactions with customers VMs. Figure 3.4 depicts H-One overall architecture. This solution assumes the hypervisor and management stack kernel are protected from tampering (remote

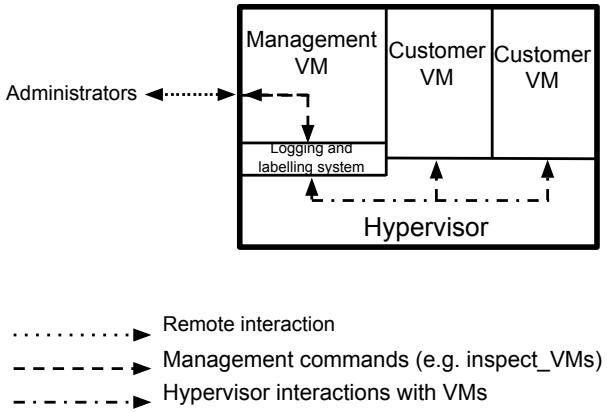

Figure 3.4: H-One Architecture

or physical) by the administrator, i.e he cannot load kernel modules, tamper with the kernel binary or access kernel memory. H-one has four main goals: producing privacy-preserving logs with enough verbosity, protected from tampering and with minimal performance and storage overheads. These objectives are achieved through information flow tracking, which allows the creation of logs oriented to cloud customers VMs (i.e. logged flows are labeled and separated by VM) and minimization of logged data in order to impose minimal performance and storage overhead, as well as protect cloud customers' privacy.

CloudFence (24) also relies on information flow tracking to allow cloud direct and indirect customers (e.g. developers and applications' end-users) a means to control and track their data in IaaS, PaaS and SaaS. The approach requires no modification of applications, which only need to use an API (called API stub) to tag data entering and leaving the application, leaving the process of logging and data tracking to the underlying data flow tracking component (DFT). In order for application end-users to keep track of their data, the authors suggest software providers to supply them with an Universal Unique Identifier (UUID), when they register on their applications, which may then be provided to a tracking service (hosted by the cloud provider) that informs them of the whereabouts of their data. CloudFence also relies on data flow tracking to allow software providers to confine the usage of their data to domains: a social network relying on web servers and SQL backends has its data confined to those services with CloudFence.

Sinha et al. (16) do not rely on hypervisors and propose a less granular auditing system (i.e. logs are not separated by VM) that leverages TPM for integrity protection of logs. The

architecture also addresses concerns such as high-frequency log updates, disk space conservation and, efficient verification of an arbitrary subset of the log. The solution assumes an adversary (e.g. administrator, external hacker) may obtain administrative privileges and take complete control of the system where the logger runs, while allowing it to log the actions that lead to subverting the machine. The system encompasses two entities: a *logger* (running on the logged node) and a *verifier* (running in another node), both sharing a secret key initially, which is typically put on the logger machine by an administrator. This key is used by the logger during execution to generate new keys (one for every log entry) and use them to compute the HMAC of each log entry to attest its integrity. TPM is used to create a blob of the last used key (in case of power failure or on reboot, the logger may retrieve the last key to generate the others), sealed under a counter stored in TPM's secure storage. The external verifier may then retrieve full logs or sequences of logs and the logger may delete older ones to save storage. This work is oriented to protecting logs, not addressing the sources of data that constitute them.

Complementary research to logging and data flow tracking systems, (19) and (22) present solutions to detect malicious insiders through rule based machine learning and sequential rule mining, respectively. On the first work, researchers simulated attacks against VMs and collected resource usage from hypervisors (e.g. CPU usage, received packets) and attempted to fit the latter, through machine learning, into seven categories: no activity, reboot physical machine, malicious insider cloning VM, malicious insider copying everything from a VM, malicious insiders taking snapshot of the VM, installing new guest VM on the same physical hardware, and turn on any guest VM on the same physical hardware. On the second work, researchers attempted to detect malicious insiders by creating profiles based on administrative patterns. They started by logging actions performed by a non-malicious insider to create a non-malicious profile, and comparing that profile with another one created for a malicious insider using sequential rule mining.

Discussion: Protecting customers workloads by limiting administrators' privileges over compute nodes files and configurations is no easy task since issues may arise from any node component which requires administrative intervention. As such, limiting privileges to prevent abuses from administrators may difficult the job of managing the cloud. Instead of controlling administrators based on privileges, an alternative approach is based on logging their actions. The key insight of these systems is that an administrator, knowing their actions are being logged,

would refrain from performing malicious actions. One of the presented architectures, H-One relies on a hypervisor and management VM kernel to label and log data exchanged between administrators and customers VMs. This solution is more oriented to IaaS and becomes of difficult acceptance for PaaS that do not rely on hypervisors.

CloudFence is more generic than H-One since the former may be used on IaaS, PaaS and SaaS, focusing on tracking and confining cloud (direct and indirect) customers' data. The fact that the system implementing the logging and flow control resides on nodes running and processing customers' applications and data, respectively, requires the nodes to enforce some degree of access control to prevent malicious administrators from compromising CloudFence. This is, in our opinion, restrictive for administrative operations.

Sinha et al. (16) does not rely on hypervisors but suffers from issues pointed out by the authors, such as: loss of logs (they are temporarily saved in memory for performance reasons) and the possibility of memory pages containing cryptographic content to be flushed to disk (an attacker may then use the flushed key to adulterate the next log entry). Both solutions suffer from the same problem: in case the logging system ceases to function, it must be restarted or even reconfigured. This gives the administrator responsible for such task a window of opportunity to violate customers' computations and data on the managed node. This work focus mostly on protecting logs, leaving the sources of logged data to be addressed. Such topic is addressed on this thesis.

Research on analysis of logging data (namely (19), (22)) emphasize the importance of logging for further analysis. While logs' analysis is not the focus of our work, we envision P-Cop to be able to process them and produce meaningful alerts. These works are, therefore, complementary to ours.

3.5 Remote Attestation

Previous sections focused on isolating applications and cloud resources from malicious accesses or logging such actions. The focus of trusted computing is to provide a means to assess the level of trust a platform offers by relying on secure hardware (e.g. TPM) to store and sign platform configurations, attestable by external entities and/or platform users. Remote challengers, upon receiving a quote from a TPM, must rely on a hash to assess the security of the platform.

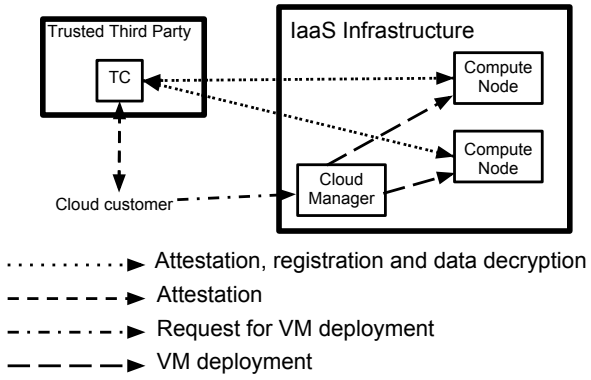

Figure 3.5: TCCP Overall Architecture

The same happens for data sealing where data is sealed based on a hash representing a "trusted configuration". There is a semantic gap inherent to this approach since platform users must trust one or more platform configurations (i.e. chained hashes) to provide meaningful security properties (e.g. privacy, integrity).

This section focus on architectures proposed to address this semantic gap. The most intuitive way to assess the security of a platform is by having a trusted third party mapping platform configurations to meaningful properties (e.g. privacy, integrity). This is the approach taken by most of the architectures presented in this section. Typical trusted parties are entities external to the cloud and inaccessible by the former, or by cloud administrators. Another approach relies on mapping sets of provable properties (e.g. hypervisor version, node location) to cryptographic keys used for data sealing.

Santos et al. (29) propose an architecture that relies on a trusted third party, which is depicted in Figure 3.5. TCCP is primarily oriented to IaaS and guarantees that VMs are executed on nodes enforcing privacy and integrity guarantees against administrators actions, using a trusted virtual machine monitor (TVMM). The solution relies on a trusted coordinator (TC) (server kept by an external trusted entity) responsible for keeping track of trusted nodes inside the cloud environment. Cloud nodes registration on the TC involves a mutual attestation between the TC and the node registering itself. VM deployments and migrations are subject to protocols that require cooperation from the TC to make sure the nodes running the VMs run the TVMM. A remote challenger may then attest the TC (the paper assumes the challenger knows the correct platform TPM configuration) to make sure it has trusted and proper platform

configurations. Customers interact with a cloud manager (public cloud point of interaction) to deploy their VMs.

A similar approach to TCCP is presented in (31) where the TC is called cloud verifier and it runs on the cloud premises. Instead of relying on a trusted third party to assess the security of the nodes, the authors suggest the administrators as the entities responsible for mapping platform configurations to properties by choosing platforms' software in a way that it maps to certain properties required by the customers. Cloud customers may then attest the cloud verifier to make sure it has a trusted configuration and the cloud verifier attests the internal nodes, choosing one that suits the customer's requirements for a platform to be trusted. Sadeghi and Stüble (28) follow the same concept of trusted third party, which is responsible for certifying mappings between platform configurations and meaningful properties, and discuss some approaches to implement the mappings (namely by extending the TPM or running a software attestation service local to the platform and protected by a mikrokernel).

While previous architectures focus on providing a means for remote challengers to assess the properties of cloud nodes, Santos et al. (30) focus on providing a property-based abstraction over TPM's seal primitive which ensures (implicitly) that only platforms with certain properties may decrypt confidential data. Furthermore, the authors address the bottleneck the TPM represents for cloud nodes attestation. Excalibur uses Ciphertext Policy Attribute-Based Encryption (CPABE) to bind encrypted data to a policy. A policy is a logical expression that uses conjunction and disjunction operations over a set of terms (e.g. "location=DE and vmm=Xen"). The ciphertext outputted by a seal based on a policy may only be unsealed by a platform complying with it. Excalibur's design is based on one or more central components called monitors, responsible for mapping TPM identities (AIKs) and fingerprints (PCRs) to attributes, generating then and sending the associated cryptographic keys for encryption and decryption of data to cloud nodes. Only a monitor can contact nodes TPMs, which increases performance and avoids cloud internals exposure. Since the monitor is the component attested by remote challengers and the TPM is relatively slow, TPM attestations are batched, by combining challengers' nonces.

Discussion: Providing meaningful properties to remote challengers (e.g. developers) based on TPM platform configurations is far from trivial and most of the presented work is built on the assumption that there is an entity (e.g. trusted third party, cloud administrators) responsible for certifying mappings between platform configurations (i.e. hashes) and properties (e.g.

confidentiality, integrity). All the proposed solutions share similarities: a component (e.g. cloud node, local protected service) responsible for attesting platforms configurations and translating them into properties. Sadeghi and Stüble (28) propose modifications to TPM or the creation of an attestation service, local to the attested node, and protected by a microkernel, which is not implementable with current commodity hardware or adds more complexity to the compute nodes. Santos et al. (29) and Schiffman et al. (31) are more adaptable to PaaS and rely on a dedicated component (either external or in the cloud premises) that attests cloud nodes. A customer attesting this trusted node and comparing its configuration with one known to be correct may rely on that node to attest and choose only secure nodes for his computations. Excalibur gives more flexibility to cloud customers since they may encrypt their data based on policies which aggregate nodes properties (e.g. location, hypervisor version), and propose a means to scale TPM attestations on the node responsible for attesting cloud nodes. Excalibur suffers itself from a semantic gap between what PaaS developers want (e.g. security, integrity) and what it offers (e.g. location, hypervisor version, OS): a PaaS user expects the platform to protect workloads from administrators, regardless of the node's location and hypervisor.

3.6 Summary

The main goal of our project is to protect cloud customers workloads, in PaaS environments, from administrators, while providing the latter with as much freedom (privileges) as possible. To the best of our knowledge, research on PaaS security fails to address such concern since they focus on a different malicious actor: the developer. In terms of building blocks necessary to create a secure PaaS environment, we surveyed four fields of research: Trusted Execution Environments, secure cloud administration, logging and information tracking, and remote attestation.

Trusted Execution Environments is essential to create a trusted environment for developers applications. Isolation mechanisms proposed by previous research rely on hardware support and/or hypervisors for isolation but are unsuited for PaaS environments since they: limit the complexity of applications, do not scale for multiple applications per node or rely on VMs to host applications (PaaS environments may not rely on hypervisors and VMs for performance reasons).

Architectures on secure cloud administration are oriented to IaaS environments but present

interesting and adaptable concepts for PaaS: segregation of administrative domains, migration of cloud customers workloads, keeping cloud nodes unusable (i.e. not hosting applications or allowing remote accesses) while administrative tasks are audited by a trusted team.

Since limiting administrators actions and assessing levels of privileges over nodes running applications may difficult non-malicious administrative tasks, a more passive approach such as logging of actions may be more usable and adaptable for any cloud environment, while providing similar security (monitored administrators are less likely to violate customers data). Solutions on logging are typically deployed on nodes, adding complexity with limited security: if the hypervisor or software stack responsible for logging actions needs troubleshooting, the responsible for the latter has total access to the system. We consider the concept of logging of most value in cloud environments and, as such, it shall constitute a key part of our final solution.

Cloud customers (e.g. developers) and application end-users require some verifiable guarantees of security which are not provided by trusted computing alone: cloud customers' demand security and privacy but TPMs provide platform hashes. Therefore, we surveyed architectures that translate TPM measurements into meaningful properties. At a high level, all solutions rely on some entity, either external or internal to the cloud environment, responsible for choosing nodes offering security guarantees for customers computations. Customers may then attest those entities using the typical TPM attestation and compare the platform signatures with a well-known and trusted one, which provides transitive security (i.e. trusted entity with correct configuration is trusted to enforce security properties inside cloud premises).

