BUILDING TRUST IN CLOUD COMPUTING -ISOLATION IN CONTAINER BASED VIRTULISATION

By

Ibrahim Alobaidan

A thesis submitted in partial fulfilment of the requirements of Liverpool John Moores University for the postgraduate doctoral degree

November 2019

Academic Thesis: Declaration of Authorship

I am Ibrahim Alobaidan

I declare that this thesis and the work presented in it are my own and has been generated by me as the result of my own original research.

Building trust in cloud computing – isolation in container based virtualisation

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Either none of this work has been published before submission, or parts of this work have been published as: [please list references below]:

[1] I. Alobaidan, M. Mackay, and P. Tso, "Build Trust in the Cloud Computing - Isolation in Container Based Virtualisation," in *2016 9th International Conference on Developments in eSystems Engineering (DeSE)*, 2016, pp. 143-148.

[2] I. Alobaidan, M. Mackay, and N. Shone, "Build Trust in the Cloud Computing - Isolation in Container Based Virtualisation," in The Tenth International Conference on Cloud Computing, GRIDs, and Virtualisation- CLOUD COMPUTING 2019 (IARIA), 2019.

Signed: Ibrahim Alobaidan

Date: 4/12/2019

Publications

[1] Building trust in cloud computing – isolation in container based virtualisation.
2016 9th International Conference on Developments in eSystems Engineering (DeSE)
Year: 2016
IEEE Conferences
Liverpool, UK

[2] Building trust in cloud computing – isolation in container based virtualisation.
The Tenth International Conference on Cloud Computing, GRIDs, and Virtualisation- CLOUD COMPUTING 2019
Year: 2019
IARIA Conference
Venice, Italy

Abstract

Container-based virtualisation has weak isolation compare with traditional VMs. Container-based virtualisation is based on kernel OS. Share kernel OS could increase the possibility of attacks. Therefore, the container-based virtualisation provides weak isolation. The lack of isolation from the host could be increase security threats on the container-based virtualisation. The attacker could gain access to all system in the container-based virtualisation because share the kernel OS. The container is a good idea to isolate the applications. However, container-based virtualisation does not provide isolation for users within containers. Therefore, each user can gain all container resources if the user gains access to the container.

Cloud computing is revolutionizing many ecosystems through offering companies computing resources that are easy to use, connect, configure, and are automatic and chosen to a suitable scale. In this project, a prototype that could represent a real world data centre is implemented by using container-based virtualisation.

TAIC allows each user in the system can perform particular actions within the container. Each user should have permission to do specific tasks within the containers. Only authorised users can access the resources within the containers that lead to making the user data availability. Set of rules using in this architecture that responsible for protecting user data and making it privacy. User data could not be changed by other users that make the user data integrity. Secure containers lead to build a secure environment that could be used in cloud computing and build trust relationships between cloud service provider and users.

This architecture modification raises a wide range of security and privacy issues that need to be put into consideration. Isolation in container-based virtualisation is a critical issue. Therefore, the thesis will also present a novel Trust Architecture for Isolation in Containers (TAIC) system to protect the containers from malicious guests and isolate users within the containers to boost the security of data that is stored in them through provide policies that allow each user to perform a specific tasks within containers and provision of data protection and security to cloud computing. Further, due to the centralized nature of data stored in cloud infrastructures, my proposed design will minimize data leakage and improve monitoring.

Acknowledgement

Firstly, I would like to express my sincere gratitude to my first supervisor Dr Michael Mackay for the continuous support of my PhD study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis.

Besides my first supervisor, I would like to thank my second and third supervisors Dr Nathan Shone and Dr Bob Askwith for their insightful comments and encouragement, but also for the hard questions, which motivated me to widen my research from various perspectives. I would also like to thank Liverpool John Moores University staff for encouraging me to carry out this project.

My deep and sincere gratitude to my wife and my brothers and sisters for their continuous and unparalleled love, help and support. To my beloved daughter Lara, I would like to express my thanks for being such a good girl always cheering me up. I am forever indebted to my parents for giving me the opportunities and experiences that have made me who I am. Thank you for encouraging me throughout the past years. Thank you for helping me through school when I was a child. Thank you for encouraging me to follow my heart, my dreams, and my God. Thank you for pushing me to be my best. Thank you for giving me the best life. This journey would not have been possible if not for them, and I dedicate this milestone to them.

Contents

1. INTRODUCTION 14
1.1 Background 14
1.2 Project Aims and Objectives 15
1.3 Novelty 16
1.4 Structure of the thesis 16
2. LITERATURE REVIEW 17
2.1 Introduction 17
2.2 Cloud Computing Background 17
2.2.1 Cloud Computing Definition 17
2.2.2 Cloud Characteristics 17
2.2.3 Cloud Service Models 18
2.2.4 Cloud Distribution Models 19
2.2.5 Virtualisation Platforms 20
2.3 Trust in the Context of Cloud computing 22
2.3.1 Definitions 22
2.3.2 Attitudes to trust 24
2.3.3 Resource isolation in Cloud Computing to Enhance Trust 25
2.4 Securing Cloud Platforms 26
2.4.1 Securing IaaS 26
2.4.2 Securing PaaS 30
2.4.3 Securing SaaS 33
2.5 Evaluation of Trust Techniques in Cloud Computing 35
2.5.1 Reputation Systems 35
2.5.2 Trusted Cloud Computing Platform to protect VMs 36
2.5.3 Third party verifaction mechanisms 39
2.5.4 Data security and integrity 41
2.6 Container orchestration platforms 42
2.6.1 Docker 42
2.6.2 Kubernetes 44
2.6.3 Docker Swarm 46
2.6.4 Azure Container Service 48
2.6.5 Google Kubernetes engine (GKE) 49
2.6.6 Amazon EC2 49
2.6.7 Cloud Foundry Diego 50
2.6.8 Container orchestration platforms analysis 50
2.7 Summary 51
3. ANALYSIS OF ISOLATION IN VIRTUALISATION SYSTEMS 52
3.1 Introduction 52
3.2 Isolation in Hypervisor-based Virtualisation 52

3.2.1 Virtualisation platforms 54
3.3 Related Work on Resource Isolation in Hypervisor-based Virtualisation 58
3.3.1 Micro Architecture: where virtualisation failed 59
3.3.2 Resource allocation with security requirements 60
3.4 Container-based Virtualisation 61
3.4.1 Linux Container features 61
3.4.2 Container Virtualisation Platforms 62
3.5 Isolation in containers 64
3.5.1 User isolation in container-based virtualisation 65
3.6 Evaluation of VM and Container-based Virtualisation Isolation 66
3.6.1 Related Work on Virtualisation Isolation Performance 66
3.6.2 Container Isolation Performance Using Docker 67
3.7 Summary 70
4. TRUST ARCHITECTURE FOR ISOLATION IN CONTAINER (TAIC) DESIGN 72
4.1 Introduction 72
4.2 TAIC Design Approach 72
4.3 Authorisation plugins 74
4.3.1 Twistlock 74
4.3.2 casbin/casbin-authz-plugin 74
4.3.3 Summary for each authorisation plugin: 75
4.4 Design Decisions and Technical Approach 75
4.4.1 Suggested system approach 76
4.4.2 Data and policies 77
4.5 TAIC System Design 79
4.5.1 Module explanation 80
4.5.2 Proposed system: 83
4.5.3 TAIC policies examples: 84
4.6 TAIC use case model 86
4.7 TAIC requirements (Verification) 86
4.7.1 Raspberry Pi Cloud (Picloud) 87
4.7.2 The Project Testbed. 88
4.8 Summary 89
5. TRUST ARCHITECTURE FOR ISOLATION IN CONTAINER (TAIC) IMPLEMENTATION 90
5.1 Introduction 90
5.2 TAIC Implementation Approach 90
5.3 Configuration project testbed 91
5.3.1 System Components. 92
5.3.2 Implemented System Architecture. 92
5.3.3 Raspberry Pi cloud. 93
5.3.4 Configuring SSH keys for each Raspberry pi nodes. 94

5.3.5 Create Docker Swarm. 96
5.4 Host implementation of TAIC model. 97
5.4.1 Host Implementation Overview 97
5.5 Authorisation Plug-in implantation 98
5.6 TAIC and TAICSSH scripts implementation 101
5.6.1 TAIC script 101
5.6.2 TAICSSH script 102
5.7 TAIC workflow model. 103
5.8 Summary 104
6. EVALUATION 105
6.1 Introduction 105
6.2 Testing Overview (Validation) 105
6.2.1 Security comparison 105
6.2.2 Testing objectives 106
6.3 Case 1: Cluster with Docker swarm only 108
6.4 Case 2: Cluster with Docker swarm and an authorisation plugin 109
6.5 Case 3: Cluster with Docker swarm and TAIC model 113
6.5.1 TAIC model testing for users that added to the system. 113
6.5.2 TAIC model testing for users not added to the system. 116
6.5.3 TAIC script testing: 117
6.5.4 TAICSSH script testing: 117
6.5.5 Protect the policy file, TAIC, TAICSSH scripts: 118
6.5.6 Analysis 119
6.6 Analysis of Results 119
6.7 Summary 121
7. CONCLUSION AND FUTURE WORK 122
7.1 Isolation in Container Based Virtualisation 122
7.2 Objectives and Contributions 122
7.3 Concluding Remarks 123
7.4 Further Work 124
7.4.1 Isolate users within the containers and protecting particular resources 124
8. REFERENCES 126

List of figures:

FIGURE 1. IAAS, PAAS AND SAAS SERVICE MODELS. 19
FIGURE 2. PRIVATE, PUBLIC AND HYBRID CLOUD DEPLOYMENTS. 20
FIGURE 3. THE DIFFERENCE BETWEEN HYPERVISOR TYPE1 AND 2. 20
FIGURE 4. HYPERVISOR VS CONTAINER-BASED VIRTUALISATION. 21
FIGURE 5. FACTORS AND LEVEL OF CONCERN ABOUT DATA PRIVACY. 25
FIGURE 6. SECURITY FEATURES FOR CLOUD IAAS, PAAS AND SAAS. 28
FIGURE 7. TRUST PLATFORM MODEL. 29
FIGURE 8. DEFAULT FIREWALL SETTING FROM MAJOR CLOUD PROVIDERS. 30
FIGURE 9. A COLLABORATIVE INTRUSION DETECTION AND PREVENTION SYSTEM. 30
FIGURE 10. OAUTH PROTOCOL. 31
FIGURE 11. FAMSL-MBS SCHEME. 32
FIGURE 12. BLOWFISH ALGORITHM WITH F FUNCTION. 34
FIGURE 13. USER IMAGES SHOWN AS A PIXEL MATRIX. 34
FIGURE 14. PRINT DROP WITH A DIFFERENT EN. 35
FIGURE 15. DISTRIBUTED-HASH-TABLE (DHT)-BASED TRUST-OVERLAY NETWORKS. 36
FIGURE 16. HARMONY COMPONENTS IN RESOURCE MARKET STAGES. 36
FIGURE 17. SET OF TRUSTED NODES (N) AND THE TRUSTED COORDINATOR (TC). 37
FIGURE 18. CCS REFERENCE MODEL. 38
FIGURE 19. CCS ARCHITECTURE SECURITY CONTROL FOR FOUR ARCHITECTURE BASED ON THE REFERENCE MODEL [48] 39
FIGURE 20. TM SYSTEM ARCHITECTURE. 40
FIGURE 21. CLOUD STORAGE ARCHITECTURE. 41
FIGURE 22. THE PROPOSED DATA SECURITY FRAMEWORK IN CLOUD ENVIRONMENT. 42
FIGURE 23. DOCKER SECURITY MODEL. 43
FIGURE 24. DOCKER DAEMON. 43
FIGURE 25. KUBERNETES ARCHITECTURE. 44
FIGURE 26 AZURE CONTAINER SERVICE 48
FIGURE 27. HYPERVISOR TYPE 1 52
FIGURE 28. THE SCHEME OF HYPERVISOR-BASED VIRTUALISATION. HARDWARE AVAILABLE TO GUESTS IS USUALLY EMULATED. 53
FIGURE 29. A MACHINE BEFORE VIRTUALISATION. 53
FIGURE 30. A COMPARISON CHART BETWEEN XEN, KVM, VIRTUALBOX, AND VMWARE ESX [69]. 54
FIGURE 31. XEN ARCHITECTURE. 55
FIGURE 32. KVM ARCHITECTURE. 55
FIGURE 33. A MACHINE AFTER VIRTUALISATION. 57
FIGURE 34. VMWARE ESX. 58
FIGURE 35. DIFFERENT HEURISTICS AND THEIR COMPLEXITY [30]. 60
FIGURE 36. KERNEL NAMESPACE AND CGROUP. 62
FIGURE 37. SHOWS THE PERCENTAGES OF DEGRADATION AND A SUMMARY OF THE RESULTS [7]. 67
FIGURE 38. TEST BED CONFIGURATION 68
FIGURE 39. BASE LINE HTTPERF RESULTS 69
FIGURE 40. AUTHORISATION ALLOW SCENARIO. 73
FIGURE 41. AUTHORISATION DENY SCENARIO. 73
FIGURE 42. THE SCENARIO TO ALLOW THE USER TO ACCESS A SPECIFIC RESOURCE WITHIN THE CONTAINER. 75
FIGURE 43. PROPOSED TRUST ARCHITECTURE FOR ISOLATION IN CONTAINERS (TAIC) MODEL 76
FIGURE 44. TRUST ARCHITECTURE FOR ISOLATION IN CONTAINERS (TAIC) OVERVIEW. 78
FIGURE 45. CONTAINER-BASED CLOUD DC SYSTEM ARCHITECTURE. 79
FIGURE 46. TAIC SYSTEM DESIGN. 80
FIGURE 47. TAIC SCRIPT 81
FIGURE 48. AUTHORISATION PLUGIN 82
FIGURE 49 TAICSSH SCRIPT 83
FIGURE 50. TRUST ARCHITECTURE FOR ISOLATION IN CONTAINERS (TAIC) FLOWCHART 84
FIGURE 51 TAIC USE CASE MODEL 86
FIGURE 52. AN EXAMPLE OF THE RASPBERRY PI [91]. 87

FIGURE 53. SOFTWARE STACK FOR PI CLOUD. 88
FIGURE 54. TIAC THREE-STAGE IMPLEMENTATION (TWO SCRIPTS AND AUTHORISATION) 91
FIGURE 55. SYSTEM ARCHITECTURE 92
FIGURE 56. TRUSTED CONTAINER PICLOUD IMPLEMENTATION 93
FIGURE 57. PICLOUD WITH SSH KEYS. 95
FIGURE 58 DOCKER SWARM CLUSTER 96
FIGURE 59 THE TAIC WORKFLOW MODEL 104
FIGURE 60 THREE CASES FOR SECURITY COMPARISON 106
FIGURE 61 CONTAINERS IN EACH NODE 108
FIGURE 62 THE ACTIONS THAT COULD BE PERFORM BY THREE USERS INSIDE THE CONTAINERS IN ALL NODES IN THE SYSTEM 109
FIGURE 63 TESTING RESULTS FOR CASE 1 CLUSTER WITH DOCKER SWARM ONLY 109
FIGURE 64 ACTIONS FOR ALL USERS IN TRUSTNODE01 110
FIGURE 65 THE ACTIONS THAT COULD BE PERFORM BY THREE USERS INSIDE THE CONTAINERS IN TRUSTNODE02. 111
FIGURE 66 THE ACTIONS THAT COULD BE PERFORM BY THREE USERS INSIDE THE CONTAINERS IN TRUSTNODE03. 112
FIGURE 67TESTING RESULTS FOR CASE 2 CLUSTER WITH DOCKER SWARM AND AUTHORISATION PLUGIN 112
FIGURE 68 TRUSTNODE01 USERS' PERMISSIONS 114
FIGURE 69 TRUSTNODE02 USER'S PERMISSIONS 115
FIGURE 70 TRUSTNODE03 USER'S PERMISSIONS 116
FIGURE 71 USER LEO REQUEST RESULT 116
FIGURE 72 FIGURE BELOW SHOW THAT TAIC SCRIPT AND POLICY FILE. 117
FIGURE 73 BOB AND TOM GROUP. 118
FIGURE 74 SSH CONFIGURATION FOR USER'S GROUPA AND GROUPB 118
FIGURE 75 TESTING RESULTS FOR CASE THREE: CLUSTER WITH DOCKER SWARM AND TAIC MODEL. 119

List of tables:

TABLE 1 CLASSIFICATION OF PRACTICAL MULTIPLE BIOMETRICS TECHNOLOGY FOR AUTHENTICATION 32
TABLE 2 THE DIFFERENCE BETWEEN KUBERNETES AND DOCKER SWARM 46
TABLE 3 TESTING RESULTS. 69
TABLE 4 THREE CASES SECURITY MATRICS 106
TABLE 5. PROPOESD TESTS 107
TABLE 6 SIX CONTAINERS TO VALIDATE THREE CASES. 107
TABLE 7 USERS RULES IN TRUSTNODE01 110
TABLE 8 USERS RULES IN TRUSTNODE02 111
TABLE 9 USERS RULES IN TRUSTNODE03 111
TABLE 10 USERS RULES IN TRUSTNODE01 113
TABLE 11 USERS RULES IN TRUSTNODE02 114
TABLE 12 THE RULES THAT APPLIED IN THIS CASE STUDY FOR ALL USERS 115
TABLE 13 THE ACTUAL RESULTS FOR THE AUTHORIZATION PLUGIN TESTING FOR USER OUTSIDE HOSTS, ... 117
TABLE 14 VALIDATE THE EXPECTED USERS BEHAVIOR 119

Acronyms:

Amazon EC2	Amazon Elastic Compute Cloud
ABAC	Attribute based access control
API	Application Programming Interface
ARP	Address Resolution Protocol
CAIQ	Consensus Assessments Initiative Questionnaire
CDC	Cloud Data centre
CFQ	Fair Queuing Scheduler
Cgroup	Control Groups
CM	Cloud Manager
COS	Console Operating System
CPDP	Cooperative provable data possession
CSP	Cloud Service Provider
DHT	Distributed Hash Table
EC2	Elastic Compute Cloud
En	Whereas entropy
ERP	Enterprise Resource Planning
ESX	Elastic Sky X
Ex	Expected value
GKE	Google Kubernetes Engine
GPL	GNU Public License
HCL	Hardware Capability List
He	Hyper entropy
IaaS	Infrastructure as a Service
IAM	Access Management Controls
IDPS	Intrusion Detection and Prevention System
IDS	Intrusion Detection System
IPC	Inter-Process Communication
IPS	Intrusion Prevention System
IS3	Information System Security System
KVM	Kernel-based Virtual Machine
LXC	Linux Containers
MAC	Mandatory Access Control
MPI	Messaging Passing Interface
N	Trusted Node
OS	Operating System
PaaS	Platform as a Service
PiCloud	Raspberry Pi Cloud
PID	Process ID
PLTs	Propositional Logic Terms
PMs	Physical Machines
Policy, Effect, Request, Matchers	PERM metamodel
QoS	Quality of Service
RFA	Resource-Freeing Attacks
RHEV	Red Hat Enterprise Virtualisation
RNG	Random Number Generator
RBAC	Role-based access control
SaaS	Software as a Service
SDN	Software Defined Network
SEDs	Self-Encrypting Drives
SLAs	Service Level Agreements
SMB	Small and Medium Business
SPI	Stateful Packet Inspection
SSH	Secure Shell
TAIC	Trust Architecture for Isolation in Containers
TAICSSH	Trust Architecture for Isolation in Containers SSH script

TBF	Token Bucket Filter
TC	Trusted Coordinator
TCCF	Trusted Cloud Computing Framework
TCCP	Trusted Cloud Computing Platforms
TCE	Trust Computation Engine
TI	Trust Information
TMg	Trust Manager
TNC	Trusted Network Connect
TPA	Third Party Auditor
TSE	Trust Semantics Engine
TSS	Trust Platform software Stack
TUE	Trust Update Engine
TVDc	Trusted Virtual Datacentre
TVDs	Trusted Virtual Domains
TVMM	Trusted virtual machine monitor
TZ	Trust Zone
VLAN	Virtual Local Access Network
VMFS	Virtual Machine File System
VMM	Virtual Machine Monitor
VMs	Virtual Machines
VTPM	Virtual Trusted Platform Module

1. INTRODUCTION

1.1 Background

In the cloud computing, security and privacy are the main issue that should be protected by the cloud provider to help provide trust [1]. Therefore, security guidelines and best practice recommendations are provided for cloud computing are provided by several studies that relate to cloud security. For example, the Data Security Council of India provides a study that investigated how companies in India deal with security risks when adopting cloud computing. Most of the companies are identified the solution to reduce the security risks and unavailability of data by negotiating legal terms with the cloud provider [1]. The terms of usage and conditions that are provided by the cloud provider have also been analysed by serval studies. The Pew Research Centre study shows the levels of privacy concerns in American Internet users, where 63% of the participants said that the cloud provider should not have any copy of the files that the customers tried to delete. Also, 49% of the participants said that the participant's files could be an issue of concern if the provider gave them to law enforcement agencies when asked [1].

Trust can be defined as a positive expectation towards the behaviour of another. In the context of the cloud, the cloud users trust the intentions and actions of the Cloud Service Providers (CSP) with their personal data and information [2]. Trust in this context can be explained as believing in a promise that has been made by brands or technology [3]. People tend to lose trust in a system if they find that the information given to them is inaccurate. Furthermore, customers need some form of proof in order to believe the claims made by the provider. For instance, CSP needs to show evidence to potential users that their cloud environment is secure and will protect the privacy of its users. This project aims to identify solutions that would help to create and enforce this trust-based relationship between CSP and cloud users.

Trust is a critical issue in cloud computing because of the remote Internet-based nature of the service, which makes the user dependent on their provider for the effective running of the infrastructure. This includes the privacy and security of any data stored remotely, the integrity and robustness of the services running in the cloud and the proper isolation of different tenant slices on shared infrastructures. A user must therefore trust the cloud provider's assurances that all necessary security measures are being deployed and enforced to protect their assets within the cloud. This includes a range of security and network management systems and services such as end-to-end encryption, strong user authentication, robust virtualisation, and active infrastructure monitoring to name but a few. However, cloud providers are not necessarily forthcoming regarding the configuration of their infrastructure and lack extensive third-party auditing. Therefore, customers are not sure whether they can identify a trustworthy cloud provider or not.

Cloud Computing is now a congested marketplace with many different types of services being offered by competing solutions. Choosing the best cloud service for a specific application is a challenging issue and one of the key determining factors is often security and trust. As such, users often look for guarantees that their data will be kept in the secure environment to ensure the privacy and integrity of their data and applications [4]. Service Level Agreements (SLA) and privacy-awareness are two main factors in a dynamic cloud service trust evaluation. Before subscribing to the cloud, the user should carefully review/negotiate the SLA to evaluate the commitments made by the CSP towards data privacy, integrity, and security. For example, data security can be enforced by

using strong encryption techniques. To improve trust in the cloud, much research has been done over the last few years to develop new mechanisms for improving the security of resources placed in the cloud [5] or the security of the underlying technologies [6], with differing degrees of success.

One aspect of trust that is especially critical for cloud computing is the need for isolation. Due to the shared tenancy that is a central feature of virtualised infrastructures, providers need to enforce strong mechanisms to ensure that services running on the same server do not interfere with or impede each other and that users cannot 'break out' of their allocated virtual environment. Traditional virtualisation typically provides a high degree of isolation because the level of abstraction enables the virtual machines to only share hardware resources without interference with other virtual machines' software stack so intrusions and malware that compromise one virtual machine therefore cannot easily affect the other machines running on the same host [7]. However, a recent trend in cloud virtualisation is away from traditional 'full' virtual machines and towards containers [8]. These offer significant performance advantages due to their reduction in overheads through sharing a common system up to the OS kernel, but suffer from a resulting lack of isolation. This project aims to improve the isolation offered by container-based virtualisation.

First, the evaluation of the issue of isolation in container-based clouds in details provided in this project to understand why this is an issue and how significant this difference is in practice. By performing a comparative analysis between hypervisor-based and container-based virtualisation that would help to understand where the main issues occur and how this project can address them to enhance trust in the cloud. Then, the approach and solution design are described to enforce isolation in container-based systems and identify the main components and operational considerations. Finally, the solution is implemented in a realistic container-based cloud environment and evaluated how far our system enhances trust.

1.2 Project Aims and Objectives

The aim of this project is to create a trust architecture that facilitates the building of a trust relationship between cloud service providers using container-based virtualisation, and cloud users. The cloud provider should show evidence to the cloud users that the cloud environment is secure, and the user's data is protected. The objectives of this research that will be pursued to fulfil the aim of this project are as follows:

1. Identify solutions that would help to create a trust-based relationship between cloud service providers and cloud users.
2. Prove that the container-based virtualisation has weak isolation characteristics compared with traditional VMs, by providing related works and experiments related to isolation in containers.
3. Create a trust architecture by using container-based virtualisation to build a trust relationship between cloud service providers and cloud users.
4. Implement a prototype based on the resultant architectural design
5. Evaluate and ensure that the designed cloud-computing framework will assist in overcoming security and privacy concerns to increase trust in state-of-the-art cloud computing.

1.3 Novelty

This research project has the following novel contributions:

This research will propose a trust architecture to address user isolation within containers to protect them from potentially malicious guests and improve the availability of data stored in them by providing policies for each user that allow them to only perform specific tasks within the container. This model offers enhanced data protection and security to cloud computing based on container virtualisation.

This contribution is both novel and valuable because container platforms do not currently allow the isolation of users within a container. This issue is because the containers' design only allows it to run one process for every container. While this is traditionally the case, there are instances where it would be highly advantageous to extend this approach to support multiple processes inside a single container. Therefore, this project creates the TAIC model that could help to ensure each user can only perform specific actions inside containers.

1.4 Structure of the thesis

This report consists of seven chapters. Chapter 2 is dedicated to the literature review, which first outlines the main aspects of cloud computing and explains the basic concepts. In addition, it provides a review of trust in cloud computing, including a definition of trust in this context, the main attributes of trust, and some of the trust evaluation techniques in cloud computing. A review of existing container orchestration applications and engine services are also provided at the end of this chapter. Chapter 3 is the analysis that focuses on container-based virtualisation and the differences between it and hypervisor-based virtualisation. Next, the chapter presents some related work on VM and container-based isolation and testing. Finally, the security requirements that will address the isolation issues in container-based virtualisation are identified in this chapter. Chapter 4 introduces the design approach of the proposed system, including the overall system design and model explanation. Moreover, some approaches for developing an authorisation plug-in are reviewed in this chapter. Finally, the appropriate approach is explained for this project. Chapter 5 is dedicated to the implementation, which focuses on the development of the authorisation plug-in developed using the GO language, as well as providing an introduction to the project testbed, system components and implementation system architecture. Chapter 6 is the evaluation that tests our prototype based on the use case of a real-world data centre. Finally, Chapter 7 concludes this project and outlines our proposed future work.

2. LITERATURE REVIEW

2.1 Introduction

Trust is an important issue in cloud computing. Cloud computing allows cloud users to access a large number of shared resources, software, and applications online. Building a trust relationship between cloud service providers and users could be done by improving the security in cloud environment. Physical interaction is easy to trust compared with online services because the users just access services but do not know anything about security and network management. This chapter provides related works that focus on and security in cloud computing.

This chapter is divided into six major sections. The second section provides an overview of cloud computing definitions, characteristics, service models, distribution models, and virtualisation platforms. Trust in cloud computing is defined in section three based on related works. Besides, section three introduces the concept of resource isolation in cloud computing. The fourth section provides techniques that would help to create secure cloud models, and section five evaluates trust techniques in cloud computing that provide important elements to increase trust in cloud computing. The existing orchestration applications and engine services are then presented in section six.

2.2 Cloud Computing Background

2.2.1 Cloud Computing Definition

Cloud computing can be explained as a collection of data that is placed on web based systems using isolated servers that are managed by a third party [9]. This means that users can have access to their data from anywhere and at any time using the services provided and it is not necessary for users to have any specialised local software or hardware since everything will be accessible remotely. This computing method is highly profitable for organisations, as they do not have to maintain or store different hardware and software, which results in lower expenses and operational costs [10].

There is an on-going shift in the software business moving to adopt cloud strategies, which accounts for more than twelve percent of the global software industry. There is a huge potential for cloud computing to revolutionise the computing industry, from users, to SMB, through to large enterprise organisations. However, at the same time it was also reported that more than seventy-four per cent of information professionals and top management identified security issues to be one of the leading challenges in the industry that is hindering the growth of the cloud, since organizations are still sceptical about adopting and placing their data and services remotely on the cloud [11].

2.2.2 Cloud Characteristics

According to [12], one of the key features of the cloud is that a user can have accessibility to any form of data without having to install a specific software program on their computers. For instance, users can directly access and use applications like GMail or DropBox from any country in the world that does not have restricted internet access, and know that it is safe to access. The authors in [13] have identified five different factors that can define the cloud and its services. The *first* characteristic is that the cloud provides accessibility on a massive scale that allows its users to gain access to thousands of applications, software packages and systems, on top of

having huge storage space. The *second* characteristic of the cloud is that there is the element of resource sharing between different cloud providers and users, which allows cloud users to access a single application with multiple accounts at the same or different times. This reduces restrictions on time and place of access. The *third* characteristic is that cloud services are very flexible in order to cater to their user's needs. Users have the ability to increase the resources and storage required with time and budget availability, making it highly responsive. The *fourth* characteristic is that users only have to pay for resources that they have used or will be using, which saves a lot of money when compared to investing in local resources. This also allows the users to budget according to their needs. The *fifth* characteristic is that users can allocate the required resources themselves without having to contact the service provider or seek professional assistance [14]. This gives users more independence and confidence to manage and run their own cloud profile according to how they want to manage, store and view their data.

2.2.3 Cloud Service Models

Cloud computing systems represent a wide range of service delivery models depending on the level of service being offered. The respective computer systems delivery models are Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS).

In the SaaS model, the Cloud Service Provider (CSP) hosts the application together with the data which the user deploys and operates [15]. Here, the user will have access and control over the software functions but does not incur the cost of acquiring, deploying, or configuring it. Furthermore, the cloud service provider will also provide some help in running and maintaining the software for the client. Widely understood examples of SaaS include GoogleDocs and Gmail from Google but this cloud system is widely applied and covers a broad range of services.

In contrast, PaaS provides the user with an environment in which they can deploy their own applications and services. In this case, the CSP provides the platform on which the user can deploy software, but they do not own or manage it themselves. PaaS services also typically provide different 'off-the-shelf' applications for organizations without complexity or high costs, which usually requires significant investment [16]. Besides that, the costs associated with maintaining and sustaining the platform and applications over time will be 'pay-as-you-go'. Well-understood types of PaaS are Microsoft Azure and Google App engine.

Finally, according to the IaaS model, the CSP only provides virtual server instances as well as the Application Programming Interface (API) to create, configure, access, and eliminate them. The user can then deploy to them whatever platforms, services, or applications they wish. Furthermore, the user can establish the storage systems and network modules, as well as other computing properties based on their specific requirements. As such, the CSP controls the primary cloud infrastructure, but the user has regulation over the functional structure, installed services, storage as well as applications, and partially networking modules. Organizations can opt for the different service models as well as the size of storage and networking features required, allowing for budget spending according to their needs [11]. For example, the users may be provided with storage and backup functions for files and information stored on the cloud, as well as private (firewalled) connectivity domains based on VPNs. Well-understood cases of IaaS are Google Compute engine as well as the Amazon EC2.

The service provider will also provide the infrastructure or framework required for the clients to run their business, including the servers, data centres as well as other required equipment from the cloud depending on the service agreement [16]. Figure 1 below shows the difference between IaaS, PaaS and SaaS.

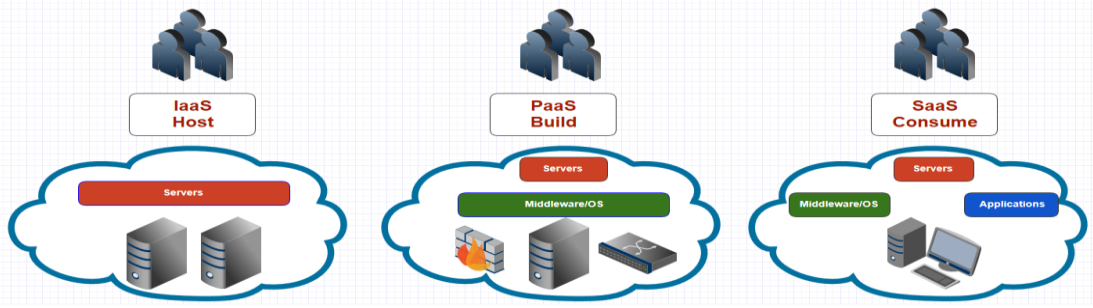

Figure 1. IaaS, PaaS and SaaS Service Models.

2.2.4 Cloud Distribution Models

In addition to the service models, there exist a variety of cloud types that are based on the diverse deployment models in cloud computing. These types are defined as private, public or hybrid. This division is based on whoever owns, manages, as well as accesses and uses the cloud resources. The same cloud applications and service models are often available on the private, public and hybrid cloud [17].This project defines a public cloud as being deployed by a single organisation for use by a range of external users, whereas a private cloud is deployed and used exclusively by a single organisation. The public and private clouds accompany themselves with multiple advantages and disadvantages.

The benefits of a private cloud include the overall control of the infrastructure, which provides maximum security, compliance with specific requirements, as well as greater customization. Businesses can therefore expect better control over their data systems due to the existence of hardware on site and there is also the ability to better monitor how the platform is performing and how it is being used. Because all private clouds are for single organisations, data storing, hardware provisioning, and network configuration, can be planned to offer maximum performance and safety. However, the private cloud comes with shortfalls, such as maintenance complexity, high costs as well as the process size ceiling. Businesses still need to procure the necessary storage, hardware as well as networking properties and maintain all these resources, which can make private clouds costly and difficult to organise and preserve. On-going maintenance and management costs may also be restrictive in this instance.