Overall, it becomes challenging to build a PaaS solution based previous research since, PaaS security issues are poorly addressed and protection mechanisms introduce hardware bottlenecks and require customers' applications to be modified. Most presented solutions also rely heavily on hypervisors and become more oriented to IaaS. A new solution that protects PaaS customers' workloads without hindering cloud administrators' actions will be the subject of the remaining document.

4 Architecture

In this chapter we describe P-Cop's architecture. Overall, we aim to provide an architecture that protects PaaS customers workloads against administrators' actions without introducing management limitations that hinder the routines of the latter. On the one hand, we attempt to provide a means for cloud customers to assess the security of PaaS nodes running their applications (by leveraging trusted computing hardware and software) and, on the other hand, we intend to protect data in transit (namely applications transferred between developers and cloud nodes). In the next chapter, we describe the implementation of its prototype.

4.1 Threat and Trust Models

We consider two classes of cloud administrators: software and hardware administrators. Software administrators may log on applications' nodes with superuser privileges and perform any action triggered by software, e.g., reboot them, observe their logs, modify their OSes, access the whole filesystem and deploy persistent threats (e.g. traffic sniffers). The hardware-related administrators are assumed to be trusted. In our work, we add another team: auditing team and rely on segregation of non-collusive administrative domains as will be discussed on section 4.2 described.

We assume all PaaS nodes have TPM and that the latter works properly and does not leak any data. Most hardware attacks are not taken into account (e.g. bus probing, side-channel attacks, memory probing, persistent storage violations). Denial of service attacks (e.g. power outages) against customers or end-users are, also, not taken into account.

PaaS software supporting the cloud infrastructure should be deployed with minimal interaction (e.g. using PXE) as stock images and, is assumed correct as far as promised functionality and security are concerned (e.g. Docker does not corrupt applications and is able to contain them). Current cryptographic protocols and standards, as well as access control mechanisms

provided by OSess or kernel modules are assumed to be trusted. Cloud services are supported by cloud providers and assumed not to run on compute nodes running developers applications. We assume no security-violating actions against these services.

Developers are trusted (intend no harm against applications end-users or the cloud infrastructure) and their applications are assumed to be non-vulnerable (our architecture does not protect against malicious outsiders exploiting applications vulnerabilities). Since we do not address the security of cloud services (e.g. caching, databases), we assume developers protect their data when using them (e.g. using encryption).

4.2 System Overview

Figure 4.1 depicts P-Cops' general architecture. We add a new cluster responsible for logging the actions of administrators on nodes running PaaS applications or cloud services. The monitored nodes will also have P-Cop agents responsible for supporting the logging system. In this work, we refer to the nodes running PaaS applications as minions. Developers interact with the cluster through monitor nodes which support the deployment and deletion of applications ruuning on the minions cluster. Minions belong to the cloud applications cluster while the monitors belong to the cloud support cluster. The operational team is responsible for administrating the cloud nodes while the auditing team is responsible for auditing those administrative actions. Our solution is built on three main ideas:

- Independent operational and auditing teams
- Operational states
- Distributed remote attestation

4.2.1 Independent Operational and Auditing Teams

This idea aims to avoid placing full trust in a single administrator by spreading the management risks across two independent teams: *operational* and *auditing*. While the first team is responsible for managing all PaaS nodes, the second is responsible for auditing the actions of the former on nodes hosting customers' applications, as well as verify that operational administrators deploy approved base images on nodes.

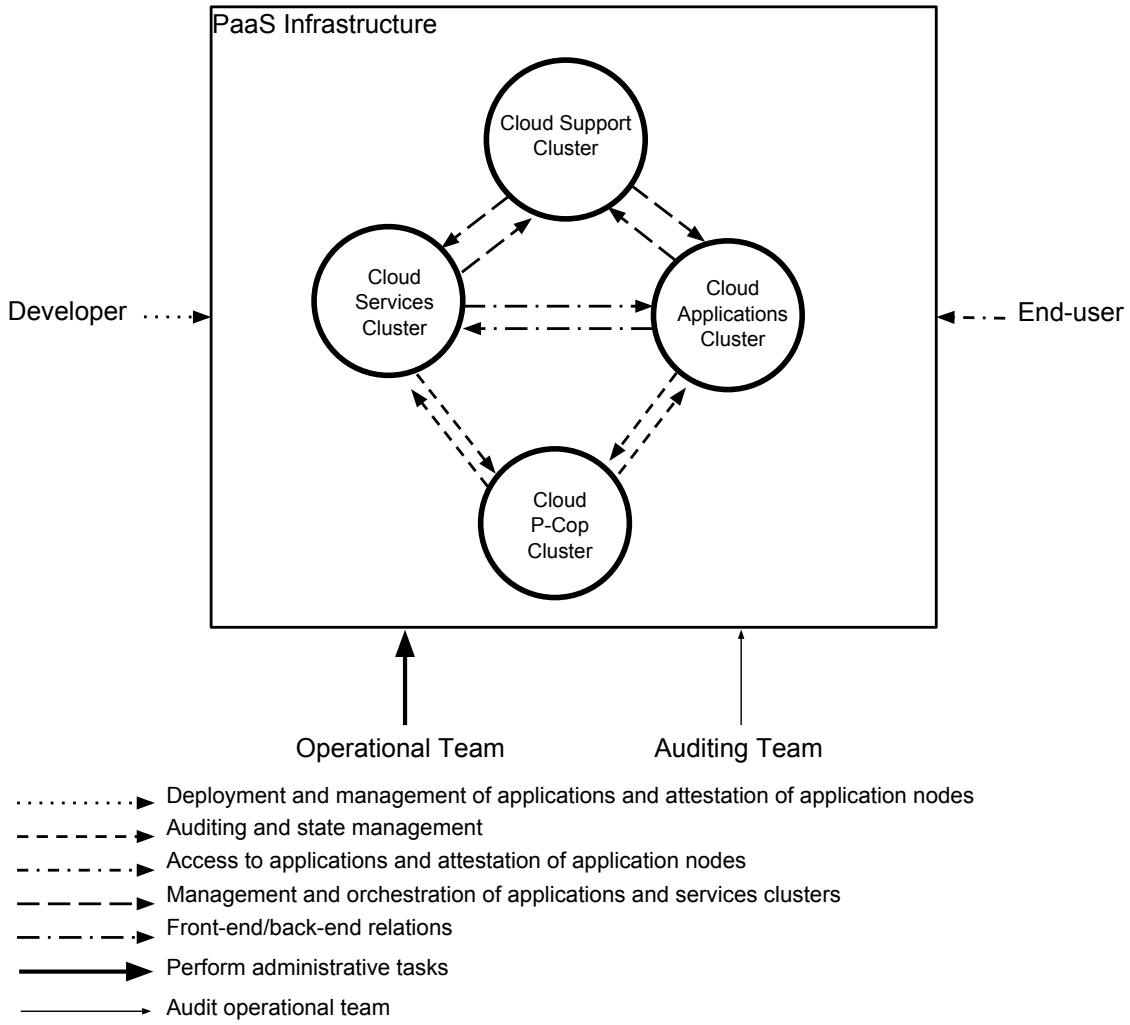

Figure 4.1: P-Cop's General Architecture

Protection of customers workloads is supported by a software agent, called *node guard*, which runs on every minion node. The software supporting the whole PaaS infrastructure is approved by both teams. Typically, aside from nodes hosting services or applications, every PaaS node (e.g. software repositories, management nodes) runs strictly software verified by the administration staff, designed to perform a limited set of tasks (e.g. interact with customers through web interfaces), decreasing its complexity and the possibility of disruptive interactions between running software. On the other hand, nodes running customers' applications or cloud services run applications with potentially complex and diverse support stacks and libraries, as well as managing large amounts of data, making them more prone to unexpected failure.

Our solution does not limit accesses with superuser privileges to nodes running services

or customer applications since with such software complexity, issues may arise from any local component (e.g. configuration file, database restriction). The other cloud nodes limit privileges to the point where an operational administrator may perform his administrative tasks with little or no impediment, while protecting the core, software and configurations supporting the infrastructure, from potentially malicious actions.

Given the superuser privileges on services and applications nodes, our solution requires remote administrative actions to be logged for auditing purposes. Such logging is performed by one or more logging components introduced by P-Cop, called auditing hubs. All logs related to administrative actions on nodes are evaluated by the auditing team.

4.2.2 Operational States

This idea aims to protect customers workloads and proposes two states for compute nodes to operate: verified and unverified. Nodes running applications or services start in verified state (allow applications and services to be ran on them). Upon remote access from an operational administrator, the node guard, kills and deletes current applications (on applications nodes) or protects security-critical data (on services nodes). During maintenance, the managed node is in unverified state, which prohibits it from hosting customers' applications or services. Once the maintenance tasks on that node end, the logged session is subjected to auditing by the auditing team and two situations arise:

- **Auditing team approves changes:** the managed node becomes verified.
- **Auditing team rejects changes:** changes are rolled back, based on stock images and then, the node becomes verified.

Since applications keep no data locally (data is kept on cloud services such as databases), they may be killed and spawned on verified nodes. As for services, they are typically replicated and, a possible approach would be to encrypt and sign security-critical data locally, disabling the services while the node is under maintenance and approval.

4.2.3 Distributed Remote Attestation

P-Cop allows the attestation of monitor nodes by developers. In terms of guarantees provided to developers, P-Cop relies on cloud administrators to choose software and configurations that provide security and confidentiality for customers workloads.

A developer deploying an application must attest the cloud monitor, which in turn has a list of nodes responsible for running PaaS applications (hereby called minions) attested and trusted by the latter. The configuration obtained from the monitor must be compared with configurations signed by multiple auditors and operational administrators.

4.3 P-Cop Features and Actors

In comparison with typical PaaS architectures, P-Cop adds a cluster (Cloud P-Cop Cluster) to the basic PaaS environment, which consists of one or more auditing hubs, responsible for interposing and logging administrative tasks on minion nodes. P-Cop also employs remote attestation mechanisms of monitors, auditing hubs and minions and SSL channels with mutual authentication to provide greater guarantees of security. Another set of actors is also added to the typical cloud environment: auditors. We now describe P-Cop's features and actors, starting with the definition of verified and unverified states for minions, which constitute a most relevant piece of our work.

4.3.1 Minions' Verified and Unverified States

Minions may be in one of two states: verified or unverified. Verified minions are eligible to run developers' applications. Unverified minions are worker nodes which have been recently accessed by operational administrators for management purposes. Since the actions of such actors may be disruptive (e.g. security compromising), no workloads are hosted on these nodes.

In order to be considered verified, a node must display an approved (by one or more auditors and operational administrators) platform configuration (assessed using remote attestation), and must not have been remotely accessed (e.g. managed by operational administrators) or, if it is not the case, the commands and changes performed on the node must have been approved by the auditors.

When a minion is booted, it attempts to register itself with the monitor. Before the minion is accepted by the monitor as a suitable node for running PaaS applications, it must be subjected to remote attestation. If the platform configuration conforms with one approved by the auditors and operational administrators, the minion is considered to be verified and eligible to run developer's applications. As soon as the minion is remotely managed by an operational administrator, it is immediately put on unverified state (i.e. not eligible to run developers' applications). The minion remains in that state until one or more auditors approve the session changes or rolls them back to an approved (verified) state. Both cases require the monitor to attest the minion node once more before setting it as verified.

4.3.2 Auditing Hubs and Auditors

Auditing hubs mediate interactions between operational administrators and minions, and implement a logging system for auditing purposes. In order for an operational administrator to remotely interact with a minion, he must contact the auditing hub which in turn contacts the required minion. Before the administrative session is established, the minion is blacklisted (i.e. auditing hub notifies monitor to cease to deploy applications on it). The auditing hub then notifies the minion's node guard to erase all traces of developers' applications. In order to reduce the impact of P-Cop's solution on the overall times taken to enter a node, the operational administrator may enter as root or non-root. The former requires the minion node to delete all the running applications and confidential data. The second would leverage some role-bases access control mechanism such as SELinux to limit the access of operational administrators on the nodes, as will be described later on this chapter. The second level of access, relies on SELinux to protect workloads while still logging all actions from operational administrators.

Stored logs may then be accessed (i.e. read) remotely by auditors which assess the security of administrators' actions and may then put the managed minions in verified state once more (if no harm has been done) or rollback the actions (if the changes are considered disruptive). In order to put minions on verified state, auditing hubs contact monitors to whitelist such nodes.

4.3.3 SSL, SSH and Remote Attestation

In order to prevent impersonation or eavesdropping on communications between entities (e.g. nodes, actors), P-Cop employs SSL with mutual authentication on all software components. This allows P-Cop entities to communicate, while ensuring confidentiality, integrity and authentication. Aside from messages, P-Cop entities may exchange more complex data (e.g. monitors transfer applications to minions). In order to provide similar guarantees to SSL, SSH is employed.

Developers, operational administrators and auditors, as well as monitors need a way to assess the security of the platforms they interact with (i.e. know what software is running and if the software is approved). P-Cop relies on trusted computing to allow remote attestation on monitors, minions and auditing hubs. More precisely, developers attest monitors, operational administrators attest auditing hubs, monitors attest minions and, auditors attest both auditing hubs and minions. The actors performing attestation assess the security of the platforms based on well-known configuration signed by auditors. All attestations are performed using SSH as the underlying communication protocol. The process to obtain these signatures is described on section 4.4.

4.3.4 Credentials Management and Distribution

SSL certificates used by P-Cop entities are issued by the cloud provider. Also, in order to give stronger guarantees of trust on the provider, its root certificate should be signed by a well-known certification authority (e.g. Verisign). The cloud provider should make available an authenticated signing service used by cloud nodes (e.g. monitors, minions, auditing hubs) and actors (e.g. developers, operational administrators, auditors) to sign their certificates which would be later used on SSL authentication. That service should use the cloud root certificate and due credentials to sign and issue the certificates. The signing service could be located/managed on/by an external and trusted company.

In order to protect the communications between cloud entities and the signing service, the service should employ SSL using a certificate signed by the provider's root certificate. The service must enforce some verification mechanism on the cloud entities to prevent malicious ones from getting trusted certificates. Cloud actors must have a combination of username and

password while nodes must be attested by the service.

In order to protect credentials' integrity and secrecy, they should be protected using secure mechanisms for credentials storage (e.g. Java keystores), i.e., nodes and actors should keep their credentials in a secure storage, locally. Since such credentials' stores have protection mechanisms such as passwords, malicious users (e.g. operational administrators, hackers) may not access them if they don't know the passwords. Such aspect is specially important for minions credentials since operational administrators may access these keystores which are used by node guards. If the passwords protecting such keystores are kept on global variables, they may be erased before an operational administrator enter the node.

P-Cop employs SSH on remote sessions and file transfers. In terms of remote sessions, there may be administrative sessions (i.e. operational administrators connecting to auditing hubs and the latter connecting to minions) or auditing sessions (i.e. auditors access auditing hubs to read logs and commit/rollback administrative sessions). Auditors use their SSH sessions to connect to the auditing hubs and run P-Cop's auditing software that allows them to add SSH users (either operational administrators or other auditors) and access session logs.

Monitors and minions may have git repositories to host developers' applications. As in other cloud providers (e.g. Amazon), developers may upload their SSH keys through a web interface which will later allow them to access their computing instances. A similar approach is used on P-Cop where the keys will be used by the git repositories to receive files over SSH. Transfers between the monitors and the minions, their base images should have all the keys set up.

4.3.5 Applications' End-users Security Guarantees

P-Cop is aimed at securing PaaS applications while allowing seamless usage for the typical application end-user. This is achieved by building a chain of trust that starts on the developer. Each developer should have an SSL certificate issued by a well-known and trusted authority (e.g. Verisign). The developer should use that certificate to sign SSL certificates for each of his applications. This certificate is shipped, together with the application files (which are stored on his computer), to the monitor. The communications between developer and monitor are secured using mutually authenticated SSH and SSL which protects the credentials from snooping or tampering. Also, the developer may attest that the monitor runs trusted and

approved configurations. Then, the monitor is responsible for picking one or more minions, also attested upon registration, to send the files, also through SSH (secure connection). From then on, the application end-user may access the application through its browser which will perform a typical SSL verification, requiring no extra software. Assuming the application end-user trusts that developer intends no arm against him, he may trust that his data is secured. The next section describes the steps to bootstrap P-Cop's cluster.

4.4 P-Cop's Bootstrap Sequence

Operational administrators are responsible for starting all cluster components: monitors, auditing hubs and minions. These nodes are bootstrapped using OS images approved by both operational administrators and auditors. Cloud nodes (i.e. monitors, minions and auditing hubs) require certificates for SSL connections. These certificates could be integrated on the OS base images or obtained by nodes later, by contacting the trusted certification service which would attest the nodes before issuing certificates. The external entity approach allows a better security assessment since this service could be ran by a trusted external and independent entity.

Once monitors are up and bearing SSL certificates, one or more auditors and operational administrators may perform remote attestation on them and compare their configurations with approved ones. If they match, auditors and operational administrators may then send monitors own configurations as well as minions' signed using their respective credentials. Otherwise, auditors/operational administrators request monitors to cease their operations (this requires a set of rejection orders to avoid denial-of-service). A similar procedure is applied on auditing hubs but minions signed configurations are not sent to them.

Monitors are then ready to receive minions' registration requests. Upon registration, minions are remotely attested by monitors and their configurations are compared with the ones signed by auditors and operational administrators. If configurations are approved and match with one or more signed ones, minions are set to verified mode and may, from then on, receive application deployment requests. It is up to the operational administrators and auditors to decide the minimum number of approvals to accept a minion as trusted. If the configurations are not approved, monitors request minions to cease their activities. The cluster is, from then on, ready to receive deployment and administration requests from developers and operational

administrators respectively. The next section describes all protocols employed by P-Cop.

4.5 P-Cop Distributed Protocols

This section describes all protocols employed by P-Cop. We start by describing the protocols necessary to bootstrap the cluster. Then, we describe the protocols used by operational administrators, auditors and developers to manage nodes, audit operational administrators and deploy applications, respectively.

4.5.1 Cluster Bootstrap

P-Cop bootstraps the PaaS cluster in two sets of steps. The first set allows nodes to retrieve SSL certificates to be used later for authenticated communications. The second step requires one or more auditors to attest monitors and minions. Monitors are also attested during the second set of steps. Both sets of steps are depicted on Figure 4.2 and Figure 4.3 respectively.

Starting with the first set of steps, operational administrators deploy approved OS images by both auditors and operational administrators, being deployed by the latter (PXE_LOAD_IMAGE). Once booted, all nodes contact the signing service using SSL secured channels, accepting the certificate of the latter if it is signed by the provider's root certificate, also signed by a trusted public certification authority (e.g. Verisign). In this case, only the signing service certificate is verified. The signing service performs a typical attestation on all nodes. It starts by sending a nonce n (TPM_ATTEST n) to prevent replay attacks, receiving then a certificate request (certificate_request) to be signed and a TPM quote $\langle n, pcr \rangle Key_{TPM}$ where $\langle x \rangle Key_k$ represents x encrypted using the private key of an asymmetric key pair. x represents the previously sent nonce, as well as the contents of the TPM's PCR(s) that store the nodes' bootstrap software signatures. Bearing the TPM's public key, the signing service may decrypt the quote and verify if the nonce matches the one sent previously and the PCRs' contents match the ones approved by the auditors and operational administrators. If both conditions are verified, the signing service creates a certificate and signs it using its certification credentials. The certificate is then sent back to the nodes (OK certificate) which may later use them to authenticate themselves and protect their SSL connections.

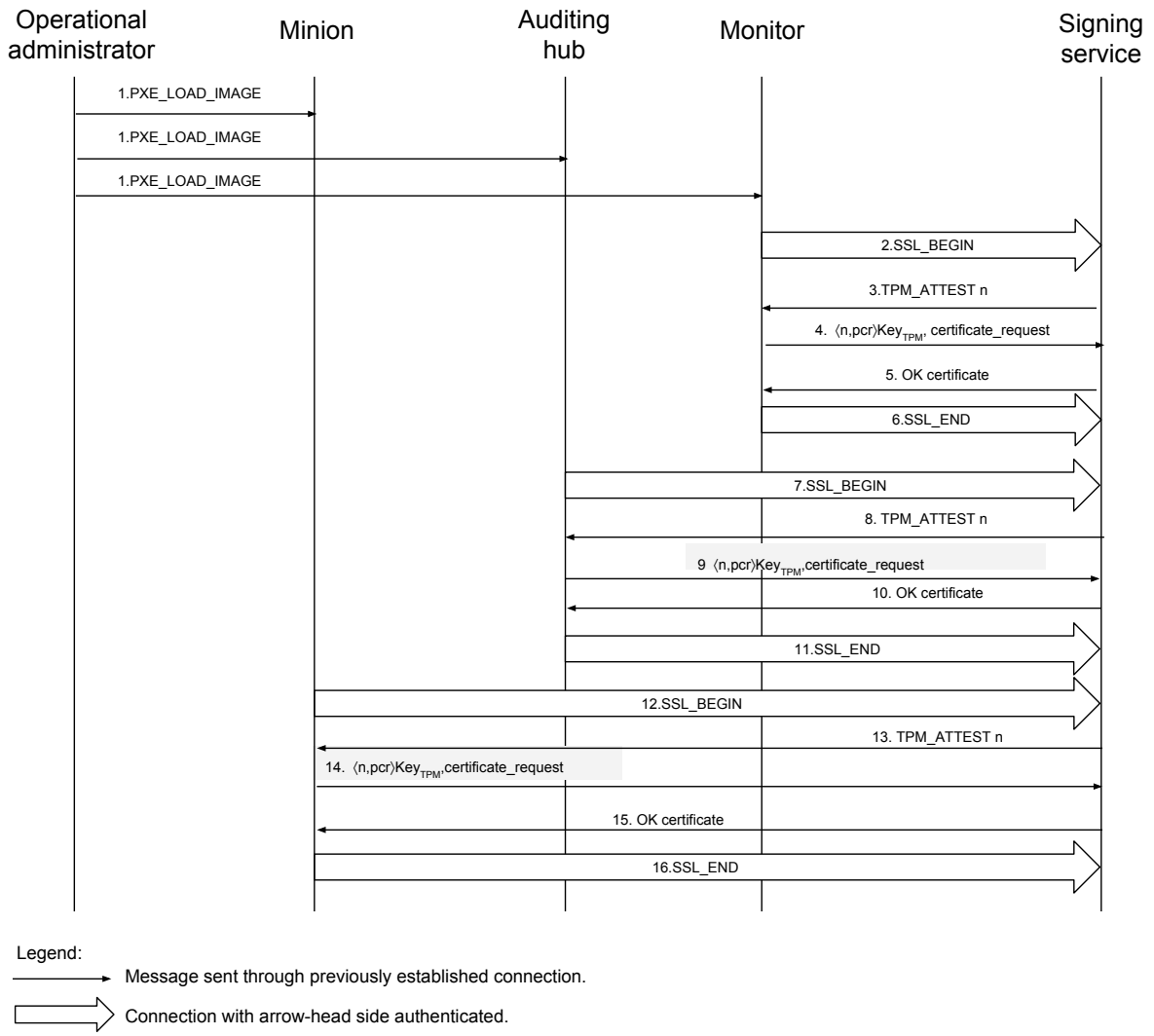

Figure 4.2: P-Cop's First Bootstrap Sequence

In the second set of steps, one or more auditors and operational administrators must attest the main points of trust: monitors and auditing hubs through mutually authenticated SSL channels created using the previously obtained certificates. Figure 4.3 only shows the case where an auditor performs attestation being the same case for operational administrators. Auditors and operational administrators assess the trustworthiness of the certificates by verifying that they were signed by a well-known authority. Only after being attested those nodes may serve developers' and operational administrators' requests. The attestation process is similar to the one previously referred. Yet, if the platforms run the expected configurations, auditors and operational administrators send back approved_pcr and $\{approved\_pcr\}Key_A$ where approved_pcr are the contents of the TPM PCR's used by the platforms to store bootstrap's software

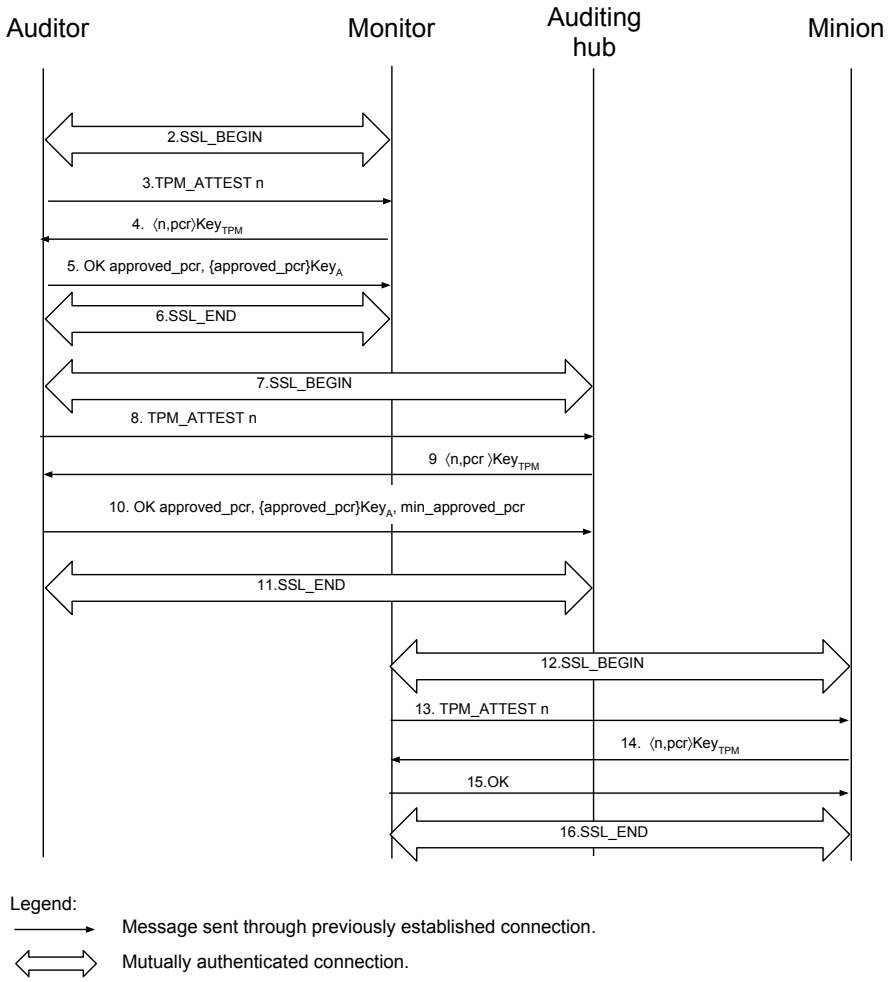

Figure 4.3: P-Cop's Second Bootstrap Sequence

signatures and $\{x\}Key_k$ is a digital signature for x using a private key k (in this case A is the auditor's key). Since monitors are responsible for choosing minions and send them applications to be ran executed, they are also responsible for attesting them before accepting them as trusted. To that end, auditors and operational administrators, upon attestation of monitors send them trusted configurations for minions (min_approved_pcr). The configurations and signatures sent back to monitors and auditing hubs are later used by developers and operational administrators to assess the security of the platforms. If the auditor or operational administrator verifies that the platforms' configurations are erroneous, he requests P-Cop's software agents running on the nodes to cease their activities. Once more, it is up to the operational and auditing teams to agree on the minimum rejections required by the nodes to cease to function. The following sections give an in-depth description of the management, auditing and deployment protocols.