In contrast to private clouds, a public cloud provides accessibility to any user through a shared network service. The cloud is often deployed on the operator's own infrastructure and can be used by a combination of companies, individuals, or other groups. Public clouds can scale across multiple data centres and offer services to users in many countries. Public clouds therefore tend to be more generic and less tailored to a specific application or user group and tenants from different organisations may share the same physical infrastructure. The advantage here is that a user needs no specific hardware or software to use cloud services, which reduces cost significantly. The issues with public clouds tend to revolve around security, privacy and reliability since users are entrusting their services to a remote platform operated by a third party.

Finally, hybrid clouds bring a combination of both the public and private model. This may be in situations where the organization may need a large capacity cloud to handle its users, while specifically providing limited accessibility of the cloud's controls to its employees using the private cloud. According to [2], there are another

two additional types of cloud deployment models, which are community cloud and partner cloud. A community cloud has an infrastructure that is capable of being shared by different organizations with similar concerns or goals similar to a computing grid. Partner based clouds are based on cloud service providers that offer limited services to a number of different parties. Figure 2 below shows the difference between private, public and hybrid clouds.

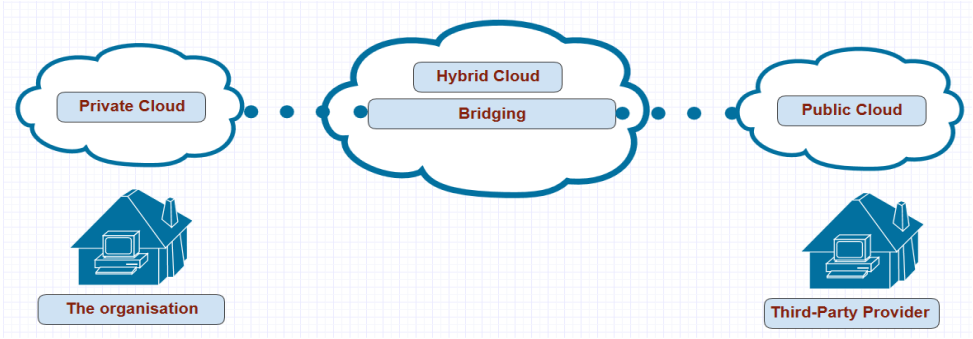

Figure 2. Private, Public and Hybrid Cloud Deployments.

2.2.5 Virtualisation Platforms

Hypervisor-based VMs are the key to scalable cloud services that allow each virtual machine to have its own operating system (OS), libraries and applications, but run on the same physical server. In this approach, VMs cannot interface with each other to protect customer's privacy, and this isolation is an important aspect of building trustworthy virtualisation. Independent OS and application software are therefore a key advantage in hypervisor virtualisation. Hypervisor type 1 and type 2 are the two main types of virtualisation [18]. Type 1 runs directly on the physical hardware while type 2 runs on top of the host OS. Therefore a host OS is not required in type 1, making it more efficient, but is in type 2. Figure 3 shows this difference between hypervisor type1 and 2.

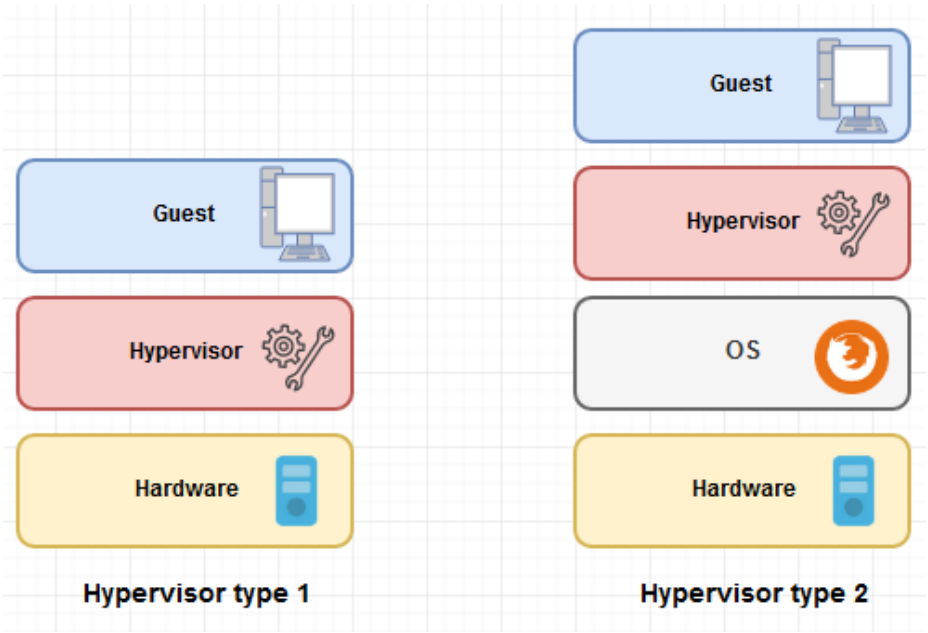

Figure 3. The difference between hypervisor type1 and 2.

Nowadays, Cloud Computing and distributed systems are increasingly supported by container-based virtualisation. In this method, nodes have numerous 'guest' simulated machines that are situated on the top of a common OS core, these isolated components are referred to as containers [19]. With container-based virtualisation, there exists no link with each host running a fully installed OS, this is shared between all instances (see Figure 4). A container-based virtual system is a good hosting choice for providers who mainly need to offer multiple instances of the same OS. These instances typically perform substantially better than traditional VMs due to the reduced overheads. There is an increase in container-based virtual applications, which includes Docker, the most de facto approach, as well as OpenVZ and LXC.

In comparison, container-based virtualisation differs from the VM based virtualisation process in that the latter is provided comprehensively on the hardware system. Additionally, in the hypervisor type 2, the main hypervisor OS has to make modifications to the guest set-up, it is only able to modify the hardware properties and not any other component. The hypervisor is located primarily between the guest system and the physical hardware and controls the allocation of resources to the guest that is located above it. The hypervisor approach to virtualisation therefore provides inclusively complete isolation of the applications but comes with a large overhead resource management. Both containers and VMs have advantages and disadvantages and therefore can complementarily work alongside each other providing a perfectly sustainable set-up depending on the application or service that should be deployed [20].

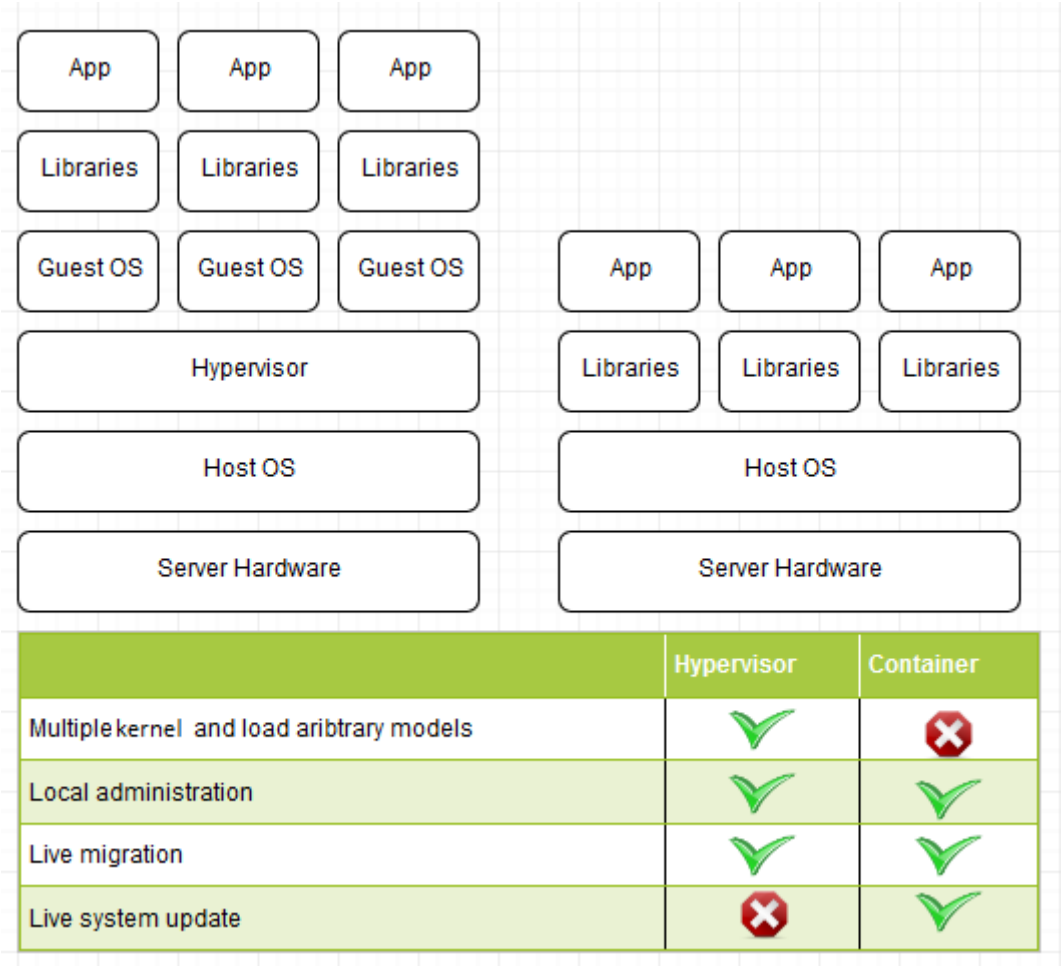

Figure 4. Hypervisor Vs Container-based virtualisation.

2.3 Trust in the Context of Cloud computing

2.3.1 Definitions

Trust can be defined as a positive expectation towards the behaviour of another. In the context of the cloud, the cloud users trust the intentions and actions of the cloud service provider with their personal data and information [15]. There are two trust categories, mainly soft and hard trust. Hard trust is explained as having to authenticate and participate in security measures when logging in and out of the cloud system. On the other hand, soft trust focuses on human-based issues such as customer loyalty to a specific service provider or even the aspect of how user-friendly the system is. Trust is often placed on a brand and its image, and therefore, if any security breach occurs, the brand image could suffer negative consequences.

It is more difficult for people to trust online services compared to face-to-face interactions because of the absence of the physical interaction that increases the possibility of trust. Trust can be achieved mostly through tangible aspects since people can believe what they see and touch as opposed to just viewing things online. Therefore, online services such as cloud services will need time to gain customers' trust in their services because the clients cannot see the specific security details for the online services but they can get services only. Also, a negative cloud reputation in some organisation could easily impact other users' perceptions of the service although other organizations may be having a good reputation with their customers. Therefore, it is important for cloud service providers to consider the issue of trust seriously [21].

Trust in technology can be explained as believing a promise that has been made by brands or technology [9]. People tend to lose trust in a brand or system used if they find out that the information given to them is inaccurate. Furthermore, customers also need some form of proof in order to trust the technology. For instance, cloud service providers need to show evidence to potential cloud users that the cloud environment is secure and will protect the privacy details of its users. Four main issues have been identified that need to be addressed in order for customers to trust the product or service provided by a CSP. The first issue is with **control** as people tend to have higher levels of trust over things that they can control. Therefore, in the case of cloud users, most of the control lies with the cloud service provider or third-party agents, and this makes potential cloud users feel less in control, and therefore, have less trust in the cloud service providers.

The second issue is pertaining to **ownership** whereby a higher level of ownership indicates a higher level of trust. This is because people tend to feel more obliged and have higher levels of trust when they own their own individual space or items compared to shared space. Therefore, in the case of cloud services, cloud users should feel that they own their own space, which will lead to more trust in cloud service providers [14]. The third issue is the issue of **prevention**. With regards to cloud computing, the cloud users will have a higher level of trust towards the provider if it shows evidence of having taken measures to prevent hacking or other malicious attacks from taking place, rather than having codes of conduct for security risks. Although many cloud service providers often describe the compensation or ways to handle users after a security breach takes place, most users still want to have some form of guarantee towards prevention. However, this is impossible for any service provider to promise to their potential customers due to the high security risks at hand.

The fourth issue is the most discussed issue in the cloud computing studies, which is **security**. Security is one of the key challenging areas for most cloud service providers and users, due to the possibilities of various attacks and security breaches that could take place. At present, many academic studies and cloud-based

organizations are in search of proper security measures to secure the cloud environment. Some of the risks involved include the management of personal data on the cloud, leakage of data due to the fact that many users share the same resource, authorisation issues and other data related issues. Examples of security measures include the VMsafe application that claims to provide a secure VM for any platform in the cloud environment [20]. However, these are still the issues that are at the lower level of security concerns, and CSP still needs to address bigger concerns in the cloud environment.

The authors in [15] clarify that trust comes with increasing security measures in a cloud environment. For instance, with the requirement of authorisation and encryption of personal data, the perception of increased security translates to a safe site. Trust is a process that has different stages such as first building the trust for potential cloud users, then maintaining the trust and then the decline of the trust. Researchers have also found that trust is difficult to develop between the service provider and the user, however it is very easy to lose this trust due to a small mistake. The fuzzy cognitive method is one approach used to study and measure the level of trust the users have towards a brand or an organization [22]. The authors also suggest the use of persistent and dynamic types of trust to continue to maintain a positive relationship with cloud users. Persistent trust is defined as the trust that is based on the infrastructure of the company, which is indicative of the technological and social means used for the brand. On the other hand, dynamic trust specifically focuses on short term information or on specific areas that can increase through social and technological means. Trust that is based on the company's assets and technology should be important to the organization and cloud providers, however, the social component that provides the more confident form of trust should always be considered to ensure that the customer is always comfortable with the brand and the organization.

Trust is a key factor that is preventing the growth of cloud computing in the market [10]. The usage of cloud services requires trust since the users would have to transfer all of their information and data, as well as control of these resources to the cloud service providers. Although cloud computing provides multiple benefits to the users, cloud service is often a highly distributed system which makes it more challenging to have a transparent system for users to gain trust in the provider and service [16]. On the contrary, some researchers argue that the security component in cloud services does not impact the level of trust in potential cloud users since there are no links or relationships established between trust and online usage [23].

Cloud computing allows cloud users to access different types of online resources, but the challenging issue is choosing trustworthy services in the cloud infrastructure. The authors in [24] have proposed a scheme for cloud storage that cloud users to trust cloud provider and use the features of cloud computing. Only authorised users can obtain the data from cloud storage and the Cloud provider uses some security method to grant access to the authorised user only. The schema provides the user data with integrity, confidentiality and dynamicity.

A selection framework for cloud services is suggested by [24] to increase the trust in cloud computing. The authors suggest a method based on two main factors, objective trust and subjective trust. Monitor the QoS is the objective of the objective trust valuation. User feedback scores are the objective of subjective trust valuation. A synthesised data set is then used in Matlab experiments. Suitable reliability is offered by these experiments but low confidentiality, scalability and safety.

Another method is provided by [25] for cloud trust and reputation evaluation via the recommendations of opinion leaders. This assesses the trust value by using five elements that are accessibility, reliability, data integrity, identity, and ability. Input and output degree, and reputation measures are the three topological metrics that are

used to offer a method for opinion leaders and malicious entity identification. Again, Matlab simulations show this method show suitable security, integrity, and safety but provide low confidentiality and low scalability.

2.3.2 Attitudes to trust

Reputation in the cloud is an important area of focus that leads to building trust. Reputation management in cloud computing mainly focuses on understanding an image or brand value from other people due to the inability to draw conclusions based on a lack of knowledge or experience [26]. This form of feedback or opinion received will then generate a form of trust that is based on either the negative or positive feedback received from others. Therefore, in the case of cloud service providers, it is important to maintain a good reputation with their customers, since this may affect the perception that other potential users will have of the provider. Furthermore, reputation also tends to grow with time based on people's observation of the quality of the service offered. Therefore, adhering to Service Level Agreements (SLAs) is important since many potential users can easily identify the company's reputation by seeing whether the SLAs for previous customers are met.

Although there are many discussions pertaining to the advantages of cloud service providers, there are still considerable discussions pertaining to the cloud users' concerns over privacy and security since there are many potential threats and well-publicised attacks that have been discussed in this area. Some of the strategies used to protect the privacy of users include limited access and encryption-based methods that have proved to be insufficient since they have been easily hacked. It was also found that intrusive attacks in cloud computing have become more challenging to identify and prevent as they become more sophisticated [12].

A study conducted by the Fujitsu Research Institute in 2010 reported that close to ninety per cent of potential users are concerned about who has control over their data access and control, and were interested to learn much more about the backend operations [21]. The study also found that most potential users are not comfortable with the private or public sector to protect their data since many potential users are suggesting that the controls access and use the data be limited to the user itself. This highlights the need for cloud service providers to identify and analyse the issues of trust in cloud computing related to the transparency of the operator and isolation in shared infrastructures. The study concluded that it was almost impossible to find a cloud computing solution that fits every need. Different factors such as age, country of origin, gender as well as the user's comfort level with technology create different views and opinions towards how they see security risks, and how they would like to address this issue. For instance, more than ninety percent of American respondents wanted cloud service providers to get their permission before sharing any of their information, while only seventy-seven percent of Japanese respondents wanted to be asked for permission in sharing their information. This reinforces the strong social aspects towards trust. Figure 5 shows some of the different factors and the level of concern to potential cloud users.

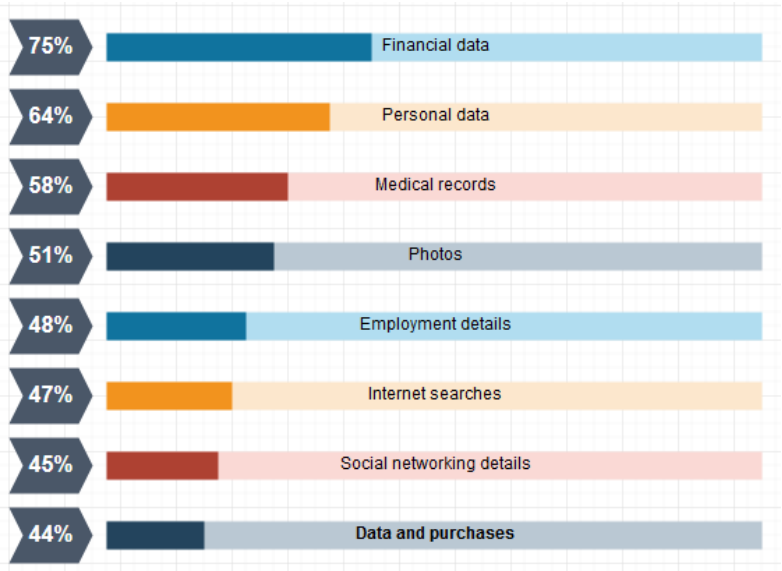

Figure 5. Factors and level of concern about data privacy.

The responsibility for security and privacy in cloud computing environments should be shared between the cloud provider and users [27]. In SaaS, cloud users can only obtain online services that are provided by the cloud provider. Therefore, security and privacy are the provider's responsibility. For example, the provider should use encryption or data colouring to protect the application services. In PaaS, the applications that are built and run on the cloud platform by the customers should be protected by the customers as well. However, isolating the customers should be offered by the cloud provider and the isolation should include applications and workspaces. The provider should use security methods to evaluate the traffic over the network to build secure IaaS. Cloud computing is based on the concept of sharing hardware resources between different customers. Sharing hardware resources is a huge concern from a customer's perspective. Therefore, the isolation is a very important issue in cloud computing to build a secure cloud environment.

Privacy is also a critical issue in cloud computing because the customers should have a high level of trust and be comfortable about sorting their data in cloud datacentres. Sharing the hardware resources between different customers however may increase the possibility of attacks and unauthorised users to access user's data in cloud datacentre. Therefore, the cloud environment should be secure and enforce privacy to make the users trust the cloud services.

2.3.3 Resource isolation in Cloud Computing to Enhance Trust

Isolate user resources in cloud computing could increase the possibility of trust and building trust relationship between CSP and users. A number of heterogeneous virtual resources, e.g. virtual machine (VM) are offered by the CSP of infrastructure-as-a-service (IaaS) to the cloud users [28]. VM is offered to isolate cloud users and each user should have its own OS, libraries and resources. A Cloud computing datacentre could have thousands of resources from different customers, which make the Management process of these virtual resources very complex and this system vulnerable because it could increase the likelihood of misconfiguration. On the other hand, cost reductions and economic benefits of the CSP are enabled by sharing a hardware resource among different customers' workloads. However, different security vulnerabilities for the customers' workloads such as cross-channel attacks and denial-of-service could be led by this situation.

According to [29], isolating different users on cloud platforms is performed by the virtualisation, which is used as the sole mechanism and could increase the trust between cloud service providers and users. In cloud computing, the developers use virtualisation, which is now widely used in the modern data centre on cloud platforms to isolate different users. The virtualisation means each user has its own OS, applications and libraries. In virtualisation, each user is completely isolated from others. That means bugs and viruses could compromise on VM but could not carry over onto others. Therefore, many applications of different kinds are provided by cloud computing. Applications over to IaaS or PaaS (Platform as a Service) should provide good security and performance for the users. In public clouds and virtualisation software stacks, several issues have been raised and security has been proven sometimes difficult to obtain. Moreover, the performance could be degraded because shared physical machines are not appropriate for high-performance applications. Isolation is consequently a critical issue for both security and performance concerns.

Isolating computing resources and networks between customers is one of the goals of the current cloud computing services because the success of cloud computing depends on the economy of a large scale [29]. Isolate the users resources is evidence that could be provided to the cloud users to make sure the cloud environment is secure. Sharing resources and multiplexing among customers are provided to the cloud customer through cloud service and this advantage is provided by the cloud computing. Different customers could share the same LAN for their data and their virtual machines could reside on the same physical machine. Therefore, cloud-computing providers should provide a high level of isolation for the users to avoid some security risks. For example [30], if the users share hardware resources, it is possible for a hacker to conduct attacks towards another Amazon EC2 user with the hacker in the cloud. On the other hand, the government cloud as proposed by Google is an example of where special customers require separate hardware, software, and administrators (with appropriate background checks) [29], which is therefore very secure and expensive as a cloud service.

2.4 Securing Cloud Platforms

2.4.1 Securing IaaS

In cloud computing, user's data is stored in cloud datacentre and the users can access their data online that could increase the possibility of attacks [31]. For example, in SQL injection attacks, the authorised user could access the sensitive data in the SQL database by using a malicious code added into a standard SQL code. The attacker could change or modify the contents of cookies through Cookie Poisoning that allows the unauthorised user to access applications or data. The user's identity is part of this cookie that could be changed by the attacker to access the user data. In addition, cloud users should be aware of account service and traffic hijacking. The attacker could steal the user account and can then obtain the user data from cloud datacentre. In addition, Cross Site Scripting (XSS) attacks are based on injecting malicious code into the web page. The attacker could obtain the user's password, username and data through the user cookies. The attacker could read all the information related to the users.

Many users share the cloud computing resources that increase the possibility of DoS attacks [32]. The high workload in cloud computing makes the cloud provider allocate more computational power. The cloud servers could be the targets of DoS attacks. Therefore, the cloud provider could lose user information or receive such as high workload that could be lead it to slow down or stop the server.

A Cloud Malware Injection Attack is based on injecting a malicious service implementations or virtual machines into the cloud system. For example, malicious IaaS, SaaS or PaaS could be created and added to the cloud system. Then, the user requests could be automatically redirected to malicious service implementation. The attacker could have full control of the user's data that are in the cloud computing datacentre. As well as this, the attacker could have privileged access capabilities to attack the security domains in the cloud system. Malicious virtual machines could also be placed in the cloud system to launching a side channel attack that targets system implementations of cryptographic algorithms or Man-In-The-Middle Cryptographic Attacks that is based on two users and an attacker between them. The communication paths could be attacked by an attacker that places himself between the users and modifies the communication path or a form of interception could be performed by the attacker.

VM security is very important to building a trusted cloud platform. According to [33], the study identified four main types of attacks that can take place in the cloud environment. The first attack is based on snapshots or memory dump taken of the cloud user's passwords and data. This can easily be done and usually targets the user's VM. Single command while logged as root could be issued by the administrator to obtain the memory dump, which would help to extract passwords. The attacker could obtain the user password by using a set of commands to obtain the memory dump that has user information. The second type of attack focuses on the stored keys used by the cloud user. Once again, the attacker can easily trace these stored keys in the VM files and use these to hack the user's account. For example, establish secure channels are provided by the Apache server to the clients. Keys are provided by Apache server and located in memory. Therefore, the attacker could have the key then use these to hack the user's account. The third attack is when hackers focus on removing data that is confidential from the user's hard disk. Finding victim VM volume groups is the objective of attack. The attacker use some techniques to find the victim VM volume groups, and then activates the logical volumes existing in it. The last attack is based on a hypothetical situation whereby the cloud environment has been fully protected using methods that do not allow any form of changes to take place, hence tightening security measures for the user's data. However, even in this situation the relocation of the user's VMs can also be done in the IaaS platform in the cloud.

A few critical issues for building trust in cloud computing were identified by The Cloud Security Alliance [5] but there is a different level of security required in public and private clouds. Data integrity, confidentiality trust between providers, individual users and user groups were the critical security issues identified in every case. In another study conducted by [6], it was reported that trust was a vital component to be combined into a computing system using an inclusive platform for cloud users' security. This inclusive platform included different factors often considered by cloud users such as privacy issues. According to the study, security is also one of the key factors that many cloud users and providers are often concerned about. Furthermore, the fact that clouds use a virtual infrastructure often requires a higher level of trust to exist between the cloud service provider and the cloud user. Therefore, having authorisation as a form of security measure is not only useful, but also highly necessary in order for trust to exist between these two parties. In summary, each level in cloud computing needs some security and privacy measures as shown in Figure 6 below.

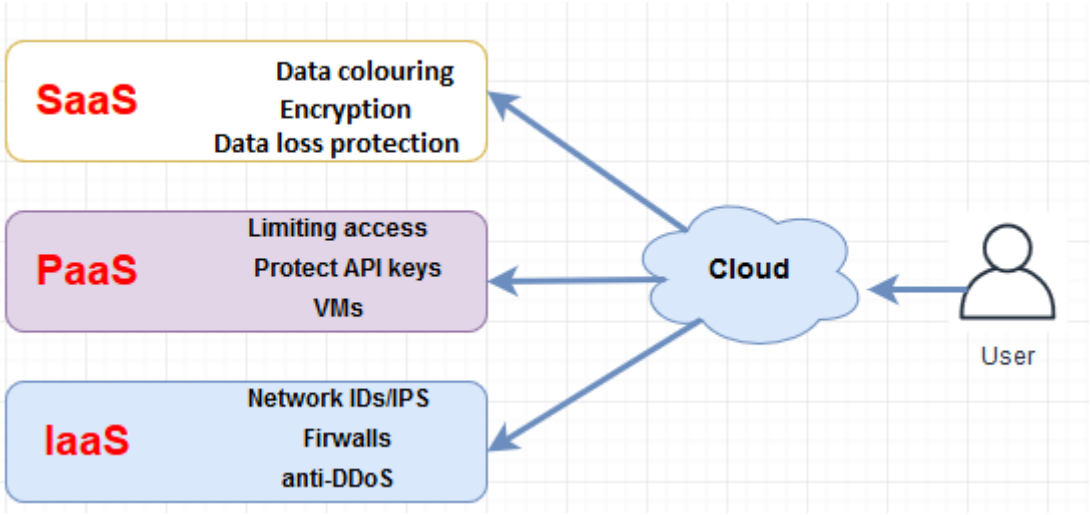

Figure 6. Security features for Cloud IaaS, PaaS and SaaS.

Present systems maintain security of the IaaS platform by focusing on the detection of malicious codes using anti-virus programs as well as firewalls. This is because the user is not in control over the infrastructure nor does the user manage the system. Therefore, it is the duty of the service provider to handle security related issues. In the IaaS, the provider could add some hardware to trace the user's behaviour to provide secure infrastructure as a service [6]. In this Trusted Cloud Platforms, there are some mechanisms, which are integrated into the hardware to trace the users and get their origin. The users and other resource trace mechanisms can be known by the cloud computing trace mechanism. Therefore, any malicious behaviour from the participants will be tracked and can be punished.

In secure cloud computing, cloud provider could use authorisation and authentication to protect cloud system [34]. The authentication is based on accepting the right user or system request. Therefore, verifying the identity of users or systems is the objective of the authentication. The authorisation is based on users privileges that provided by the authorisation. Therefore, security policies are offered by the authorisation.

In cloud computing, the host provides a trusted computing platform to guarantee the privacy and security of computations done on the platform [35]. The providers should create a secure environment for the customer's VM to guarantee their data privacy and integrity. Therefore, trust platform model ensures that only the customers can access their data and the administrator has no access to any of the customer's secured data and cannot damage its contents. Secondly, Trusted Cloud Platforms allows the user to believe that his computations are running on a trusted platform by validating whether the VM is operating on a trusted implementation or not.

Different perspectives establish different trust platform model for cloud computing. For instance, the Cloud Security Alliance created a trust platform model to the cloud. Moreover, [36] has created a trust platform model which consists of trust roots and chains. The following figure 7 describes the component details for the trust platform model. First, it has a temporary storage unit, which is Volatile Storage. A trust platform model generates public keys by using the Key Generation and random numbers by using RNG (Random number generator). Then, Anonymous Identities are responsible to ensure secure operation from a trust platform model. Trust platform model also provides a special type of encryption by using RSA and registration by using PCR [6]. Finally, trust platform model protects open/close, enable/disable and activate/inactivate commands by using a mechanism provided by an Opt-In component.

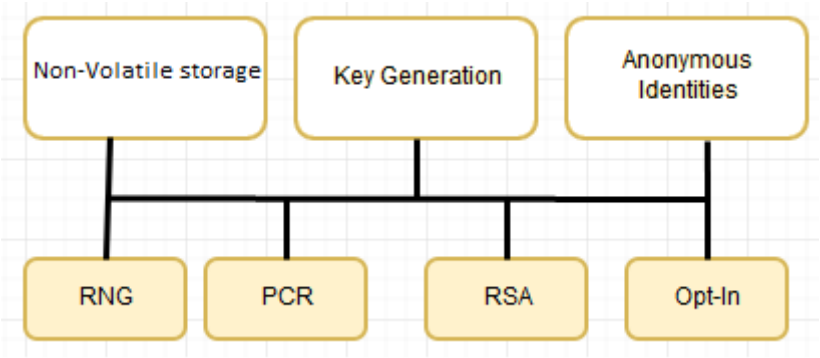

Figure 7. Trust Platform Model.

Data Security in the cloud is based on Trust cloud platform which is also built on the Trust Platform Model. Authenticated boot and encryption are provided by Trust cloud platform and designed to work together. Trust cloud platform provides an authenticated boot service, which monitors the OS software and applications. The components in the system (both hardware and software) are monitored by trust platform model, which ensures that each computer will report its configuration parameters in a trustworthy manner. Trust platform model can interact with other system modules through Trust Platform software stack (TSS). In Trust cloud platform, different entities can appeal to join the cloud-computing environment; therefore, the cloud system administration should verify the entities' identities. However, users and resources from different sources represent a large number of entities, which could be incorporated and need to be authenticated.

The Trust cloud platform is based on the logical independence of a private master key to provide protected hardware via the trust platform model, this can also protect the information stored in the cloud. The trust root can be provided to users by the trust platform model because this will have the hardware certificates; this can be used to generate session keys with other co-operators by using the unique sub key when it wants to join. The session key will be not useful when the configuration in the local machine is changed. This also makes it easy for the cloud computing system when tracing the users because it will have full information about their identity and platform. This system uses trusted authentication service client to trace the client. The server could identify the client through the encrypt information sent by the client to the server. Then, the identity of server determined by the client. Virtual server obtains service information from the server to identity information.

According to [37], secure communication and network are also very important to build secure IaaS. Firewall is one technique to make the network secure, which can reject or accept the outgoing or incoming traffic from and to the device or network. The firewall will evaluate all incoming connections and an unwanted connection or attack will be discarded using Stateful Packet Inspection (SPI). Based on the definition above, most providers offer a firewall such as Amazon's Elastic Compute Cloud (EC2), Microsoft's Azure Cloud Platform and Google's Compute Engine [38]. User configurable firewalls are provided by EC2 and Azure as a security group, which offers a basic level of security. Comparison of firewall provision from different providers as shown in Figure 8.

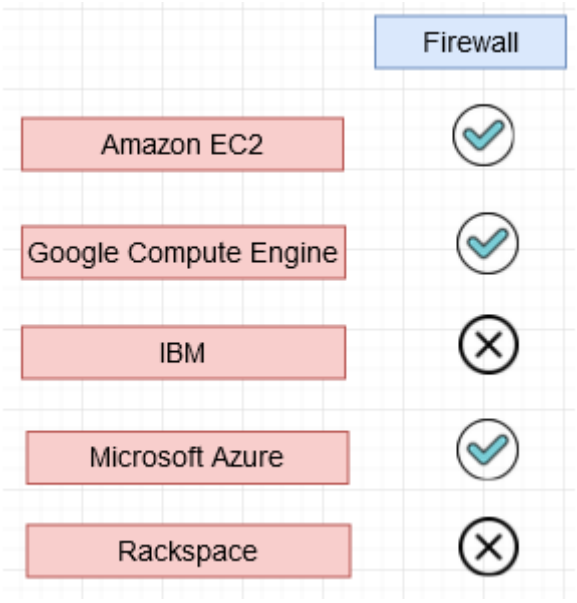

Figure 8. Default Firewall setting from major cloud providers.

Moreover, there are other techniques to build a secure network such as IDS and IPS [37]. Intrusion Detection System (IDS) is a system that analyses all the traffic passing over the network (including the payload) for anomalies. The drawback in this system that it does not perform any reaction to the virus or any malicious content of behaviour, but just sends a warning to the network administrator. IDS detection is typically based on information that is saved in a database. Intrusion Prevention System (IPS) extends this to detect threats and then execute a specific reaction to the threats. There are two types of IPS, Host Based IPS that protects a single device and Network Based IPS which is typically hardware located between router and switch. Recent studies in cloud computing proposed that a collaborative intrusion detection and prevention system (IDPS) working with a hybrid detection technique can make IaaS as shown in Figure 9 [39]. This model consists of Intrusion Detection and Prevention System functions based on distributed IDS and IPS.