4.5.2 Operational Administrators' Administration Protocols

Besides being responsible for deploying OS base images on cloud nodes and attest the latter, operational administrators may access minions through SSH for administrative purposes. Minions run potentially hundreds of applications with multiple complexities, making them error prone. As such, P-Cop allows superuser privileges over these nodes, through remote SSH sessions. If administrative privileges over minions are required, operational administrators may establish administrative sessions by contacting auditing hubs which will mediate and log all interactions (i.e. commands and responses). Figure 4.4 depicts a management session where the administrator requires root access.

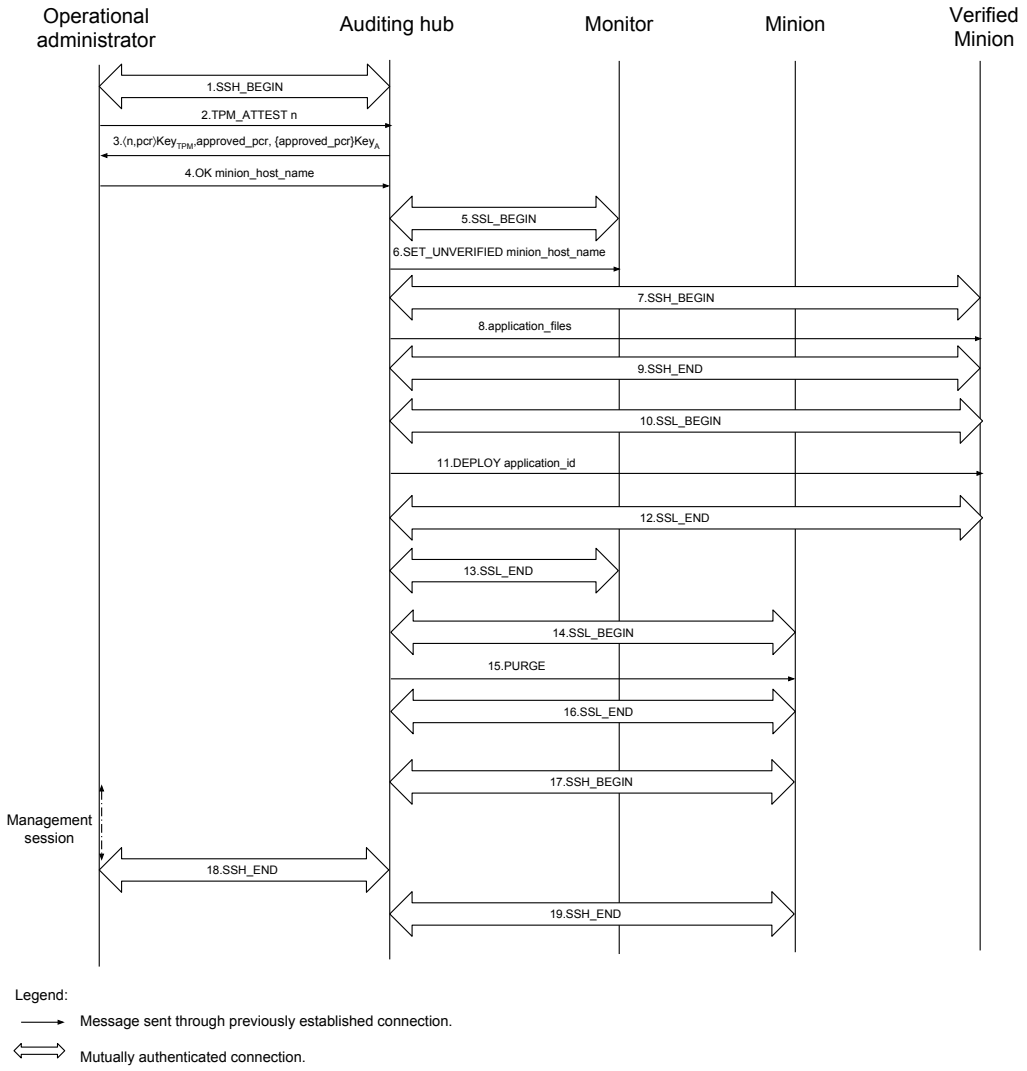

Figure 4.4: Management Session

Operational administrators are responsible for managing monitors and auditing hubs. In

this work, we do not address how these nodes would be protected but a mandatory access control could be employed in a future version of P-Cop. Since the cleaning of minions and the creation of new instances of the running PaaS applications may introduce overhead, we envision that P-Cop could allow to degrees of privilege over minions: superuser and restricted. With restricted, the PaaS applications and security-critical components would be protected using a mechanism such as mandatory access control. Such mode is not addressed in this thesis and will be left for future work.

Before the administrative session starts, a remote attestation is performed on the related auditing hubs to check if they are running configurations approved by the auditors (TPM_ATTEST). The auditing hubs send back to the administrators a set of fields: $\langle n, pcr \rangle Key_{TPM}, approved\_pcr, \{approved\_pcr\}Key_A$. Bearing the TPM's and auditor's public keys, the administrator performs the following operations:

1. Decrypts $\langle n, pcr \rangle Key_{TPM}$
2. Verifies that its nonce (n) is there and that the pcr is the same as *approved_pcr*
3. Verifies the signature of the auditor.

With all these verifications, the administrator verifies that the platform has a configuration previously approved by the auditor. Then, the auditor sends the desired host to be managed (OK minion_host_name). Through an SSL connection, the auditing hub requests the monitor to remove the minion from the list of verified nodes (SET_UNVERIFIED minion_host_name) disallowing further application executions on it. The latter instruction triggers a deployment of the applications running on the minion on one or more verified nodes to replace the missing instances (steps 7-12). Once the the applications are scheduled to run on replacement minions, the auditing hub contacts the minion to be managed and notifies it to clear all security-sensitive data such as applications and credentials (PURGE). After this, the auditing hub establishes an SSH session with the minion and redirects all traffic from the administrator's SSH session to the latter. All traffic crossing the bridged connection is logged locally on the hub. Once the administrator is done with the minion, he closes the session with the hub and the hub closes the session with the minion. The logs are now ready to be evaluated by the auditors as described on the next subsection.

4.5.3 Auditing Protocols

Auditors' roles encompass checking if monitors and auditing hubs run correct base configurations, as well as to audit operational administrators' actions on minions. The first role was described previously while the second is described on this section. Auditors access the auditing hubs through SSH and run P-Cop's auditing software to read logs related to administrator's management operations on minions and commit/rollback their actions if non-malicious/malicious respectively. Figure 4.5 depicts a committed session while Figure 4.6 depicts a rolledback one.

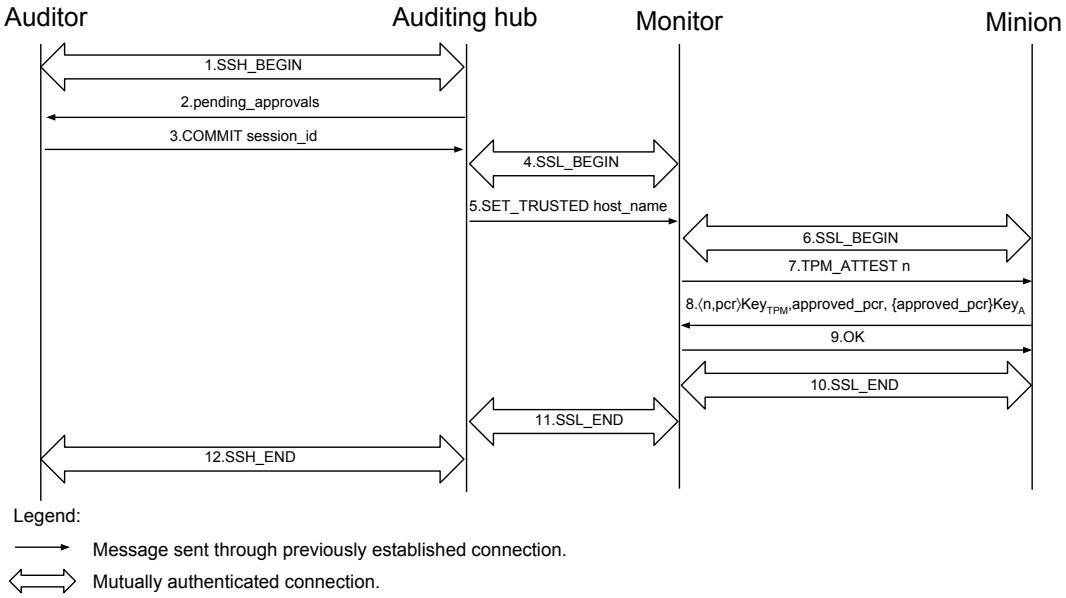

Figure 4.5: Committed Management Session

First, the auditor checks the sessions requiring auditing (pending_approvals). Then, he checks the logs related to those sessions. Internally, logs are files with names that allow auditors to identify the auditor, the minion and the time when the session began. Assuming the operational administrator operadmin starts an administration session on minion managedminion.provider.cloud at 11:13:05 on 17 November 2015, the log file would be called: operadmin@managedminion.provider.cloud_17-November-2015_11-13-05.log. The log structure allows the auditors' to check the flow of commands and data across the administrative session. Figure 4.7 depicts an example of a session log. Session logs give enough information for auditors to understand what data crossed the connection (Admin->Host and Host->Admin), as well as when such data was logged (e.g. Nov 17, 2015 11:14:05 AM). After the flow direction and before the next timestamp are the commands and responses from administrators and minions,

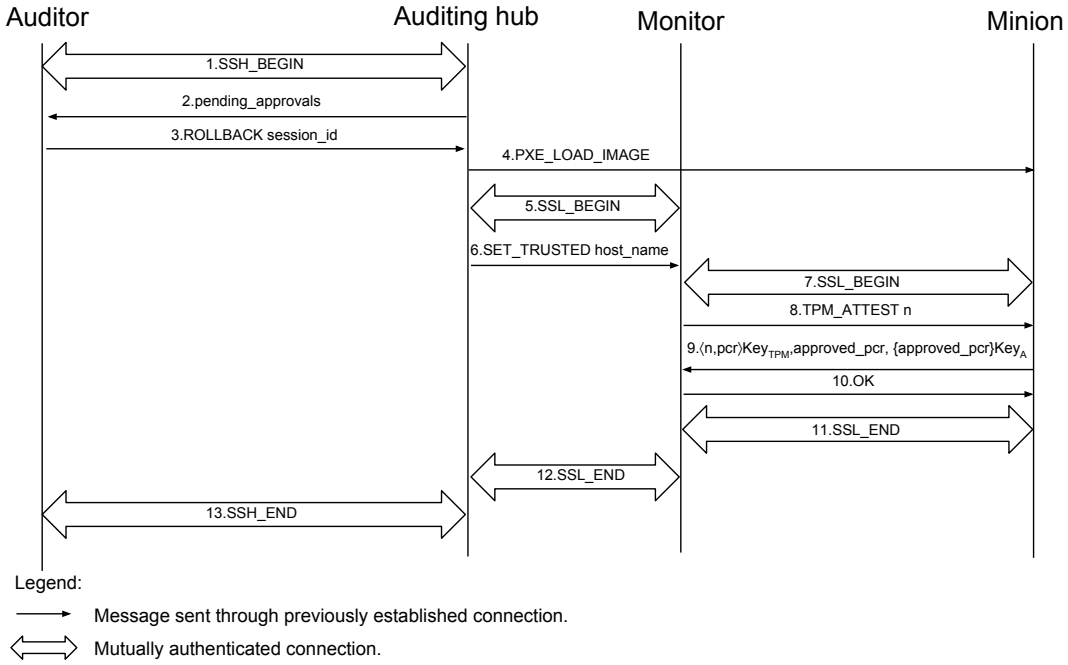

Figure 4.6: Aborted Management Session

respectively.

```
Nov 17, 2015 11:14:05 AM
ALL: Admin->Host:ls

Nov 17, 2015 11:14:06 AM
ALL: Host->Admin:
File1
File2
File3
Folder1

Nov 17, 2015 11:15:05 AM
ALL: Admin->Host:cat File1

Nov 17, 2015 11:15:06 AM
ALL: Host->Admin:
Hello world!

Nov 17, 2015 11:15:10 AM
ALL: Admin->Host:exit
```

Figure 4.7: Session Log

After checking the logs, the auditor may choose to commit or rollback changes. A commit requires the monitors to be notified that the minion may be put on the list of verified nodes, i.e., the minion may receive and run applications (SET_TRUSTED host_name). The monitors,

in turn, before considering the minion verified, attest the latter. If the attestation process is successful, the minion is considered verified. This extra attestation prevents malicious auditors from setting the minions to verified state when they are insecure. If, on the other hand, the auditor chooses not to commit the changes, the auditing hub performs a rollback of the minion's configurations using a default and approved base image (PXE_LOAD_IMAGE). Then, the process is similar to the commit where, the monitor is notified to set the minion as verified, which triggers an attestation to assess the security of the minion. There should be, also, multiple auditors performing the log verification and approval/rejection to stop malicious/neglecting auditors from undermining the security of developers' workloads (e.g. by accepting malicious actions from operational administrators or rejection benign ones).