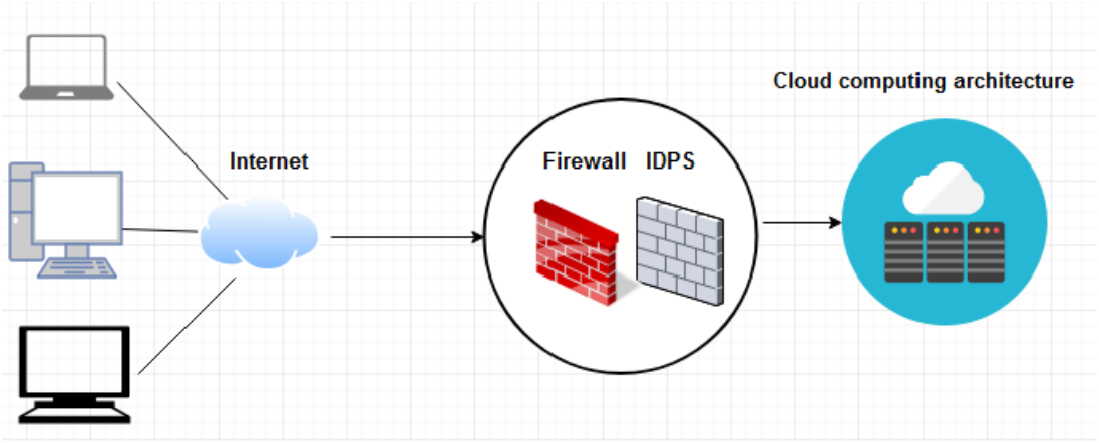

Figure 9. A collaborative intrusion detection and prevention system.

2.4.2 Securing PaaS

The authors in [5] state that current platform services such as Google App Engine and Microsoft Azure provide extensive secure services for users. These services allow cloud users to make use of different applications

and software tools that are in-built into the system. The trust for PaaS service providers lies in securing the VMs of cloud users as well as maintaining good relationships in order to establish trust among the cloud users and service providers.

Confirming the truth of an attribute of a datum or entity is primarily provided by authentication. This manages the access permissions to authorized users only and restricts the unauthorised users [40]. This authentication is important and complicated because a large number of entities require legitimate access, such as users and resources from different sources in the cloud computing. Trust cloud computing platform helps to process the authentication in cloud computing. For example, Windows Azure uses Multi-Factor Authentication that helps to limit unauthorized access to applications in the cloud. Multi-Factor Authentication is a collection of authentication methods from independent categories of credentials to build a security system to verify the user's identity for a login or other transaction. Therefore, Multi-Factor Authentication provides more secure access and data protection. Multi-factor authentication also commonly serves as the identity provider for secure Web services, such as with Google Docs or Salesforce cloud apps. A good example for authentication is Google Sign-In, which is based on the OAuth 2.0 and OpenID Connect protocols. Web Apps, iOS, and Android can use Google Sign-In services. OpenID provides information about the user, therefore it is responsible for authentication. OAuth is used to grant access to data, application and resources, which means that it provides authorisation.

The OAuth protocol allows managed access control based on a two-step process, the authorisation and the authentication processes [41]. This model has a security manager, which grants or deny user access through the service provider. The security manager has a local database that stores the user ID, as shown in figure 10.

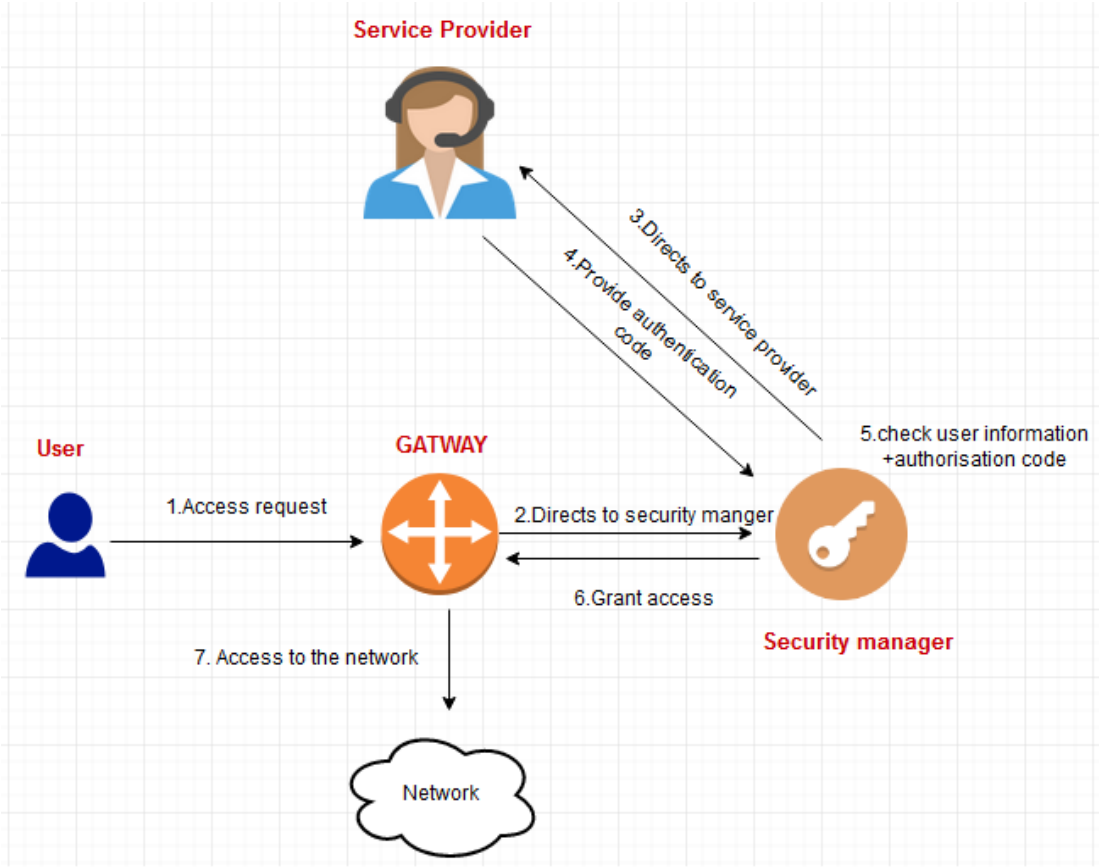

Figure 10. OAuth protocol.

According to the above figure, first, the user contacts the security manager when they access. The security manager contacts the service provider about the user access request to get the authorisation code. Then, the security manager verifies the client id and authorisation code and uses an access token which is granted by the service provider to access the user information by performing the API call. It compares the user ID, which is obtained from the service provider, with the list of users' IDs in the local database and determines whether it should accept or deny the user request.

The security levels in cloud computing could be further improved by using stronger authentication and access control processes. The authors in [42] suggest that Multi-factor Authentication based on Multimodal Biometrics called MFA-MB could make the cloud-computing model more secure. FaMSL-MBS, which is the proposed system, depends on a number of traits, sensors, and feature sets used, a variety of scenarios are possible in an MBS. Figure 11 shows the FaMSL-MBS scheme.

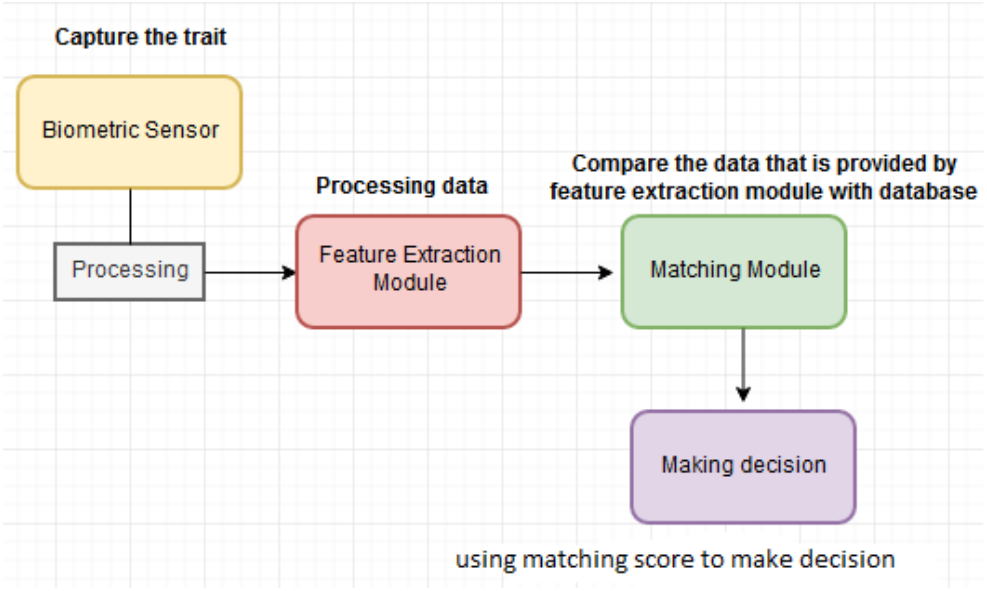

Figure 11. FaMSL-MBS scheme.

In this scheme, a user's password, and arithmetic allow the creation of a random projection of biometric data using a random key. A classification of practical multiple biometrics for FaMSL-MBS was established in the following table 1.

Table 1 Classification of practical multiple biometrics technology for authentication

Bimodal biometrics	Trimodal biometrics	More than trimodal biometrics
Face-Fingerprint	Face-Fingerprint-Iris	Proposed trimodal biometrics remain can be combined in this case
	Face-Fingerprint-Signature	
Face-Iris	Face-Iris-Fingerprint	
	Face-Iris-Signature	
Iris-Signature	Iris-Signature-Fingerprint	

FaMSL-MBS scheme has several biometric sensors that responsible to capture the trait. Feature extraction module is responsible to processing the data. The third step is matching module that is compare the data that is

provided by the feature extraction module with the database. Finally making the decision step that is using the matching score to make the decision.

To further secure PaaS, providers could use some approaches to limiting system access to authorized users such as Role Based Access Control. Here, there are several classes/groups for the users who hope to access cloud computing service that are categorised according to the type and level of access required [6]. The different between authentication and authorisation is the authentication is confirm the user identity while the authorisation is provide permission to the user to allow or deny access to the resources in the system.

2.4.3 Securing SaaS

In SaaS, the main aim is to protect data stored in clouds which is dependent on respective technologies, policies and controls [43]. Identification of assets that are vulnerable to attack, identifying potential threats and countermeasures, and examining existing policies to counter the threats are the most important priorities for the security model. Assets in cloud computing are; customer data, applications, and computing devices and these all require the service provider to assure confidentiality, integrity, and availability on them. Apart from that, [5] also reported that SaaS platform service providers such as Microsoft and Google secure their platforms by ensuring that data stored on the system is protected from any form of hacking activities that could lead to loss of data. Other useful options to secure SaaS platform is the encryption of data and colouring options, which are supposedly useful in retaining the privacy and security of the user's data.

One of the key problems identified in terms of SaaS trust is the fact that the security solutions given by the service providers are often too flexible, which leads to security issues and then creates lower levels of trust. On the other hand, overly complicated security measures have also proven to be difficult for cloud users to adapt to whilst supporting their customers. Therefore, this leads to the problem of creating a balance between the issue of security measures and user-friendly methods that would make it easy for cloud users to access and use the system.

There is also still a vague understanding by users pertaining to what happens to their stored data, should the cloud service provider have some technical problem with the storage system. Although encryption is the common approach to protect data and information of individual users such as Blowfish and RSA, there is still no guarantee of the safety of the data [6]. Encryption and decryption provided by Blowfish is based on a secret symmetric key system. Blowfish has four SBox arrays that have 256 independent keys, and one PBox which contains 18 32-bit keys. Each input block is 64 bits long and the key is 448 bits long. Key expansion and data encryption units are the main parts of the Blowfish algorithm with each round implementing the Feistal (F) function, as shown in Figure 12.

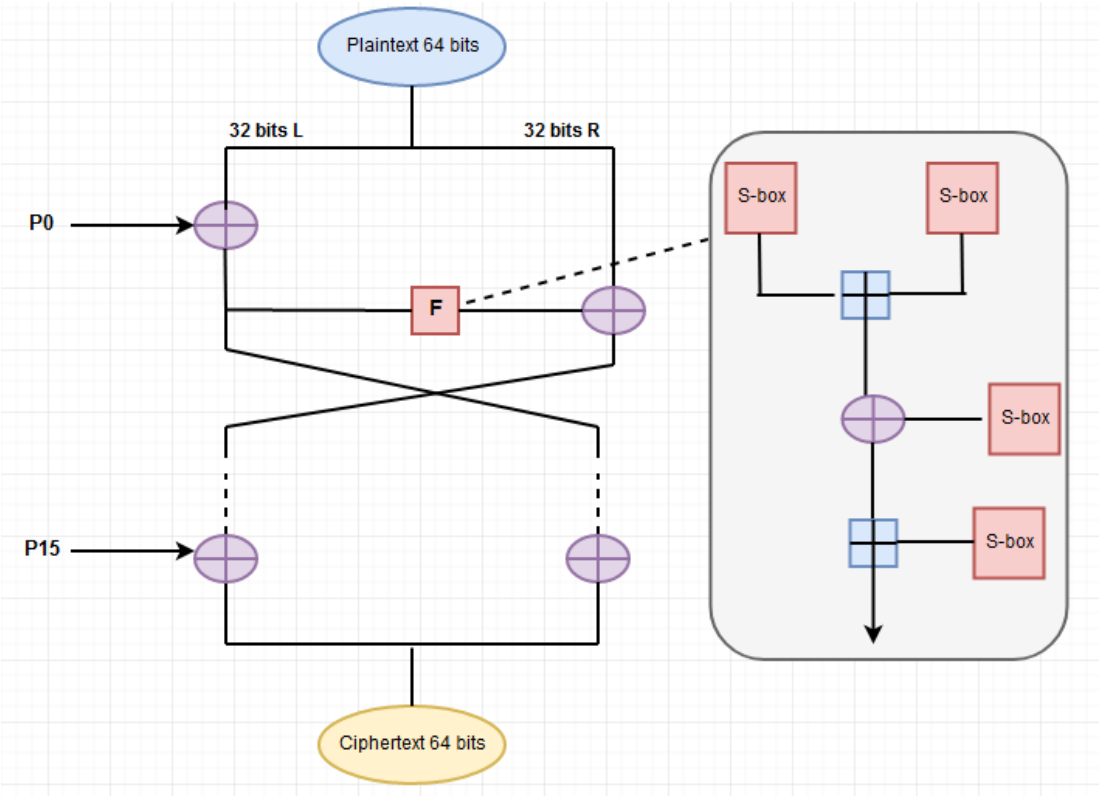

Figure 12. Blowfish Algorithm with F Function.

According to [44], a data colouring model could also help to create secure user data in SaaS. A data colouring model consists of an expectation (Ex), entropy (En) and hyper entropy (He) characteristic. The data owner provides Ex while En and He are produced by negotiation with the data owner and service provider. A data colouring model does not use conventional encryption and decryption in the process of data colouring. Images could be an example for data colouring where each image can be shown as a pixel matrix with values in the grey-level ranges between 0 and 255. The image in figure 13 will be obtained after traversing the matrix line by line from top to bottom.

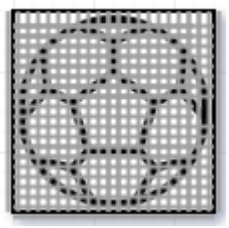

Figure 13. User images shown as a pixel matrix.

In this example, certain images will be transferred to uncertain print drops, which will affect the discernment of the paint by En and He. Figure 14 shows different paint drops according to different En (set He = 0). In addition, it is apparent that the paint is relatively clear when En 6 0:1; on the contrary, the paint becomes unclear with atomization.

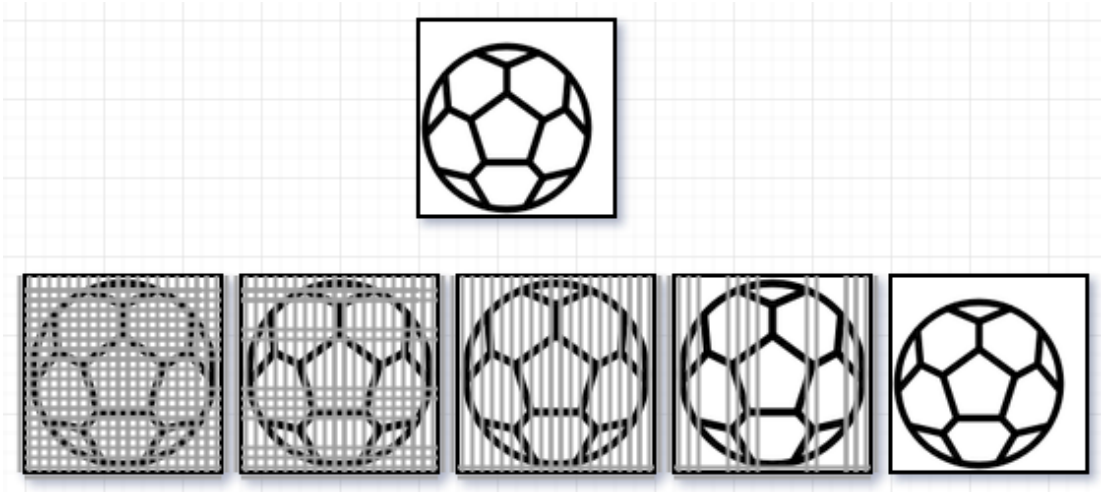

Figure 14. Print drop with a different En.

2.5 Evaluation of Trust Techniques in Cloud Computing

Cloud computing has significant advantages but Security, Privacy and Trust (SPT) are concerns that resulted from consumers' loss of control over their confidential data since they outsource it to the cloud with no knowledge of storage location or who is accessing and maintaining it [6]. The author suggests that Trusted Cloud Computing Frameworks should be built to overcome cloud computing problems such as security, privacy and trust. Trusted Platform Module (TPM), Virtual Trusted Platform Module (VTPM), Self-Encrypting Drives (SEDs), Trusted Network Connect (TNC) and Trusted Software Stack (TSS) are the main components for Trusted Cloud Computing Framework (TCCF). This section further explores some of the previous technical developments related to building trust in cloud computing.

2.5.1 Reputation Systems

Reputation-Guided Data-Centre Protection

The authors build a reputation system to protect cloud platform resources or user applications on the cloud [5]. Scalability and reliability are very important for handling failures and these features can be provided by Distributed reputation systems that help providers to create content aware trusted zones. The authors support trusted cloud services by implementing a two-layer trust-overlay network to model the trust relationships between data-centre modules to build trust between service providers and users using a distributed hash table (DHT). DHT helps to achieve fast aggregation of the global reputations from large numbers of local reputation scores. The bottom layer, which is the trust overlay for distributed trust negotiation and reputation aggregation, handles user or server authentication, access authorisation, trust delegation, and data integrity control. A content-poisoning technique is also presented for copyright protection in P2P networks. This protection scheme could be extended by the authors to stop copyright violations in a cloud environment surrounding multiple data centres. High level security features like worm signature generation, intrusion detection, anomaly detection, DDoS defence, piracy prevention, and so on is dealt with in the upper layer, as shown in Figure 15.

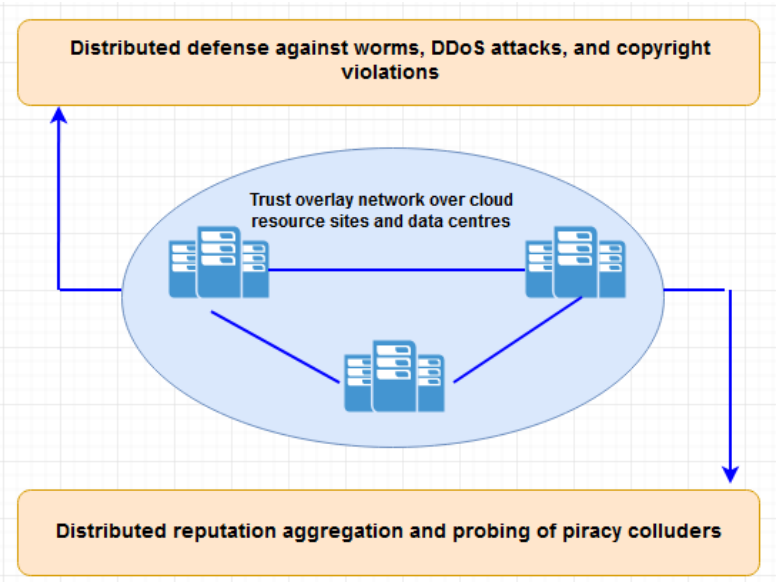

Figure 15. Distributed-hash-table (DHT)-based trust-overlay networks.

Reputation Control

Achieving the highest profit and maintaining a good reputation are the main goals for each node in the harmony design [45]. The resource price for nodes can be adjusted adaptively to achieve the desired profit and maintain a good reputation while ensuring that the node is not being overloaded. This can help ensure efficient utilization of resources in the system. The design of components has three motivations, which are Integrated Multi-Faceted Resource/Reputation Management, Multi-QoS-Oriented Resource Selection and Price-Assisted Control. Together, these constitute the three key innovations, which are incorporated by Harmony as shown in Figure 16. In Harmony, the provider can control the price attribute only and avoid overloading nodes, which strengthens the cooperative.

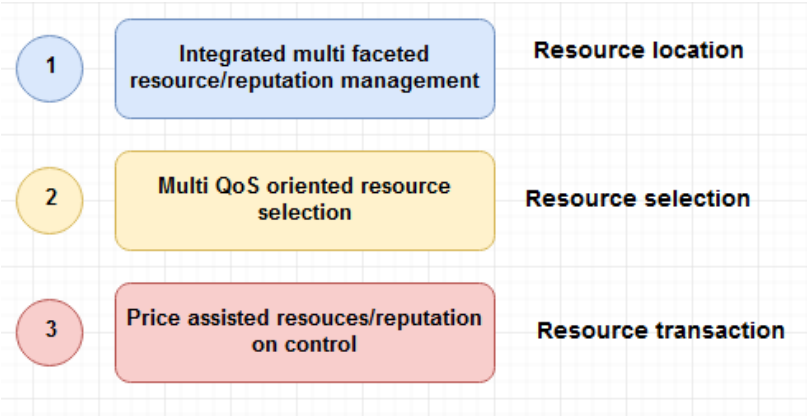

Figure 16. Harmony components in resource market stages.

2.5.2 Trusted Cloud Computing Platform to protect VMs

Trusted Cloud Computing Platform

Trusted Computing (TC) functions can be supported by some new hardware which is added by manufacturers to each computer [46]. Therefore, a combination of software and hardware can help trusted cloud platforms to operate. Authenticated boot and encryption are two basic services that are provided by Trust cloud

platform and are designed to work together. The authors propose building a trusted cloud computing platform (TCCP) to address the confidentiality and integrity of user data. Through this design, Infrastructure as a Service (IaaS) will be able to provide a closed box execution environment that guarantees confidential execution of guest virtual machines. In addition, IaaS providers will be testable by clients who can determine whether the service is secure before they launch their virtual machines.

TCCP was designed to build a trust-based cloud service that employed the use of IaaS based service platforms such as Amazon to incorporate a closed environment. IaaS service providers allow customers to access and use remote VMs and data storage on an infrastructure managed by a third party such as Amazon. One of the difficulties of addressing the issue of security in IaaS is that service providers often have to access or manipulate customers' data, which makes it difficult for providers to guarantee safety. This design enables IaaS to provide a closed box to maintain the confidentiality and integrity of the user data. With this trust-based design, cloud users can now test whether the IaaS service provider is trustable and whether the connection is secure before they log onto their own systems and services. Furthermore, this TCCP system also shows a promising impact towards securing private and confidential information in different user based VMs.

TCCP aims to provide a closed box to better enforce user data confidentiality and integrity. The closed box consists of a set of trusted nodes (N), a trusted coordinator (TC) for external nodes, a trusted virtual machine monitor (TVMM) and the cloud manager (CM) to provide services for the users (Figure 17).

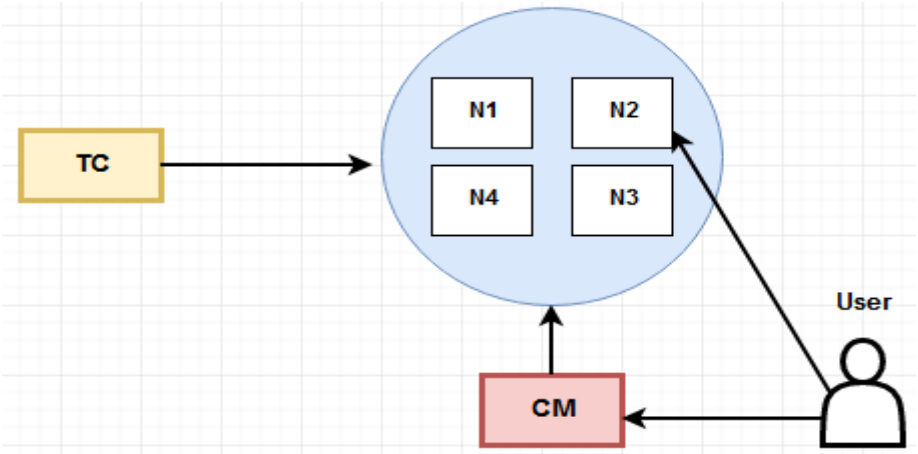

Figure 17. Set of trusted nodes (N) and the trusted coordinator (TC).

In this design, the clusters are managed by a cloud management system, Eucalyptus by a set of components, which are in a single cloud manager (CM) that handles a single cluster. The TVMM is run by a trusted node and prevents privileged users from inspecting or modifying them. The TC manages the set of nodes that can run a customer's VM securely.

Cloud-Trust Security Assessment Model for Infrastructure as a Service (IaaS) Clouds

In this study, the author presents a cloud architecture reference model to quantify the degree of confidentiality and integrity offered by a cloud service provider (CSP) [47]. This model consists of a wide range of security controls and best practices, and a cloud security assessment model. They define a Trust Zone (TZ) which is implemented using physical devices, virtually using virtual firewall and switching applications, or using both physical and virtual appliances. Access management controls (IAM) are responsible for making access

decisions based on an access control list (ACL). Subnets, firewalls, domain controllers, and internet access points are used to isolate cloud service provider management and security servers from cloud tenant VMs which are networked using a software defined network (SDN) shared by all cloud tenants.

The CCS reference model is shown in Figure 18. There are six elements in the CCS model which are the CSP TZ, tenant TZs, contain cloud management servers, SDN controller servers, CSP tenant IAM servers, and CSP Information System Security System (IS3) servers. CSP TZ is isolated from other elements. Firewall and Internet ports isolate CSP communications traffic and allow CSP admins to communicate with the CSP management system to isolate CSP management from cloud tenant VMs. Suspect data flows or configuration changes and automated software distribution systems rapidly patch OS installations and applications are identified by Network performance monitoring tools such as Netflow, and log file analysers.

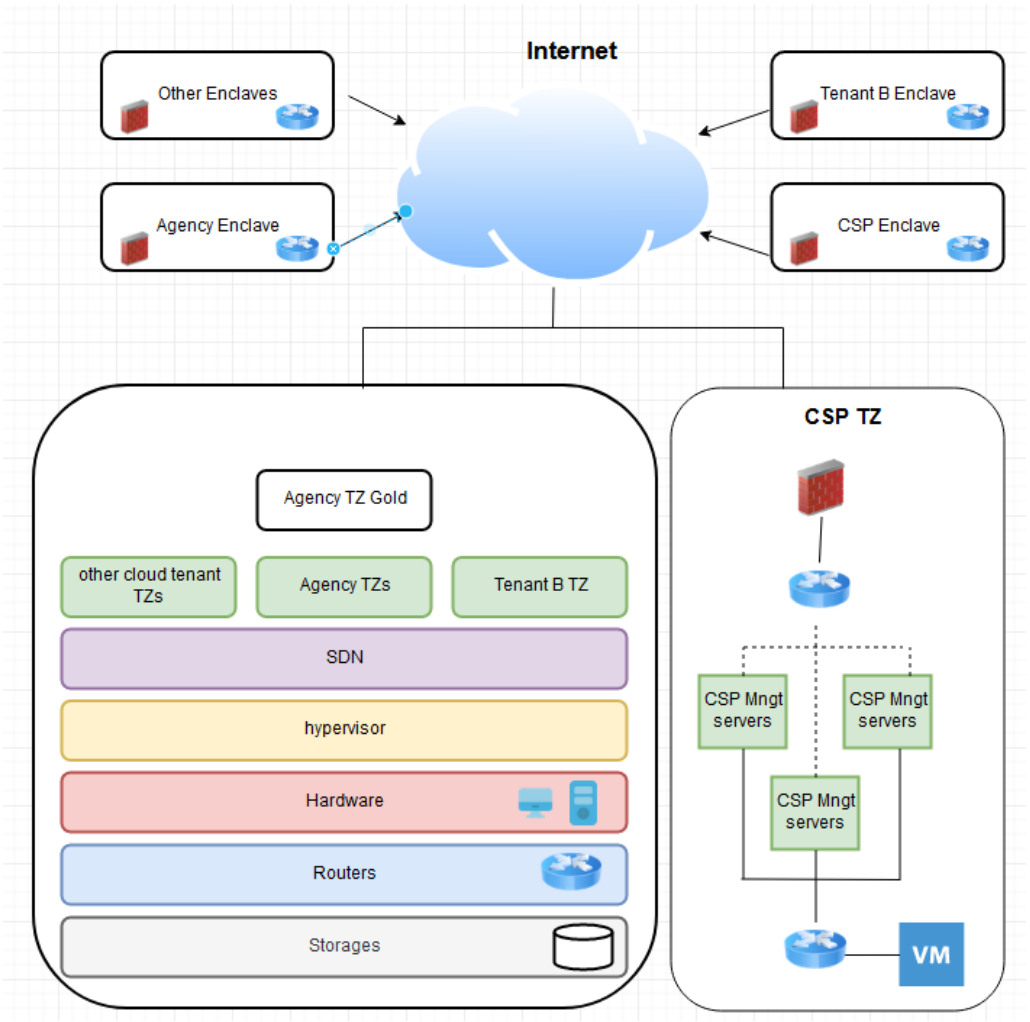

Figure 18. CCS reference model.

Security controls for four cloud architectures based on the reference model are shown in Figure 19. SDN is used for tenant VM networking by all four architectures. In the first architecture, the cloud data centre can be entered by any CSP employee, who can access the data centre through electronic access control cards and their movements are tracked in the data centre. In the second architecture, two factor authentication must be used for CSP admins to login to CSP management servers and they must sign in as named local users and not as root. The

second architecture has the same features as the first with some added features such as additional data centre physical access, CSP admin authentication, and server hardware port controls. Also, in the second architecture, the data centre is not permitted to the CSP sys-admins. These security controls in the second architecture are the same in the third architecture. Agency admin and regular users have these security controls when they want to login. Time sensitive two factor authentication methods are responsible for performing the access for all Agency cloud users to Agency VMs. Additional cloud infrastructure hardening measures are adding to the fourth architecture. Encrypt VM image in storage which are monitored for access attempts, image changes, and TZs are isolated using more robust measures.

	VM Images At Rest	VM Migration	CSP Sys-admin IAM	Data Center physical security	Hypervisor, BIOS, CPU	VM Isolation	Tenant IAM	App. White-listing
Cloud Arch 1	Not encrypted	Unencrypted memory pages and packets	Single factor	All CSP employees have access	HV, BIOS not signed CPU without TPM	No network, CPU isolation	Single factor	No
Cloud Arch 2	Not encrypted	Unencrypted memory pages and packets	2 factor – time limited token code	CSP employee access limited & controlled + USB server ports disabled	HV, BIOS not signed CPU without TPM	No network, CPU isolation	Single factor	No
Cloud Arch 3	Not encrypted	Unencrypted memory pages and packets	2 factor – time limited token code	CSP employee access limited & controlled+ USB server ports disabled	HV, BIOS not signed CPU without TPM	No network, CPU isolation	2 factor – time limited token	No
Cloud Arch 4	Encrypted at rest + file access monitoring	Encrypted memory pages and packets	2 factor – time limited token code	CSP employee access limited & controlled+ USB server ports disabled	Signed HV, signed BIOS CPU with TPM	Virtual PANs, temporal CPU isolation	2 factor – time limited token	Yes

Figure 19. CCS Architecture security control for four architecture based on the reference model [48]

2.5.3 Third party verifaction mechanisms

Trust Management system architecture

The authors of [48] develop a new Trust Management system architecture that caters specifically for cloud service providers and users as depicted in Figure 20 below. The proposed multi-faceted trust system can efficiently assess between the different levels of service quality offered by providers, making it a useful broker system. Moreover, the system also focuses on creating a trust score, which is largely based on different factors that are chosen by the customers. This allows customers to be selective according to their specific trust needs. Furthermore, the system will increase transparency between potential cloud users and existing cloud service providers, which is key in ensuring that the cloud market grows.