4.5.4 Developers' Protocols

In order to deploy their applications or interact with their resources, developers contact a monitor. Figure 4.8 depicts an application deployment on P-Cop.

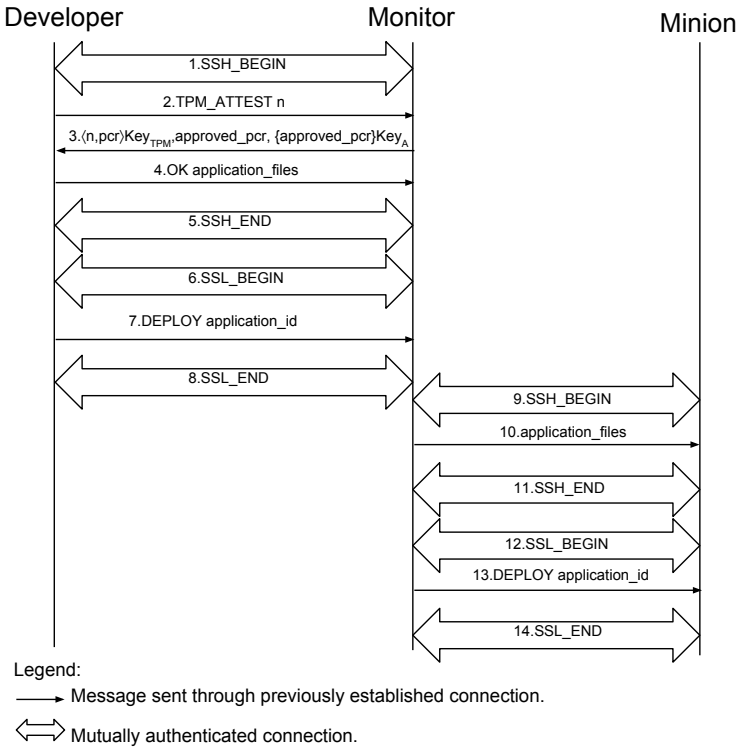

Figure 4.8: Application Deployment

Before sending requests, developers perform remote attestation of monitors configurations. As in the case of the administrators' attestation, the monitor sends to the developer its configuration and developer's nonce (encrypted using the TPM private key), as well as an approved platform configuration and due auditor's signature. The verification process is once more:

1. Decrypts $\langle n, pcr \rangle Key_{TPM}$
2. Verifies the signature of the auditor.

Once the verification is complete, developers' may send the requests to monitors. Assuming an application deployment, the application files are sent to the monitor through SSH (e.g. using git). Then, through the previously established SSL connection, the developer notifies the monitor to deploy his application. The monitor performs a similar procedure to deploy the applications on minion(s).

4.5.5 Protecting Cloud Support Nodes and Administration Logs

While our work focuses mostly on operational administrators' actions on nodes running developers' workloads (i.e. minions), those administrators are still responsible for managing the rest of the cluster (i.e. monitors and auditing hubs). As previously referred, these nodes run specific software (e.g. PaaS support software on monitors and logging system on auditing hubs) while minions run more heterogenous and complex applications/software stacks (e.g. Apache web server with Php application side-by-side with Tomcat with Java servlets). Minions may be more susceptible to failure or erroneous behaviours since Docker runs potentially thousands of application containers which take relatively-high physical space and computing resources. As such minions should require root access with more frequencey (e.g. to restart Docker daemons, debug misbehaving hardware).

Taking these arguments into account, we envision that monitors and auditing hubs may be protected using mandatory access control mechanisms such as SELinux to limit the resources and operations available to operational administrators. The same principle should apply to limit the operations that auditors may perform on auditing hubs. Since the former require SSH access to auditing hubs to run the logging system interface, SELinux could be used to limit the operations performable by auditors to running the interface. Logs could be protected by

allowing the interface process to access the logs with limited privileges as well (e.g. read to avoid a flaw on the interface to be exploited and used to erade or tamper with logs). If root access is required by operational administrators on monitors or auditing hubs, an approach similar to P-Cop's could be employed: intercepting and logging administrative sessions, notion of verified and unverified states, as well as remote attestation. Yet, protecting root access on these nodes is out-of-scope of our work, being left for future work.

4.6 Summary

In this chapter, we gave an in-depth description of P-Cop's architecture, features and protocols. We propose the segregation of the administrative staff in two non-collusive teams: operational administrators (manage clouds nodes remotely) and auditors (responsible for auditing the formers' actions). P-Cop imposes no restrictions on operational administrators' privileges on nodes running cloud customers' applications (minions) by allowing the former to remotely access minions with two conditions: customers' applications are protected (e.g. erased) and the remote sessions are intercepted logged by a distributed logging system (auditing hubs). In order to avoid a high impact related to the erasing of customers' applications and instantiation of replacement ones, P-Cop offers two modes: access with mandatory access control enforced and access with superuser privileges. Both modes are logged but, while the former protects customers workloads using kernel technologies such as SELinux, the latter deletes customers' data and allows unlimited control over the node. Nodes not running applications are protected using mandatory access control with rules specified by both operational administrators and auditors.

We proposed the notion of operating modes for minions: verified (running applications) and unverified (previously accessed for maintenance). The latter requires auditors to check session logs and decide whether they represent malicious administrative sessions or benign ones. Auditors are also responsible, together with operational administrators, for attesting cloud support nodes (e.g. monitor, auditing hub) and provide those with signed configurations (which are approved by both teams) which are supplied to entities accessing those nodes (i.e. developers and operational administrators, respectively) for further verification against TPM quotes issued by the nodes. Monitors are responsible for attesting minions when they register with the former, using signed configurations sent to them by auditors and operational administrators.

5 Implementation

In this chapter we describe P-Cop's prototype implementation. First, we describe the PaaS base cluster and then, we proceed to describe P-Cop's security mechanisms. All P-Cop software components (monitor, minion's node guard, auditing hub and, developers', auditors' and operational administrators' software) were developed using Java 8 with Java extensions for SSL. SSL credentials are generated using OpenSSL 1.0.1f and stored using Java keystores. Developers' applications are deployed using Docker 1.9 running in Linux CentOS 7.1.1503 with kernel 3.10.0-229.4.2.el7.x86_64. Interactions with the TPM to generate quotes are performed using Trousers 0.3.7. Monitors, minions and auditing hubs run OpenSSH 6.6.1 servers.

5.1 PaaS Base Cluster and Developer-side Software

We developed a basic PaaS software that allows developers to deploy and delete applications, by interacting with the monitor which in turn interacts with the minions. We then developed the auditing hub's software and implemented P-Cop's security protocols on top of the basic PaaS software. P-Cop's software for monitors, minions and auditing hubs is multithreaded and defines dedicated threads (translated into dedicated host ports) to handle requests from other nodes or actors. Threads dedicated to process requests are called requests handlers. We now describe the software components and interactions for the base components: monitors, minions, as well as developers' client interface.

5.1.1 Monitors

Figure 5.1 depicts the monitor node structure of our prototype. Upon boot, Monitor (Java application entrypoint) spawns multiple threads called requests handler. This threads handle requests from auditors, developers, auditing hubs and minions and perform the following operations, respectively:

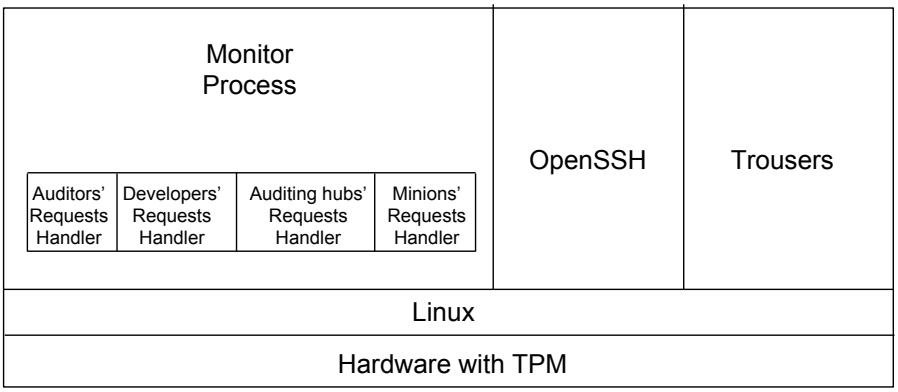

Figure 5.1: Monitor Structure of P-Cop's Prototype

1. Auditors' requests handler: Process attestation requests from auditors. Responsible for receiving and storing approved configurations for minions and monitors themselves. In our prototype, this handler is responsible for the bootstrap attestation process and we limited the attesting actors to auditors, leaving operational administrators for future work.

2. Developers' requests handler: Receive requests from developers to deploy or delete applications on/from minions, respectively. After an SSL connection is established with this handler, a TPM attestation on the monitor is performed, before any further data is exchanged. This handler sends a TPM quote, together with a plain and approved platform configuration, accompanied by the latter signed by the auditor. The choice of which minion hosts an application is based on a Java's java.util.Random to produce a certain degree of load balancing.

3. Auditing hubs' requests handler: Used by auditing hubs to modify the state of minions: verified to unverified and vice-versa. Setting a minion to unverified triggers the spawning of replacement instances (running on the minion) on verified minions. Setting the node to verified triggers a remote attestation on the minion, prior to making it verified.

4. Minions' requests handler: Processes registration requests from minions. Before adding the minion to an internal list of verified minions, the former is attested.

The monitor runs an OpenSSH server used by developers to upload their applications' files, using Linux scp command-line tool, to their home directories. Then, a request is sent to a developers' requests handler to fetch the application files (based on the application folder name,

and local path) and send them to one or more minions (according to the required number of instances), also using scp.

5.1.2 Minions

Figure 5.2 depicts the minion's structure of our prototype. When compared to monitors, minions run another software component that allows developers' applications to run in containers (Docker). Minions main entrypoint called Node Guard is responsible for registering the minion on a master, which requires an attestation (monitor attests minion) before the former becomes verified and suited to receive and run developers' applications. After a successful attestation, the node guard spawns two handlers: monitors' and auditing hubs' requests handlers with the following functions, respectively:

1. Monitors' Requests Handler: Responsible for receiving requests from monitors that comprise: deployment and deletion of applications, as well as TPM attestations. Application deletion calls an external bash script with the container id as argument.

2. Auditing hub' Requests Handler: Auditing hubs use this handler to force the minion to delete all security-sensitive data (all Docker containers on the prototype). On the prototype, such operation calls a bash script (from the Java software) that calls Docker primitives to remove all containers (either active or inactive).

OpenSSH is used to receive application's files from monitors and to receive administrative session requests from auditing hubs (and started by operational administrators). In the first case, the files are copied from a monitor using scp and then, the monitors' handler is notified to fetch the files (from local storage) and schedule the execution of a Docker container with those files. These interactions with Docker are performed directly from the Java application.

5.2 P-Cop Security Features

5.2.1 SSL and Java keystores

We did not implement a signing service. Instead, we generated a root certificate which belongs to the monitor and then we used that certificate to sign the certificates for all other

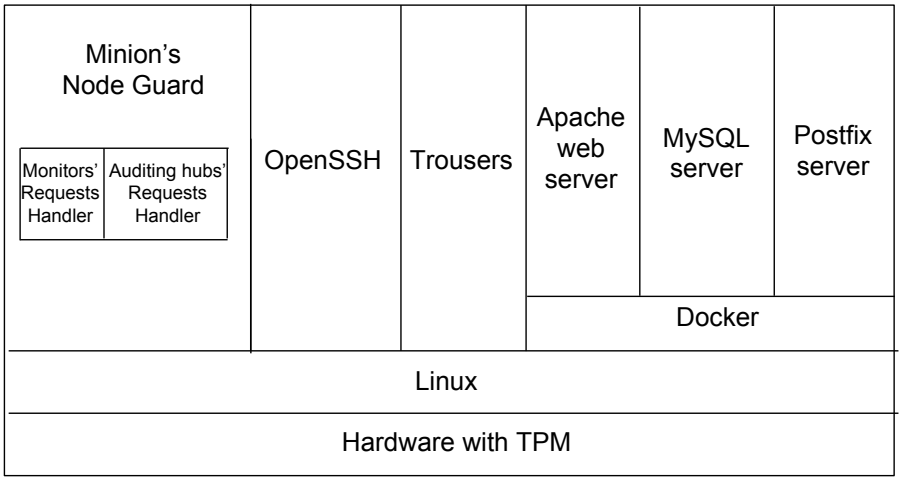

Figure 5.2: Minion Structure of P-Cop's Prototype

actors and nodes. The certificates are generated and signed using OpenSSL and then imported to Java keystores using keytool command-line tool. Besides having their own credentials in stores (persisted on their local hard drives), all nodes and actors have trust stores used to enforce mutual authentication and bootstrap java extensions SSL contexts on our prototype's software. As an example, a developer contacts a monitor to deploy an application. Both developer and monitor have their credential stores (with their certificates) and keystores with the monitor and developer certificates, respectively. On our prototype, these stores are generated using bash scripts and uploaded directly to the nodes or actors directories. Generated credentials use RSA-2048 bits keys.

5.2.2 Auditing Hub

Figure 5.3 depicts the auditing hubs' structure of our prototype. Auditing hubs may receive requests from auditors and operational administrators handled by two dedicated handlers:

1. Auditors' Requests Handler: Processes attestation requests upon bootsrap. If the platform runs approved software, the auditor sends an approved configuration and its signed version.

2. Operational administrators' Requests Handler: Receives management requests from operational administrators. This handler is executed on a local interface, and is accessed through OpenSSH's tunneling capabilities as will be described shortly. This handler is also

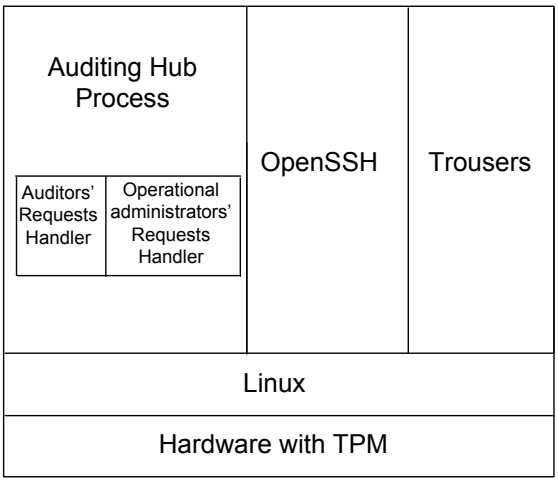

Figure 5.3: Auditing Hub Structure of P-Cop's Prototype

responsible for notifying both monitors and minions to change states (verified/unverified) and delete security-sensitive data, respectively. Operational administrators attest auditing hubs with these handlers.

Figure 5.4 depicts the steps taken to establish a logged administrative session on our prototype. P-Cop's prototype takes advantage of OpenSSH traffic tunneling capabilities to implement the logging system. Operational administrators access minions through a client application which expects, as arguments, the auditing hub and minion hostnames, as well as their credentials. After that, the application creates a local SSH proxy and connects to it. Data sent to the proxy is passed through a tunnel to the auditing hub. The auditing hub, upon receiving an SSH connection from the proxy, begins another connection to a local server (auditing hub main application) that is responsible for starting operational administrators' requests handlers. These handlers start the logging system (Java logging classes), spawn a SSH session on a process and attaches to its input and output streams. From then on, the handler logs all data crossing the session and redirects it back and forth across the latter. Our implementation can only log data to outputs of request response commands (e.g. ls, cat), not allowing more interactive commands such as (nano, vim) since they are more complex to log. Logs are files named and structured as explained on the architecture section. These logs are stored on a local folder.

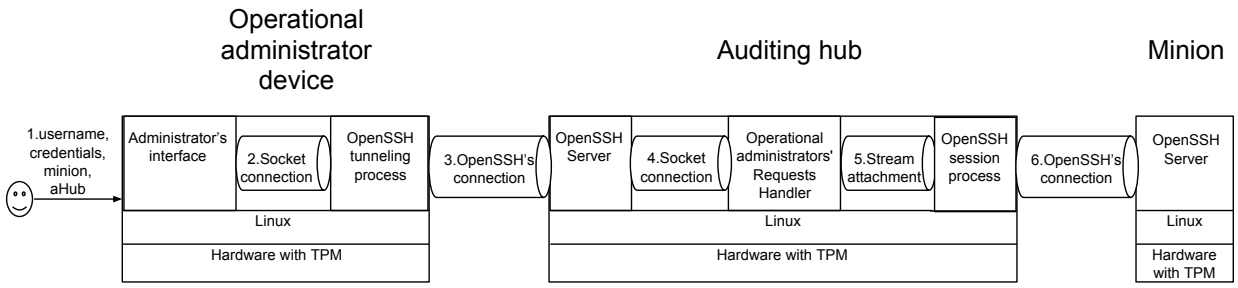

Figure 5.4: Management Session on Prototype

5.2.3 Auditors and the Auditing System

Auditors have two purposes on P-Cop: audit logs and attest key nodes (i.e. monitors and auditing hubs). The second purpose will be explained on the next section. The first purpose requires auditors to access auditing hubs through SSH and running an auditing software client. This client allows them to list and read logs, and commit/rollback operations. Commit leads to the establishment of an SSL connection with the monitor to notify it that a minion may be set to verified mode. As explained before, the monitor attests the minion and adds it to the internal list of verified nodes if the former displays and approved configuration. On our prototype, the rollback operation is not implemented but it could be, by deploying a stock OS image and applying the administrative operations that would not constitute security threats. We leave this topic for future work.

5.2.4 Remote Attestation

In order to assess the level of security of cloud nodes: monitors, auditing hubs and minions, P-Cop employs remote attestation mechanisms. Auditors, developers and administrators have client software which simulates remote attestation on the previously referred nodes. In terms of implementation, the TPM quote is a hard-coded (on the source code) string which represents the bytes of a well-known nonce n, concatenated with a well-known PCR value, encrypted with a TPM's private key. TPM keys are also hard-coded representations of an RSA-2048 bits key pair. Trousers is used on our prototype to measure the times taken to generate a quote.

The first actors employing remote attestation are the auditors using their client software interfaces. Auditors attest minions and auditing hubs by sending the well-known nonce n and

by verifying that the static quote is the nonce n concatenated with the well-known pcr value, encrypted using the TPM private key (quote is decrypted using the hard-coded TPM public key). Then, the auditor signs the well-known pcr and sends both plain and signed versions of it to the auditing hubs and monitors so they may be verified and verify other nodes later (monitors verify minions upon registration). If the attestation process fails, the auditing hubs and/or monitors are ordered to exit the application.

Later, developers and operational administrators may assess the security of monitors and auditing hubs, respectively, since those nodes supply a TPM quote, together with a plain and signed PCR content. Since developers and operational administrators have the public credentials of the auditors on their keystores, they may perform this verification.

5.3 Summary

In this section, we described what technologies we used to implement the P-Cop's prototype. We developed a base PaaS software with limited functionality: deploy one or more instances and delete all instances of an application, using Docker. Then, on top of it, we built P-Cop's security features which encompass: the logging system (auditing hubs), mutually authenticated SSL connections, distributed attestation, as well as deletion and instantiation of applications upon administrative sessions' requests. In terms of implementation, monitors, minions and auditing hubs use multithreading to receive and process requests, and rely on SSH to transfer files and create communication tunnels that are intercepted and logged by the auditing hubs. Our implementation of the purging process (deletion of security-critical data on minions) deletes application files and uses Docker to erase containers and their images. Since our focus is to provide unlimited access to minion nodes, we did not implement the mandatory access control rules to protect, both support nodes (e.g. monitors and auditing hubs) and minions, leaving that for future work.

6 Evaluation

In this chapter, we describe the procedure used to evaluate the developed prototype. We start by describing the testbed and the applications used to test the prototype. We then describe the impacts on deployment, management and auditing tasks, followed by the impacts caused by the logging system. Then, we discuss P-Cop's resilience to security threats. We finish the chapter with discussions of results, impacts and viability of our solution.

6.1 Testbed and Test Applications

We evaluated our prototype of P-Cop using Emulab nodes. Our cluster consisted of ten minions, one monitor and one auditing hub. All nodes had two Intel(R) Xeon(TM) 3.00GHz, 2GB of RAM and an ethernet interface of 100Mbps. The base images used were Linux CentOS 7.1.1503 with kernel 3.10.0-229.4.2.el7.x86_64.

In order to simulate the connection speeds of a domestic user (for developers), we ran the client software for developers outside Emulab's premises from a desktop running Ubuntu 14.04.3 LTS 3.16.0/3/0/generic, with 8GB of RAM, an Intel(R) Core i7-4790K 4.0GHz and an ethernet interface of 100Mbps. The home network had a download rate of 55.7 Mbps and and upload rate of 5.6 Mbps. The client software for administrators and auditors was ran inside Emulab's premises from a node running Ubuntu 12.04.5LTS 3.8.0-33-generic, with 1GB of RAM, an Intel(R) Core 2 2.40GHz and an ethernet interface of 100Mbps. Times were measured using the Linux time command-line tool. As previously referred, P-Cop brings most of the security verifications to nodes, auditors, administrators and developers, leaving the applications' end-users with a typical HTTPS certificate verification. As such, we focus on measuring the impact of our solution on the former entities.

We used two PHP applications of different sizes and complexities, running on an official Apache Web server (version 2.4) container, as proofs-of-concept: a simple hello world static web

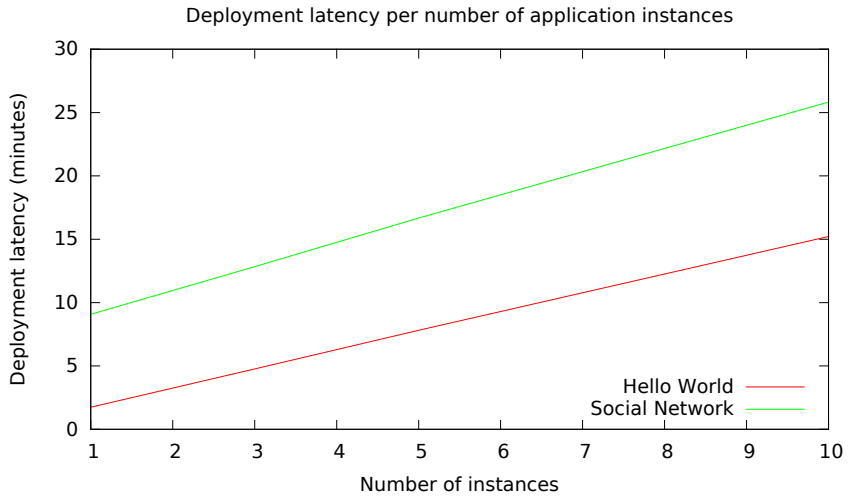

Figure 6.1: Deployment Latency for various Instances of Hello World and Social Network

page (1.5Kb) and a social network (11Mb) (10). The applications were chosen as extreme cases: hello world is a simple static page and social network a complex and heavy application. We wanted to evaluate the impact of our solution on deployments and administrative tasks.

6.2 Performance Evaluation

In this section, we discuss the time overheads imposed by P-Cop on deployment, administration and auditing (attestation and mode transitioning). We finish the section with a discussion of the results.

6.2.1 Impact on Deployment

We wanted to assess the impact of our solution on the deployment process (Figure 4.8), according to the application sizes and instances. As such, we used the hello world and social network applications as the deployed applications and changed the number of instances. Figure 6.1 depicts the evolution of the deployment times for both applications according to the number of instances: 1, 3, 5, 10. Both hello world and social network display times that grow linearly which is a consequence of the synchronous nature of the deployment (monitor waits for a minion to deploy the container before deploying it on another minion). The time leap between the two cases is a consequence of the different sizes of the applications: the transfer times between

Docker base image	Pull time (minutes:seconds)
Apache Web Server	0:19
Nginx	0:33
Tomcat	0:49
Wordpress	1:29
Nodejs	1:32
Django	0:51

Table 6.1: Docker Pull Times, for Typical Web Technologies, from the Public Docker Repository

the client and the monitor are much slower when compared to the ones between the monitor and the minions (developer is outside the cluster). The Docker daemon and the process of creating a container represent most of the performance impact as well, as we will discuss next. We measured the times taken with PHP applications but Docker allows other languages and supporting technologies such as: Nginx, Tomcat, Wordpress, Nodejs and Django. Table 6.1 shows the times taken (in minutes) to pull Docker base images, with typical web technologies uses in PaaS environments, from the public Docker repository (DockerHub). These pulls wre performed on one of the minion nodes. The penalty imposed by the pull process may be removed if the cluster minions already have the necessary base images which is acceptable since most cloud providers offer a limited set of supported technologies. In our prototype, no base containers where on the nodes, which explains the added performance penalty as well. In order to evaluate the performance penalty introduced by the Docker daemon on deployments, we measured the times taken to deploy each test application, taking into account the following cases:

1. Deploy a single hello world instance
2. Deploy two hello world instances with different names and different index.php files (but with same overall size)
3. Deploy a single social network instance
4. Deploy two social network instances with different names and different index.php files (but with same overall size)
5. Deploy one hello world instance followed by a social network instance.

Cases 1 and 3 are used to measure the impact of having no base container images cached, which requires the images to be pulled (we called it cold deployment). Cases 2 and 4 show the

Test case	Hello world		Social network	
	Cold deployment	**Cached deployment**	**Cold deployment**	**Cached deployment**
Different names + different files	01:27	00:51	01:37	00:59
Social network after Hello world	00:58			

Table 6.2: Container Deployment Times for Different Cases (minutes:seconds)

added performance of having a set of base containers on the node as a cache (cached deployment). Case 5 simulates the case where totally different applications, with the same base container, are deployed on the same node. Table 6.2 shows the measured times in minutes and allows us to conclude that:

1. The size of the application affects the deployment times.
2. Having the base container (e.g. Apache Web Server) on the node improves the deployment times. Social network took less time to be deployed when hello world was deployed first.

Overall, we may attribute the deployment latency to the following sources:

1. TPM quote
2. File transfers between developer and monitor, and between monitor and minions.
3. Deployment of applications on minions
4. P-Cop operations: SSL connections, exchanged messages, quote verification

As an example, for five instances of hello world and social network, Table 6.3 shows the overall deployment times, with all previously referred components, discriminated. The majority of the performance impacts come from file transfers and application deployments, i.e., from scp and the Docker daemon (pulling base image, building container, copying application files to the container and starting it), leaving less than half of a minute for P-Cop operations.