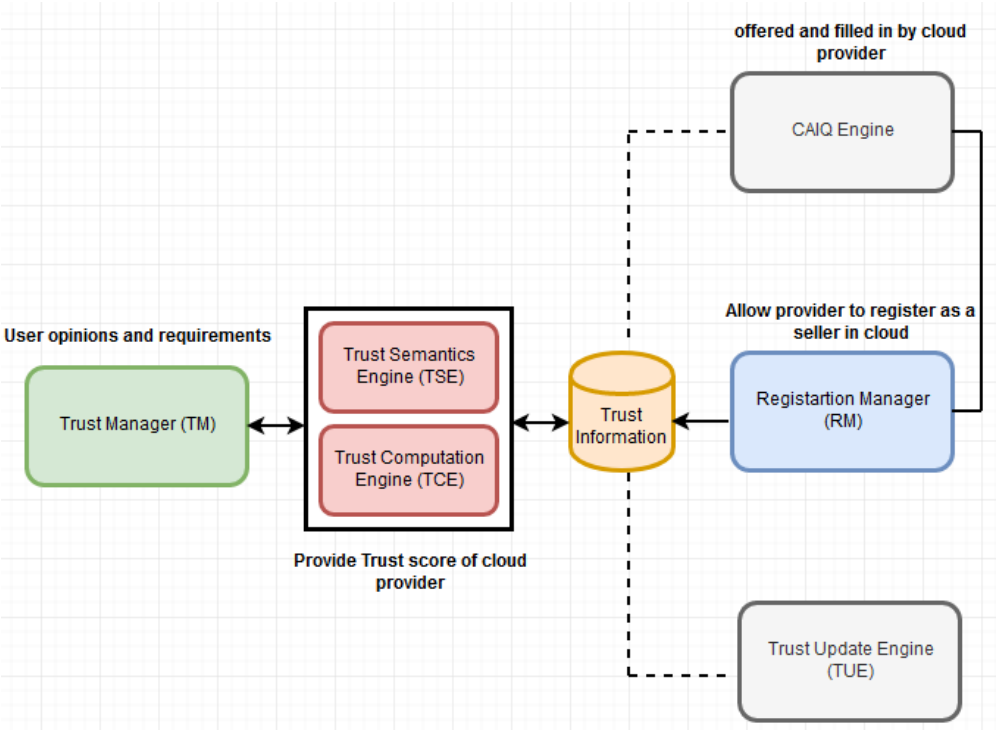

Figure 20. TM System Architecture.

According to Figure 20, there are necessary tools for assessing, representing and computing trust. The TM system architecture consists of six components which are Registration Manager (RM), Consensus Assessments Initiative Questionnaire (CAIQ) Engine, Trust Manager (TMg), Trust Semantics Engine (TSE), Trust Computation Engine (TCE) and Trust Update Engine (TUE).

RM enables Cloud providers to register to act as sellers in a cloud marketplace. A Consensus Assessment Initiative questionnaire (CAIQ) should be offered and filled in by the cloud providers and the RM will forward the answers of the questionnaire and system/service description to the CAIQ engine and TI (Trust Information) respectively for further processing.

The user's requirements and opinions will be specified through the TMg when accessing the trust score of cloud providers. The user can interact with a web-based front end to specify their requirements. The trust Semantic Engine (TSE) and Trust Computation Engine (TCE) help the TMg to provide the trust score of cloud providers Based on the user's requirements and opinions. The behaviour of a cloud provider in terms of a specific attribute is called propositional logic terms (PLTs). The PLTs are configured by The TSE models. The CAIQ answers, which are stored in the repository (TI), are the basis of a default configuration of PLTs. Furthermore, the TSE should convert all trust relevant information into PLTs which are evaluated by the TCE.

The opinions from various sources and roots about the trustworthiness of cloud providers are collected through The TUE. Through this engine the user could use the valid opinions according to their requirements because all the opinions that are collected here should be filtered. For instance, the TI repository is responsible for storing junk or useless information and then updating the trust value for cloud providers according to the filtered opinions.

Secure Dynamic Data Support and Trusted Third Party Auditor In Cloud Computing

In this study, the author proposes a flexible distributed storage integrity third party auditing mechanism. By using the token keys and privacy preserved sharing of keys it improves user data privacy, integrity and availability as well as achieving fast data error localization and ensuring strong cloud storage correctness guarantee [49]. The User, Cloud Service Provider and Third Party Auditor Users are the three main elements for this cloud architecture. Users have a large amount of data stored in the cloud and Cloud Service Providers (CSP) provide these services, the Third Party Auditor (TPA) is used to store verification parameters and offer public query services for these parameters. Figure 21 shows that Cloud Storage Architecture.

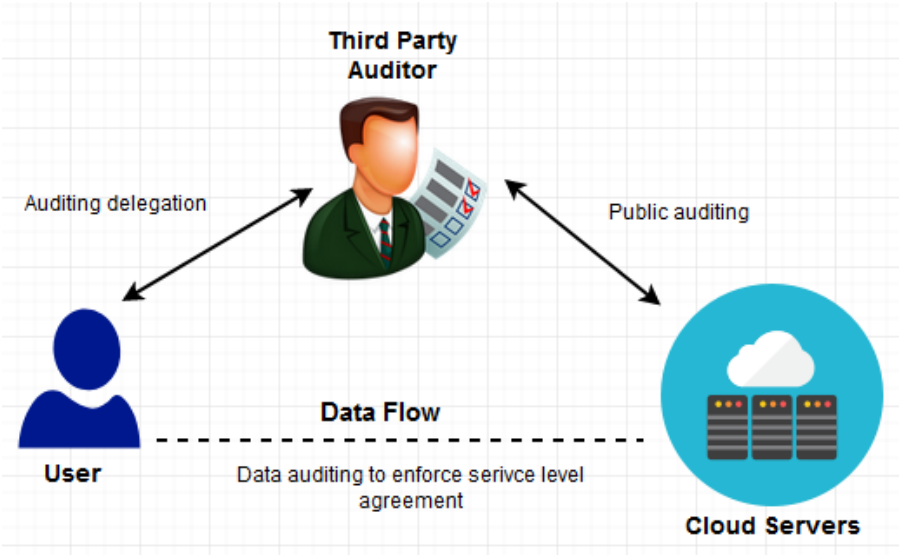

Figure 21. Cloud Storage Architecture.

2.5.4 Data security and integrity

Cloud Computing Data Storage Security framework relating to Data Integrity, Privacy and Trust

In cloud computing, there are severe security concerns leading to wide adoption of cloud computing such as data privacy, integrity and trust issues [50]. The encryption and decryption approach facilitating the cloud user with data security assurance are the main idea for the framework in this study. Therefore, this model could help to reduce data security threats in the cloud environment because it provides transparency to both the cloud service providers and the cloud users.

Cloud provider should protect user privacy. For example, malicious users could get some personal information about other users without a permission and change the privacy policies for these users. Data Privacy can be defined as the ability a user has to isolate their personal data and information from other users. The reliability, validity and uniformity of data is also very important in cloud computing and these features could be provided by the cloud provider to improve the users' data integrity. Through data integrity, the users will know about any modification or tampering of their data. Data modification attack, data leakage attack and tag forgery attack are the attacks, which could be carried out on the user data. In the cloud computing, there are several mechanisms to make the users' data integrity such as cooperative provable data possession (CPDP) which is the combination of hash indexing hierarchy and Homomorphic verifiable response. Homomorphic Encryption is encryption method that allow compute on data while the data is encrypted. The encrypted data does not need to

decrypt by the cloud provider to apply functions. Public key use in Homomorphic Encryption to encrypt the data and private key need to decrypt the data.

This framework provides data security through three layers, which interact with each other, as shown in Figure 22. User authentication is provided by the first layer, which allows the cloud service provider to use authentication methods for verifying the genuine user. Enhancing the data privacy and data integrity in cloud computing will be provided by the second layer which communicates with the previous layer to make sure that only an authorized user can send and receive the data. The proposed data security uses Proxy Based encryption and Homomorphic Based encryption methods to achieve a practical preferred solution for data security in cloud computing. Protecting the data before storing the data in the resource pool is the job of the third layer. Real time monitoring, forensic virtual machine analysis and data masking are features in this layer, which are used for the protection of users' data.

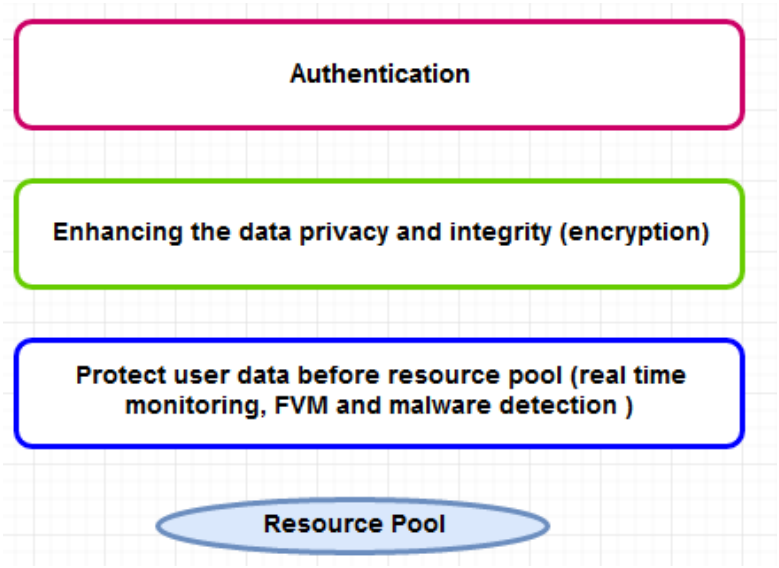

Figure 22. The proposed Data security framework in cloud environment.

2.6 Container orchestration platforms

This section will review Container management and orchestration platforms in order to evaluate how they are currently managed and secured, particularly in the context of the previous work in this chapter on cloud trust and security in general.

2.6.1 Docker

Two main fundamental units make up the Docker which are Docker Hub and Docker Engine [51]. Docker Hub is for downloading and sharing images of Docker and its work as a Software-as- a-Service (SaaS) platform. Docker Engine is an open source tools that is a solution for virtualisation. Docker hub work as a Master Storehouse for all the digitally signed images for security and can be exchanged by users because sharing and downloading images might be both public and private. Docker engine is a packaging tool, which helps to create and run Docker containers. The virtualisation that is based on containers has the same Docker engine architecture. Each container in Docker is run and managed by the unit called the daemon, and RESTful APIs help the Docker client to connect

to containers through a UI. The Docker daemon on the host allows the Docker client to execute a specific task on the containers. Figure 23 below shows the Docker Security Model.

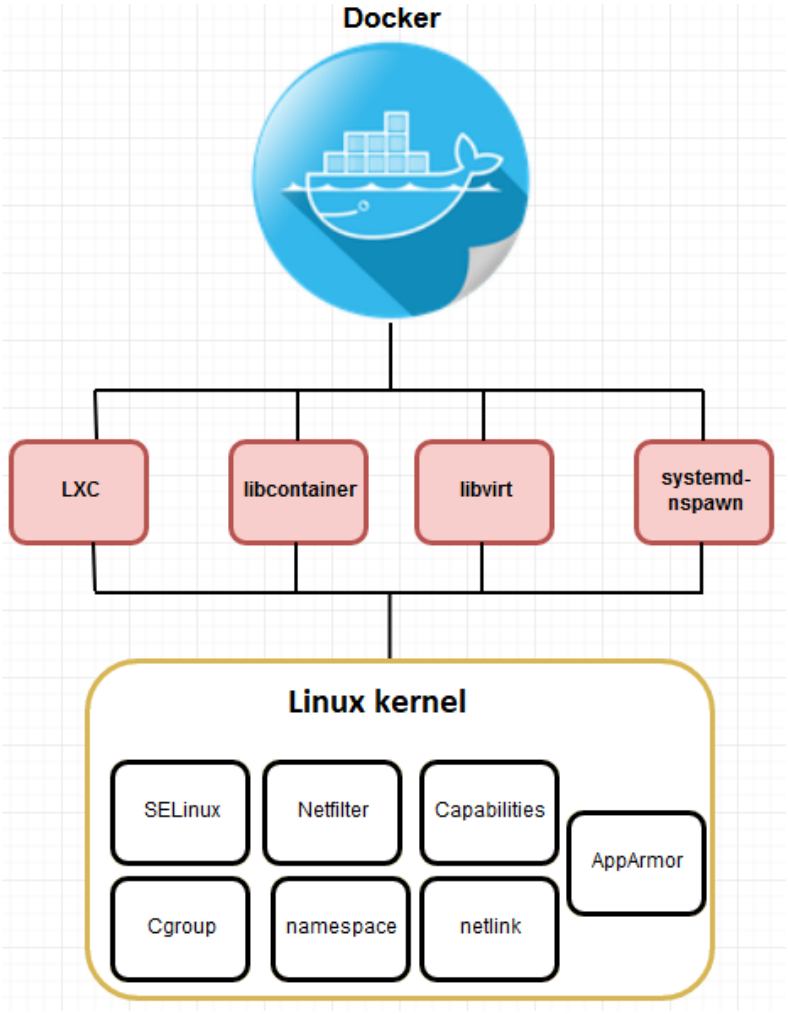

Figure 23. Docker Security Model.

The focal point of all communication to and from containers is the Docker daemon [52], which runs on the host machine and needs root privileges. The Docker daemon helps to run a container where the host directory will be free to operate without any restrictions, thereby, sharing the directory among the Docker host and guest clients provided by the Docker daemon. Figure 24 below shows how the Docker daemon allows the users to communicate with the containers.

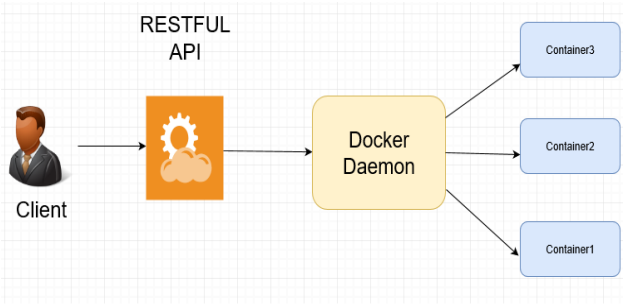

Figure 24. Docker daemon.

2.6.2 Kubernetes

Kubernetes is a cluster management tool that is developed by Google for managing containerised applications [53]. Container applications can be deployed in the public cloud and private cloud on one big computer that is consists of a set of nodes managed by Kubernetes. Clusters have applications determined by the Kubernetes master node. Container-based applications can use Kubernetes to assist in the automation of the utilisation, management, and scaling of applications that have been containerised.

Docker containers can be managed by Kubernetes which is an open source tool cluster manager [54]. Containerised clusters could be managed by Kubernetes that based on the CAAS (Container as a service) level. The basic deployment unit in the Kubernetes is Pod that can be both operated and managed. Kubernetes consists of three main elements that are master, node and Etcd. The master node, Container arrangement, and management in the cluster is the job of the master node. There are three main components related to the master node, the API server, the Scheduler, and The Controller. The entrance to all services is the API server. Scheduling the application to the node is the job of the Scheduler; finally, the state of the application is managed by the controller in Kubernetes Node is the second element in Kubernetes. The lifecycle, volume and network management of containers in the cluster are managed by the Kubelet, according to the node. The state of the entire cluster is stored in the Etcd that keeps the applications in the desired state. Figure 25 below shows the Kubernetes architecture.

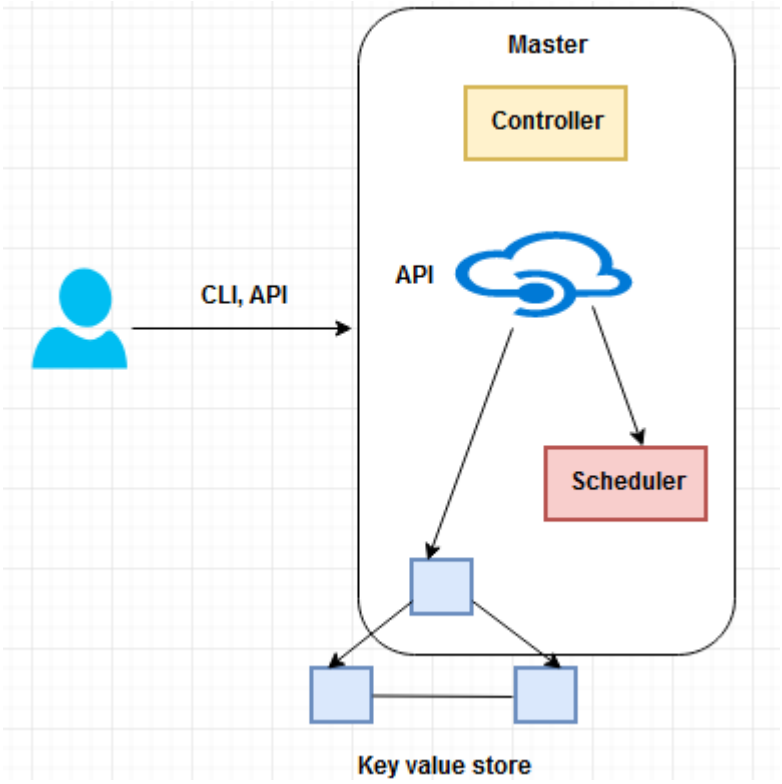

Figure 25. Kubernetes architecture.

Kubernetes has several features [55]. Automatic bin packing is the application packaging and container scheduling based on the requirements. Balancing between critical and best-effort workloads are provided by Kubernetes to ensure complete utilisation and to save unused resources in addition to managing the network and

communications role of the Kubernetes. A means of reduction of traffic inside the cluster is provided by the Kubernetes that are automatically assigned IP addresses to containers and a single DNS name for a set of containers. Besides, the Storage system could be chosen by the users through Kubernetes so the user or developer can have local storage and public by a select public cloud provider such as GCP or AWS.