Application	TPM quote	File transfers	Application deployment	P-Cop operations	Total
Hello world	00:03	00:21	07:15	00:10	07:49
Social network	00:03	08:15	08:05	00:18	16:41

Table 6.3: Discrimination of Deployment Latencies for Five Instances of Hello World and Social Network (minutes:seconds)

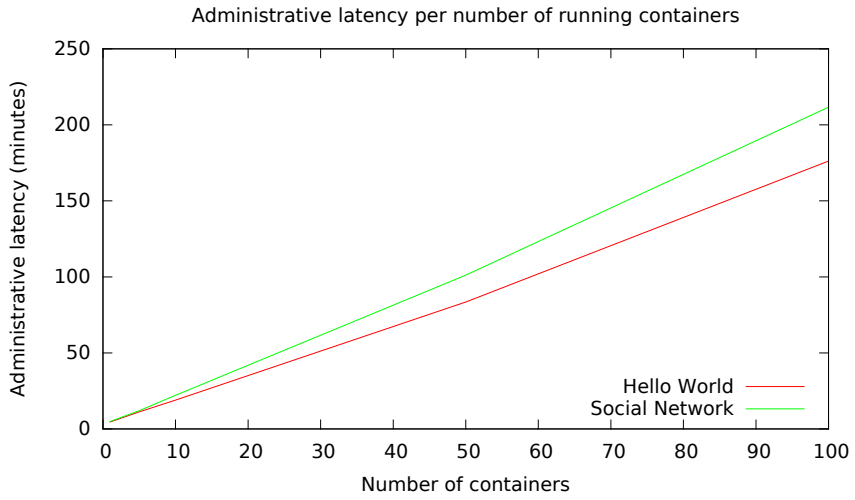

Figure 6.2: Administrative Latency for n Instances of Hello World and Social Network Running

6.2.2 Impact on Administration

In order to evaluate the impact of P-Cop on administration, we deployed hello world and social network multiple times (with different names and different index.php files, but with the same size) on a given minion and tried to begin an administrative session later on it. Such approach simulates the possibility of having multiple applications with hello world's and social network's complexity running (low vs. high workload). The applications are then migrated to another node. In order to simulate the protection of the node against snooping and tampering, we used the Docker daemon to remove all Docker images and containers on the node (including the base images), and deleted all the application files sent by the monitor and used to build the containers. Figure 6.2 shows the time taken from the moment that the operational administrator requests access until he can access the node.

Since the management procedure involves a step where the monitor deploys applications on minions, following a synchronous process similar to the deployment, the times grow in an approximately linear manner. As opposed to the graph on Figure 6.1, the latency leap between

Application	Base image		Application container image		Application container			Total (m:s)	Total (5 apps)
	Size (Mb)	Remove (m:s)	Size (Mb)	Remove (m:s)	Size (Mb)	Stop (m:s)	Remove (m:s)		
Hello world	193.4	02:13	193.4	00:20	193.4	00:0.5	00:19	02:53	05:54
Social network	193.4	02:13	207.7	00:20	207.7	00:0.6	00:19	02:53	05:39

Table 6.4: Removal Times for Docker Containers and Images

the hello world and social network applications is in the order of seconds, instead of minutes. This happens, mostly, because there is no file transfer between the developer and the monitor, which took a relatively big chunk of the total deployment time.

During our tests, we noticed that most of the total times measured were spent by Docker deploying and deleting containers. Since the deployment times were already analyzed on the previous section, we now address the purging (deletion of Docker data and application files) impact. Figure 6.2 encompass the times taken to:

1. Stop containers
2. Delete containers' images
3. Delete base images
4. Delete applications' folders on minions: 0.008 and 1.14 seconds for hello world and social network, respectively.

Most of the time taken by the purging process comes from the Docker daemon. As such, we measured the time taken by the daemon to delete a base image, and the application containers and containers' images. In order to delete a container and its image, it is necessary to stop the container first, then the container may be deleted, followed by its image. Table 6.4 shows the measured times for all the steps.

The sizes of application containers and their respective images, for the same application are the same. This happens because, the container hasn't add any files or data to the base container (i.e. no files created, edited, removed), leaving the overall size unchanged. Tables 6.2 and 6.4 tell us that, if we keep the base containers (which are not confidential and may be inspected, but not tampered with), we boost the performance of the deployment and cleaning processes. Also, the application sizes do not affect the times significantly, which means that, purging a

Applications	TPM quote	File transfers	Application deployment	Minion purge	P-Cop operations	Total
Hello world	00:03	00:15	05:00	05:54	00:13	11:25
Social network	00:03	00:40	05:45	05:45	00:14	12:27

Table 6.5: Discrimination of Management Requests' Latencies for 5 Applications of Hello World and Social Network

node running hello world or social network (i.e. small or complex applications) takes about the same time.

Overall, we may attribute the administrative access latency to the following sources:

1. TPM quote
2. File transfers between monitor and minions.
3. Deployment of applications on minions
4. Removal of application files and containers from the minion to be managed
5. P-Cop operations: SSL connections, exchanged messages, quote verification, process creation (i.e. local proxies)

Taking the case where there are five instances of hello world or social network, Table 6.5 discriminates all the times taken by each of the previously referred sources. The application deployment and minion purge parcels were obtained by deploying and deleting applications and Docker data, respectively. Also, the purging times take into account the times taken to delete the application files.

6.2.3 Auditors' Attestation and Mode Transitioning

Auditors's tasks encompass attesting monitors and auditing hubs before they start serving requests, and evaluate the logs of a minion's administrative session. We evaluated our prototype by attesting a monitor and a auditing hub. Table 6.6 shows the measured times. As expected, the monitor takes slightly more time to be attested since signatures for both monitor and minion must be generated and sent.

In order to evaluate the second task, we measured the time taken by the auditing software to approve one log. Since the process of evaluating the log takes variable time (i.e. different

Attested node	Attestation times (m:s)
Monitor	00:05
Auditing hub	00:04
Total	00:09

Table 6.6: Attestation Times

auditors may apply different auditing techniques), it is not taken into account. The time taken to complete our test was 3.98 seconds. The next section describes our quantitative and qualitative evaluations of the logging system.

6.2.4 Discussion of Results

Both deployment and administration are heavily impacted by the Docker daemon since deployment and removal of containers is slow. Running applications without Docker (e.g. directly on Tomcat, Apache web server) may improve, largely, the performance of our solution since, even though Docker containers are relatively light, they are heavy when compared to simple web applications running on software supporting web applications. In our prototype, we chose to use Docker as a means to run applications and simulate a PaaS environment. P-Cop does not require Docker to work and may be extended to deploy applications without its support. Having a limited number of languages and supporting software boosts performance since, with Docker, if the base containers are not present on the node, they must be downloaded.

P-Cop only provides superuser privileges to operational administrators. Yet, it could be extended to provide a more restrictive level of privilege that would protect PaaS applications and security-critical components on minions. In cases where non-root privileges suffice, there is no need for Docker containers and applications to be deleted. Simple access control mechanisms (e.g. mandatory access control) should suffice to limit the extent of the damage inflicted by a malicious administrators. His actions would be logged, nevertheless, but the purging and migration overheads would disappear, improving our solution. This approach introduces less impact but requires the auditors and operational administrators to design and implement rules that reduce the attack surface of the minions.

6.3 Logging System Evaluation

Auditing hubs mediate the administrative sessions and log all traffic crossing it to text files with a structure similar to the one on Figure 4.7. In order to evaluate our logging system, we simulated seven test cases of typical actions operational administrators would perform on minions. We also simulate a case where an administrator attempts to bypass the logging system. We then measure the logs size: input, output and overheads. We now describe the test cases and the logs' measurement.

6.3.1 Test Cases

Misbehaving Docker daemon: In this case, we evaluate the case where a Docker daemon show abnormal behaviours such as: not running applications, not allowing external accesses,etc. Since all minions would run the same Docker version, as well as the same approved configurations, a misbehaving Docker daemon may be a consequence of overload or some degree of conflict with a PaaS application. The procedure evaluated here would be a simple restart on the Docker daemon (i.e. service Docker restart). Table 6.7 shows the measurements for this case.

Source	**Size (bytes)**
Input	24
Output	54
Total	1536

Table 6.7: Log Sizes for the Docker Restart Case

Zero day requiring urgent updates: A zero day is being used to exploit a Linux kernel flaw on the network stack, which allows an attacker to compromise a machine through a specially crafted UDP packet. Linux community urges administrators to update their Linux hosts (e.g. yum -y update). Table 6.8 shows the measurements for this case.

Source	Size (bytes)
Input	24
Output	15564
Total	16384

Table 6.8: Log Sizes for the System Update Study

Abnormal traffic leaving a minion: The cloud administrators find, through a national CERT, that one of their nodes is taking part on a sophisticated botnet. Since the CERT is not sure if the infected machine is spoofing the IP address to incriminate the cloud providers, administrators need to verify the threat. The CERT informs the administrators team of the C&C address and port and the latter team decides to run tcpdump on the affected machine, with the given data (C&C address and port) and verify is there is a threat. Table 6.9 shows the measurements for this case.

Source	Size (bytes)
Input	61
Output	7168
Total	7680

Table 6.9: Log Sizes for the Compromised Node Case

Minion not able to receive more applications: The monitor is reporting that one of the minions refuses to receive more application files due to no space left on the node. The administrator must, therefore, understand the source of such issue. This would require the usage of commands such as df to check the used space, as well as to find the most storage consuming folders/files (e.g. du -ah / — sort -n -r — head -n 100 to list the 100 biggest files or folders on the node). Table 6.10 shows the measurements for this case.

Source	Size (bytes)
Input	37
Output	4812
Total	5529

Table 6.10: Log Sizes for the Full Disk Case

Minion with connectivity issues: One of the minions becomes inaccessible from time to time, either through SSH or HTTP/HTTPS (used to access applications). Since the minion is one of the oldest, administrators need to troubleshoot a potentially faulty NIC. Therefore, they read /var/log/messages and /var/log/dmesg using a tool such as cat. Table 6.11 shows the measurements for this case.

Source	Size (bytes)
Input	44
Output	108748
Total	111923

Table 6.11: Log Sizes for the Misbehaving NIC Case

Auditing of external attacks against the cloud: A misconfiguration on one of the external firewalls allowed attackers to access all ports on minion nodes. This, combined with a zero day on minions' firewall, allowed attackers to bruteforce the SSH servers on those nodes. Administrators need to understand if the attackers managed to enter the minions. To this end, they use lastlog to check logged users. Table 6.12 shows the measurements for this case.

Source	Size (bytes)
Input	8
Output	2457
Total	2969

Table 6.12: Log Sizes for the SSH Bruteforcing Case

Operational administrator attempt to bypass logging system: The cloud provider decides to hire a new administrator which learns of the logging system in place. In order to evade the system, the administrator decides to log in one of the minions, disable the firewall, add his public SSH key to the file of approved keys and log in directly. What happens then is not logged and therefore, anything may have happened. Table 6.13 shows the measurements for this case.

Source	**Size (bytes)**
Input	504
Output	54
Total	1126

Table 6.13: Log Sizes for the Malicious Administrator Case

6.3.2 Results' Discussion

On the previous cases presented, we attempted to show how the logging system behaves, in terms of the space taken by the logging system, by varying the amount of output resulting from operational administrators commands. In most cases, the outputs represent most of the space taken by logs. Commands used to update systems, check logs and dump are, typically, very verbose. The last case has a bigger input since the administrator writes his RSA key to the file of authorized public keys and such keys are large (e.g. 1024 bits). The space remaining, input and output aside, represents timestamping and some other fields, as depicted on Figure **??**, the log for the first test case.

For administrative sessions where the the inputs and outputs are small, the logging introduces relatively high overhead. Since we used the Java logging system, the logs have some fields (e.g. auditinghub, AdminSessionRequestHandler and launchManagementSession) which represent the package, class and method where the logging system was called. In a real case, these fields would not be on the log, leaving all the overhead associated with timestamping and flow identification (i.e. Host->Admin and Admin->Host), which are relatively small.

Overall, the size of the logs on our test cases, proved to be relatively small, even when

command leading to verbose outputs where used (e.g. tcpdump, yum update). Also, since these logs may be deleted later after approval/rejection, minions do not require constant accesses, and the logging system is distributed, we envision that our system does not bring storage concerns.

6.4 P-Cop's Security Analysis

P-Cop's threat model is focused, mostly on software-based attacks, assuming the hardware is properly secured. P-Cop only protects communications using mutually authenticated SSL and SSH, leaving hardware attacks for future work.

As for software-based attacks, we provide security analysis of our system in each of the four of P-Cop's actors goes astray:

- Developers
- Applications' end-users
- Auditors
- Operational administrators

We focused, mostly, on operational administrators since they may directly affect developers' and end-user's workloads and data, respectively. Yet, before discussing operational administrators' threats, we overview potential threats represented by other actors and discuss P-Cop's solutions and limitations.

Developers may attempt to compromise end-users' devices by deploying malicious applications that actively attempt to exploit, for instance, vulnerable browsers. They may also steal and disclose end-users' data managed by their applications. Both issues are not addressed by P-Cop that employs HTTPS certificates, issued by well-known certification authorities. End-users must, therefore, trust that the certification authorities scrutinized developers' personal data to prove that the latter are, in fact, trustworthy. The ubiquity of web applications leads to an ubiquity of these issues throughout the Internet. Therefore, awareness and, updated anti-malware software and operating systems are the way to limit these issues.

End-users may attempt to exploit developers' applications to attack other applications or even the underlying operating system. P-Cop makes no assumptions about the correctness of

the application code. Yet, even if malicious end-user attempts to compromise the application and the webserver, the latter may run as a non-privileged user. As such, the attacker may, at most, compromise files owned by the web server. This is an issue in the application's domain but not for other containers and applications, as well as the underlying operating system since Docker mechanisms (i.e. namespaces, Linux containers) prevent container applications from accessing the surrounding environment.

Auditors are the main point of trust in P-Cop. Yet, we may assume the case where some auditors become rogue. Auditors are responsible for attesting monitors and auditing hubs before they start serving requests. They are also responsible for verifying logs and approve or reject changes. In the first case, auditors may attest the nodes and approve (sign malicious configurations matching the ones on a malicious node). Developers and operational administrators may therefore be tricked into accepting monitors and auditing hubs, respectively, as trusted. P-Cop solves this issue by having the monitors and auditing hubs receive multiple signatures from multiple auditors before starting to serve requests. As the number of required signatures grows, so does the time taken by developers and operational administrators to assess the level of trustworthiness of monitors and auditing hubs, respectively. On the second case, auditors may delete, create or tamper with session logs since the logging system interface is ran directly on the auditing hub (i.e. requires SSH access to the latter). They may also accept or reject logs when the proper action would be otherwise. Integrity of logs could be achieved through mandatory access control (e.g. using SELinux) where the logging system interface software would be runnable by every auditor and confined to a set of read-only files (e.g. logs). Malicious accepts and rejects could be avoided using a technique similar to the one presented for malicious attestations: the logs must be approved or rejected by n auditors before committing the operations. Once more, as the level of trustworthiness grows, so does the time taken to change minions' states.

Operational administrators represent the biggest direct threat to developers' computations. Since they may access minions, remotely, with potentially unlimited privileges (e.g. root), they may extract confidential data from applications, delete them, tamper with them or even undermine the overall security of the minions. P-Cop mitigates this by migrating erasing all traces of applications and their respective containers from the minions before operational administrators enter. In order to assess the level of compromising of a minion node, we also employ a logging system to log all data crossing the administrative session, which is later on evaluated by the

auditors. A doable attack on our architecture was exemplified on the last case study, where an operational administrator disables the firewall and adds his key to the list of SSH's authorized keys. This allows the direct establishment of an SSH connection to the minion bypassing the logging system, yet, the actions leading to the bypassing were logged and auditor should, therefore, reject the changes and rollback the node's state. Operational administrators are also responsible for performing bootstrap attestations on monitors and auditing hubs. Since they are the ones deploying the base images on nodes, they may deploy malicious ones and sign their malicious TPM signatures. Since both monitors and auditing hubs must accept multiple signatures from both teams, the developers and operational administrators attesting monitors and auditing hubs, respectively, should decide how many signatures are enough to validate the given configurations. The same applies to monitor and auditing hubs which must be configured with to require a n minimum signatures before starting to serve requests.

Operational administrators are responsible for managing monitors and auditing hubs which may be dangerous if operational administrators become rogue. In order to protect such nodes from abusive administrators, mandatory access control mechanisms such as SELinux could be used to limit the operations and resources accessible by operational administrators. In our work, we do not address this issue in-depth since the decision of which resources are critical/non-critical should be agreed upon by both operational administrators and auditors.

The purging process employed is relatively weak in terms of security guarantees since we delete Docker data through its daemon and remove application folders using simple rm commands. This is a superficial removal since, with root privileges, operational administrators may read raw sectors from the disk. Such case is cumbersome, both for the administrators which needs to check an entire volume for confidential data, and for the logging system that will grow to a point where the node running it cannot take the data being dumped to text files. We assume such case is unlikely to occur. Yet, a possible approach would be to use secure-wipe tools that overwrite the hard drive sectors with zeros or random data. Another approach would be to encrypt the hard drive and use TPM to secure the encryption key.

6.5 Summary

In this section we described evaluation of P-Cop's prototype and discussed its viability. We used Emulab's testbed for testing purposes and started by measuring latencies imposed on: deployment by developers, management by operational administrators, and attestation and mode transitioning (from unverified to verified) by auditors. While the overall times are relatively-high, the impact imposed by P-Cop's features is relatively-low, leaving most of the latencies for file transfers and containers' instantiation/deletion by the Docker daemon. Measured times for deployment and management grow approximately linear according to the number of instances to be deployed and migrated, respectively.

We then evaluated the storage impact of the logging system by describing a set of typical administrative tasks and issues that lead to them. In terms of storage, even the most output verbose sessions (e.g. system updates) leave relatively small logs. Since these logs may be deleted later after approval/rejection and since minions do not require constant accesses, and the logging system is distributed, we envision that our system does not bring storage concerns.

Finally, we discussed P-Cop's resiliency to malicious threats. P-Cop leaves most hardware-based attacks out-of-scope (e.g. bus/RAM probing) as well as DOS attacks, focusing mostly on operational administrators and their tasks on compute nodes (minions). We assume that developers are not malicious, as well as application end-users. Auditors are also more trusted than operational administrators but the former are also required to assess the security of deployed configurations on minions.

7 Conclusions

7.1 Conclusions

PaaS environments provide convenient way for developers to deploy and scale their applications. Such environments are managed by administrators which may require superuser privileges, over nodes supporting developers' workloads or hosting application-related data, in order to perform privileged maintenance tasks. Existing research work fails to address PaaS security issues properly, by offering solutions that are hardly adaptable and add unnecessary complexity with little gains.

This thesis presents P-Cop: an architecture to provide a secure environment built on top of the current PaaS architecture that provides privacy and integrity of data and applications to developers and application end-users, in the face of potentially malicious administrators. The solution combines concepts from research, such as: segregation of administrative domains, access control, logging and trusted computing attestation. We have described the design, implementation and evaluation of our solution.

Our prototype was implemented by building a basic PaaS software to deploy applications in Docker containers, and then, adding P-Cop's security-related components: auditing hubs for logging, attestation and mutually authenticated SSL sessions. Evaluation was performed on the latencies and overheads introduced by P-Cop on: deployment, auditing and administration of computational nodes (minions). Overall, the impact of P-Cop represents a relatively minimal fingerprint in terms of performance penalties and storage overheads, being acceptable for cloud environments.

7.2 Future Work

P-Cop is mostly oriented to logical attacks, leaving hardware-based attacks unattended. Hardware administrators may snoop on/ tamper with nodes data through direct access of storage volumes. They may also probe RAMs or buses and use side-channels to extract information. Storage volumes could be protected with encryption while RAM and bus probing could be mitigates using commodity-hardware mechanisms such as Intel SGX. We leave these issues for future work.

P-Cop also focus on protecting minions, leaving nodes with services such as: databases, queues,etc. Since Docker imposes relatively-high performance penalties, we envision P-Cop to be more flexible to protect customers' workloads with classic PaaS (e.g. Apache Web Server shared by multiple applications).

Monitors and auditing hubs are protected by limiting the set of resources and operations available to operational administrators and auditors. The definition of rules to be applied requires a more in-depth analysis left for future work.

Administrators are also forced to choose the auditing hub that intercepts their connections. A broadcast approach would be more interesting where the administrators supply an address to the administrative interface, and one or more available auditing hub responds and proposes itself as the logger for that session. This could even help balancing the load across hubs. We envision our logging system as a network of P2P auditing hubs that exchange information about ongoing sessions, managed nodes and pending approvals with respective logs. The logging system also lacks redundancy which, in the event of an auditing hub having its storage hardware failing, represents an issue. The interception system used by the logger is relatively-heavy in terms of resources usage since it requires the auditing hub to launch and attach to a process that connects to the minion to be managed. The operational administrator device is also required to run an SSH proxy to create a tunnel to the auditing hub, which in turn connects to the local logging process. In our future work, we would like to implement all the logging capabilities from scratch or by modifying OpenSSH.

In our work and prototype, upon administrative requests, we deleted applications and launched new ones on verified nodes, regardless of ongoing sessions from application end-users. We leave such challenges to be addressed in our future work.

Finally, we would like P-Cop to be able to analyze administration logs and detect malicious patterns, allowing it to decide whether the node should be trusted or not, instead of relying on one or more auditors.

References

[1] AMD Web Page. http://www.amd.com/, 2015. [Online; accessed 27-May-2015].

[2] Apache Stratos Web Page. http://stratos.apache.org/, 2015. [Online; accessed 3-May-2015].

[3] AWS Elastic Beanstalk Web Page. http://aws.amazon.com/pt/elasticbeanstalk/, 2015. [Online; accessed 25-April-2015].

[4] AWS Web Page. http://aws.amazon.com/, 2015. [Online; accessed 25-April-2015].

[5] Azure Web Page. http://azure.microsoft.com/, 2015. [Online; accessed 25-April-2015].

[6] Docker Web Page. https://www.docker.com/, 2015. [Online; accessed 25-April-2015].

[7] Drawbridge Web Page. http://research.microsoft.com/en-us/projects/drawbridge/, 2015. [Online; accessed 27-May-2015].

[8] Google App Engine Web Page. https://appengine.google.com, 2015. [Online; accessed 25-April-2015].

[9] Intel Web Page. http://www.intel.com/content/www/us/en/homepage.html, 2015. [Online; accessed 27-May-2015].

[10] Social Network GitHub Page. https://github.com/opensource-socialnetwork/opensource-socialnetwork, 2015. [Online; accessed 20-January-2016].

[11] TCG Web Page. http://www.trustedcomputinggroup.org/, 2015. [Online; accessed 3-May-2015].

[12] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications From an Untrusted Cloud with Haven. *Proceedings of the 11th Usenix Symposium on Operating Systems Design and Implementation*, 2014.

[13] Sören Bleikertz, Anil Kurmus, Zoltán a. Nagy, and Matthias Schunter. Secure Cloud Maintenance. *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security - ASIACCS '12*, page 83, 2012.

[14] A Brown and J S Chase. Trusted Platform-as-a-Service: A Foundation for Trustworthy Cloud-hosted Applications. *Proceedings of the ACM Conference on Computer and Communications Security*, pages 15–20, 2011.

[15] William R Claycomb and Alex Nicoll. Insider Threats to Cloud Computing Directions for New Research Challenges. *Proceedings of the 36th Computer Software and Applications Conference (COMPSAC)*, 2013.

[16] Paul England, Limin Jia, James Lorch, and Arunesh Sinha. Continuous Tamper-proof Logging using TPM2.0. *Proceedings of the 7th International Conference, TRUST*, 2013:0–31, 2014.

[17] Afshar Ganjali and David Lie. Auditing cloud administrators using information flow tracking. *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, pages 79–84, 2012.

[18] T Garfinkel, Ben Pfaff, and J Chow. Terra: A Virtual Machine-based Platform for Trusted Computing. *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, 2003.

[19] Md Tanzim Khorshed, A. B. M. Shawkat Ali, and Saleh A. Wasimi. Monitoring Insiders Activities in Cloud Computing Using Rule Based Learning. *Proceedings of the 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 757–764, 2011.

[20] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trust visor: Efficient TCB Reduction and Attestation. *Proceedings of the IEEE Symposium on Security and Privacy*, 2009:143–158, 2010.

[21] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 42(4):315, 2008.

[22] Lucky Nkosi, Paul Tarwireyi, and Matthew O Adigun. Insider Threat Detection Model for the Cloud. *Proceedings of Information Security for South Africa*, 11062500001:1–8, 2013.

[23] Emmanuel Owusu, Jorge Guajardo, Jonathan M. Mccune, Jim Newsome, Adrian Perrig,

and Amit Vasudevan. OASIS: On Achieving a Sanctuary for Integrity and Secrecy on Untrusted Platforms. *Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security (CCS'13)*, (i):13–24, 2013.

[24] Vasilis Pappas, Vasileios Kemerlis, Angeliki Zavou, Michalis Polychronakis, and Angelos D. Keromytis. CloudFence: Enabling Users to Audit the Use of their Cloud-Resident Data. 2012.

[25] Jianbao Ren, Yong Qi, Yuehua Dai, Xiaoguang Wang, and Yi Shi. AppSec: A Safe Execution Environment for Security Sensitive Applications. *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments - VEE '15*, pages 187–199, 2015.

[26] Indrajit Roy, S.T.V. Srinath T V Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and Privacy for MapReduce. *Proceedings of the 7th USENIX conference on Networked Systems Design and Implementation*, pages 20–20, 2010.

[27] A. Srivastava S. Butt, H. A. Lagar-Cavilla and V. Ganapathy. Self-service Cloud Computing. *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[28] Ahmad-Reza Sadeghi and Christian Stüble. Property-based Attestation for Computing Platforms. *Proceedings of the 2004 workshop on New security paradigms - NSPW '04*, page 67, 2005.

[29] Nuno Santos, Krishna P Gummadi, and Rodrigo Rodrigues. Towards Trusted Cloud Computing. *Proceedings of the 2009 conference on Hot topics in cloud computing (HOTCLOUD)*, 10(2):3, 2009.

[30] Nuno Santos and Rodrigo Rodrigues. Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services. *Proceedings of the 21st USENIX Security Symposium*, 2012.

[31] Joshua Schiffman, Thomas Moyer, Hayawardh Vijayakumar, Trent Jaeger, and Patrick McDaniel. Seeding Clouds with Trust Anchors. *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop*, pages 43–46, 2010.

[32] Fengzhe Zhang, J Chen, Haibo Chen, and Binyu Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 203–216,

2011.