Kubernetes has various features to improve security in the cluster, including Self-Healing. The containers that do not respond to user-defined health checks could be automatically killed by Kubernetes. Furthermore, containers that fail during execution can be restarted automatically. Kubernetes and Docker Swarm are both open-source platforms that are vital for the orchestration of containers to assist in the automation of the utilisation, management, and scaling of applications that have been containerised[56]. Both platforms have various features to enhance security and trust in the handling of applications. Kubernetes security features include application monitoring and auditing, and the ability to integrate and deliver systems continuously and frequently [57]. Kubernetes also scans container images and has a declarative and immutable design to keep users informed and avoid errors. Moreover, Kubernetes is capable of supporting higher and more sophisticated demands and is therefore preferred by Internet organisations with heavily used services [58]. Kubernetes uses a flat model for networking, using an overlay method, which allows interaction between all its pods. User policies are available to guide this interaction. This model requires multiple CIDRs (Classless Inter-Domain Routing), one to provide IP addresses for the pods, and the other for the services. Kubernetes allows the distribution of all its pods among its nodes and tolerates application failure, enhancing its accessibility.

Kubernetes authentication is enabled by establishing TLS certificates [59]. The authentication system obtains the client certificate to grant access to the cluster. Client certificates, bearer tokens, an authenticating proxy, or HTTP basic auth are used by the Kubernetes to authenticate API requests through authentication plugins. The authentication plugin will allow or deny user access to the cluster. The client certificate authentication should be enabled. The API server should have the --client-ca-file=SOMEFILE to enable the client certificate authentication [60]. The authorised certificate for the users should be in the references file to validate client certificates presented to the API server. When the user logs in to the system, the Kubectl uses the user details adding them to the kubeconfig. The kubectl sends the user details of the authentication system, then the authentication system decides if the user is authorised or not.

Four modes are used by the Kubernetes API server to authorise the request. Node, Attribute-based access control (ABAC), Role-based access control (RBAC) and Webhook. Node is an authorisation mode that is used by the kubelets to grant permissions based on the pods they are scheduled to run. Role-based access control (RBAC) is an access control mechanism that allows users to perform some action by using some roles that grant or revokes user access. This method makes data more secure from internal and external security threats. In RBAC, the organisation should apply the action on objects to create permission, which is assigned to roles. Users are then assigned to specified roles, which are a bridge between permissions and users. Thus, the Roles can avail of different permissions. In ABAC, User attributes are the main factor in authorization decisions. Four sets of attributes are evaluated by rules in ABAC. First, person or actor attributes which are Subject Attributes. Attributes of the target or object, which are Resource Attributes. Attributes of the target or object, which are Action Attributes and finally Environment Attributes, which include attributes such as the time of the day, IP subnet. Finally, WebHook is an HTTP callback. A WebHook is an HTTP callback: Webhook is a simple notification when that is provided by the HTTP POST. URL will obtain the message from the Webhook when certain things happen.

2.6.3 Docker Swarm

The container cluster is the number of containers distributed across the cluster [61]. Multiple systems that have containers should be managed and launch as a single body. The challenging issue is managing several containers or creating a single service [59]. Docker Swarm, which is a container cluster management facility provided by Docker, utilises the Container Cluster and acts as the clustering and orchestration tool that should be installed in multiple host machines. Multiple containers are automatically deployed by the Docker Swarm inside the host machine to detect any container failure of the host and automatically redeploys the failed container into another host machine.

Manger and work nodes are two types of nodes in Docker Swarm. The membership and delegation process are managed by the manager node. Swarm services are run on the Docker swarm in work node indicating that the Docker host machine could be in either manager node or worker node. The services IP address and port that are located inside the swarm allow the manager node to expose to the outside world. The request from the outside world will be routed by the manager node to the designated worker node using its internal load balancing mechanism. Monitoring resources utilization of each host machine is not provided by Docker Swarm which can lead to unequal load distribution between the host machines. Table 2 below shows the difference between Kubernetes and Docker Swarm.

Table 2 The difference between Kubernetes and Docker Swarm

Features	Kubernetes	Docker Swarm
Graphical user interface	Dashboard Graphical user interface	Graphical user interface
Scalability	Highly scalable and scales fast	scales 5 times faster than Kubernetes and Highly scalable
Auto-Scaling	Auto-Scaling provided by Kubernetes	Auto-Scaling is not provided by Docker Swarm
Load Balancing	Traffic between different containers in different Pods need to manual intervention to load balancing.	manual intervention does not needed in Docker swarm because it provide auto load balancing
Rolling Updates & Rollbacks	Automatic Rollbacks and Rolling updates could be deployed	Not automatic Rollbacks and Rolling updates could be deployed
Data Volumes	Only the containers in the same pod can share storage volumes.	All containers Can share storage volumes
Logging & Monitoring tools	In-built tools	3rd party tools like ELK

Several debates pit Kubernetes against Docker Swarm, and this section aims to compare the two platforms by investigating their operational differences and their pros and cons [58]. Kubernetes is a product of Google, while Docker is a product of Docker Inc., and Swarm is a built-in function of the Docker Engine that converts a collection of Docker hosts into a single virtual host. Both platforms have similar uses but differ fundamentally in their operation. Docker Swarm is simpler and is therefore preferred by developers looking for simplicity and rapid deployment. In terms of the deployment of applications, Kubernetes uses a blend of micro-services, pods, and deployments. Conversely, in Docker Swarm, the deployment of applications is done as micro-services of clustered

services. The identification of multi-containers is done using YAML files, while the installation of applications is done using Docker compose. In Docker Swarm, the combination of nodes and swarm clusters leads to the generation of overlay and bridge networks for all Docker swarm hosts or specific Docker hosts, respectively. The scalability of Kubernetes and Docker Swarm also differs. Kubernetes scales better and the deployment of containers is slower than Docker Swarm because it is more complex and offers a stronger assurance of the state of clusters and an integrated API set.

Kubernetes and Docker swarm have various security features to enhance the security in the cluster. Docker Swarm allows the replication of services in its nodes, which enhances availability. It has nodes that manage whole clusters and control the resources available to worker nodes. In terms of the set-up of containers, the API that Docker Swarm uses exhibits most of Docker's functionality but lacks some of its commands. When it comes to loading balancing, Kubernetes uses services to expose its pods, which are then deployed in the clusters as load balancers. Kubernetes also typically uses ingress as a load balancer. Conversely, Docker Swarm performs load balancing using DNS components, which are used for the distribution of requests that are inward-bound to the names of services. The services are then automatically assigned or run on user-specified ports. Moreover, Docker Swarm security features include its overlay network, which helps secure communication between containers [57]. Swarm also has tools that ensure the security of container images downloaded from registries such as the Docker Content Trust. Additionally, Swarm has a tool for the management of secrets like keys, passwords, and tokens. Other security features include the Public Key Interface and the ability to detect threats during runtime.

As for authentication and authorisation in Docker swarm, any Docker client comment can be processed by any users by using the Docker API [62]. An authentication system provides access control to the system. Enabling a TLS certificate allows the developer to protect the Docker daemon and allow only particular users to connect with the Docker daemon. The authentication subsystem has two field that is the client certificate subject common name and Authentication Method. The client certificate subject common name is set as user field and the Authentication Method represents the TLS field. Most importantly, the username does not pass to the Docker daemon only the client certificate subject common name. If the user has permission to access the Docker daemon, then the authorisation plugin provides rules to access the containers in the system. The authorisation plugin only obtains the user name that is the client certificate subject common name and the authentication method. Subsequently, the authorisation plugin allows or deny users access to the containers.

The usage of both Kubernetes and Docker Swarm has various advantages and disadvantages [63]. The benefits of using Kubernetes include fast deployment and infrastructure, which is fixed and clear. The infrastructure design enables Kubernetes to keep an updated record of the system's state, which in turn helps users to avoid errors. Other advantages of using Kubernetes include the various scaling techniques it enables, a high level of availability due to maintenance of the health of containers and nodes, and the availability of storage. Docker Swarm also has various advantages, including faster running speeds, comprehensive documentation, and its configuration is rapid and simple. Additionally, Docker Swarm allows the isolation of applications, which enhances security and improves its operation. It also enables users to control the version by allowing them to pursue successive container versions, explore the differences, or restore the prior versions. Using Kubernetes and Docker Swarm also presents various disadvantages. One disadvantage when using Docker Swarm is its dependence on a single platform. This means that despite supporting Mac and Windows operating systems, running it on non-Linux systems still requires virtual machines. Other disadvantages include the lack of storage

and proper monitoring of the containers. On the other hand, Kubernetes also has some disadvantages. They include difficulty installing Kubernetes on clusters in the absence of cloud providers such as Amazon or Azure, and a delay in the initialization process. Furthermore, significant effort is required for the migration of applications to stateless.

2.6.4 Azure Container Service

A cluster of virtual machines that run containerised applications could be simply created, configured, and managed by Azure Container Service [64]. Both Kubernetes and Docker Swarm are supported by Azure container services. Azure Container Service provides a number of services that allows the developer to use container-based applications on Microsoft Azure.

Azure Container Registry is one service that is provided by Azure Container services. Private Docker registry is provided by the Azure Container Registry to store container images. This service needs to be set up first to make the developer or user choose deployment of the containers. Azure Container Instance is another service provided by Azure Container services. The creation and management of Docker containers can be performed by this service in Azure without the need for virtual machines or use of a higher-level (and more complex) orchestration service. The Azure Portal allows the developer or user to set up this service. Azure Web App for Containers allows the user to perform a specific task without needing to set up an orchestration service and allows users to set up a web app, as well as managing the CPU or RAM which is a feature not provided by Azure Container Instances. The container allows the developer to run multiple applications on the same hardware. The Azure allows the developer to build the Docker images and run them. Figure 26 below shows that Kubernetes and Docker Swarm are supported by Azure container services.

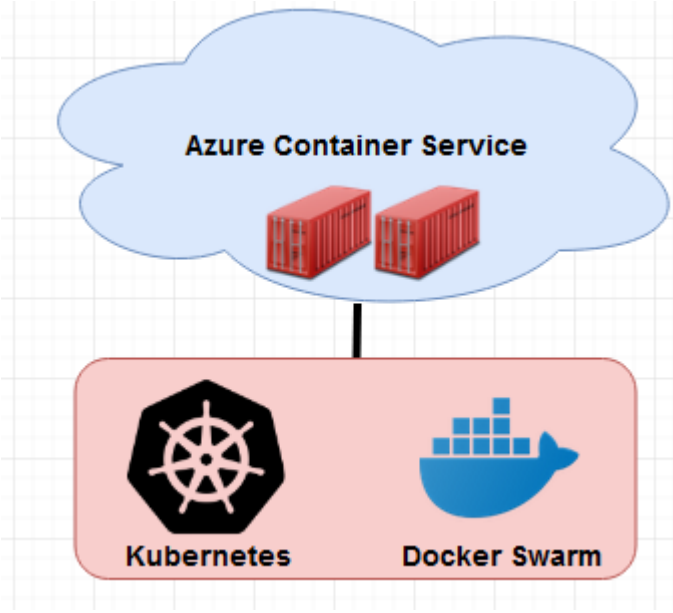

Figure 26 Azure container service

Image security in Azure container, the repository is the store of the image that is created in the containers and can be related to public or private container registries [65]. For example, Docker Hub is the public registry and the Docker Trusted Registry is the private registry in the Docker. The private Registry could be installed in

the private cloud. Azure Container Registry is one example of the cloud-based private container registry services. Container images consist of multiple layers and each layer has vulnerabilities. Therefore, the public registry could provide the container image but not guarantee security. In private cloud computing, security is an important issue to protect user data. Therefore, the private registry in Azure that is Azure Container Registry supports an authentication system that enhances security in the container images. Moreover, Azure container service is based on sharing the kernel. The containers running with root privilege run a higher risk of attacks because full root access to the host could be obtained by a hacker.

2.6.5 Google Kubernetes engine (GKE)

Managed environments for deploying, managing, and scaling the containerised application are provided by Google Kubernetes Engine that uses the Google infrastructure [66]. The cluster in Google Kubernetes Engine consists of multiple machines (specifically, Google Compute Engine instances).

The Kubernetes open source cluster management system runs the GKE clusters. The interaction mechanisms that are provided by Kubernetes allow the user to interact with the specific cluster. The Kubernetes commands and resources have several features such as executing administrator tasks; provide policies, application management and workload monitoring. Moreover, there are benefits that are provided by the GKE cluster. Loading balancing is provided by the Google Cloud Platform's for Compute Engine instances. Additional flexibility is another good feature of the GKE cluster because subsets of nodes within a cluster are designated by Node pools. Automatic scaling and upgrades for the cluster are provided by GKE. Maintenance of node health and availability is provided by Node auto-repair in GKE.

A similar structure rules exist with Kubernetes and Google services. Kubernetes depends on Google services rules to provide the same benefits, such as container monitoring, update rolling, automatic management and more. Google Kubernetes engine is supported by Kubernetes [60] so provides the same security features provided by the Kubernetes. Kubernetes provides an authentication system that allows or deny the user access to the cluster. Moreover, the API server in Kubernetes could be used to verify the authorised user’s request. Multiple modes are used by the Kubernetes API server to authorise the request

2.6.6 Amazon EC2

Compute capacity is secure and resizable in cloud computing by using Amazon Elastic Compute Cloud (Amazon EC2) that is a web service [67]. With this, developers can obtain and configure capacity with minimal friction by using a simple web service interface. Control the computing resources is easy and obtaining and booting new server time is reduced, which also provides quick scale capacity. Finally, the developer pays only for the capacity used, resulting in improved economy of scale.

Building failure resilient applications and isolating them from common failure scenarios could be performed through tools provided by Amazon EC2. Firstly, Bare Metal instances, the processor and memory of the underlying server can be accessed directly by Bare Metal instances. In addition, applications that need to run in non-virtualised environments could use this feature that could be a perfect method. Secondly, Amazon EC2 Fleet provides Compute Performance improvement and optimizes the cost. Provision of computer capacity across EC2 instance types Availability Zones, and purchase models are provided by Amazon EC2 Fleet through a single API call to help optimize scale, performance and cost. With the use of GPU Graphics Instances, this feature allow the

customers to obtain high graphics capability. Finally, High I/O Instances could be ideal for the customers that require very high, low latency, random I/O access to their data.

2.6.7 Cloud Foundry Diego

The deployment of application containers is also provided by Cloud Foundry with Diego cells that run the application containers [68] so each number of application containers could be run by the Diego cell. Therefore, the use of Cloud Foundry app that is related to Diego container could be an option.

Cloud Foundry Diego and Kubernetes are quite similar. However, creating container images is an essential step to allow the developer or user to create a container in Kubernetes. In Kubernetes, the run and build container need to install and initialize commands while in the Cloud Foundry Diego, the install and initialise commands are not required to run and build container. Diego provides the same Kubernetes security feature such as TLS certificate=fate that could be used to allow or deny an authorised users to access the system. Moreover, the Diego system provides an authorisation system that authorises access to objects based on Cloud Controller roles.

2.6.8 Container orchestration platforms analysis

Deploy and manage container could be provided by Azure Container Service, Google Kubernetes Engine, and Docker. Therefore, the configuration of popular open-source tools should be optimised by the engines. These services help the developers to hosts numbers, size, select tools of orchestrator, and container services.

On the other hand, an open-source orchestration system for Docker containers could be the Kubernetes, Cloud Foundry Diego, or Docker Swarm. The computer cluster has nodes that are scheduled by the orchestration, and managing the workload is the other main feature of orchestration. Kubernetes and Docker Swarm isolate containers individually or within their pods or clusters for Kubernetes and Docker Swarm, respectively, to maintain security and trust. However, both platforms do not allow the isolation of users within a container. This is because the containers' design only allows it to run one process for every container unless child processes are resulting from the process. Isolating users within a container results in the running of multiple processes that are unrelated within one container, which creates problems in the management of logs and in keeping all the processes running. These problems include the user needing a method to restart individual processes in the event of a crash, and having difficulties determining which processes logged to which output, since they all log to a similar output. The problems resulting from the attempted isolation of users within a container indicate that for Kubernetes and Docker Swarm, the container design only allows for the running of one process for every container. Hence, the users within a container cannot be isolated, as this would generate multiple processes. The inability to isolate users within a container led to the creation of pods for Kubernetes and clusters for Docker Swarm to enable the joining of containers and their management as one unit. Containers in pods or clusters share an almost similar environment, enabling the sharing of resources such as namespaces, network interfaces, and hostnames. The clustering of containers, therefore, creates an illusion of the processes running concurrently while taking advantage of the features that containers make available. In conclusion, Kubernetes and Docker Swarm are both vital platforms for the automation of the usage, management, and scaling of containerized applications. They are both similar in use but differ in their basic operations and the advantages and disadvantages they present.

Containers as a Service includes both the orchestrations and engine services. Creating a container could be performed by the engine services that could support orchestrate applications such as Apache Mesos, Docker Swarm, or Kubernetes. All cloud types could use one of the orchestration applications to manage the cluster.

In conclusion, orchestration applications provide a system for containers that are configured and optimised using container engine services. The orchestrate applications provide a solution for managing the containers, in that they are simple, powerful, open sources and make the cluster strong. A container engine can be easily setup, support orchestrate applications while providing a command line interface (CLI) tool.

2.7 Summary

In cloud computing, providers should protect the cloud environment to provide user data privacy and integrity. In hypervisor virtualisation, each user has his own computer that has an OS, applications and software. Cloud computing has three models IaaS, PaaS, and SaaS. IaaS is the physical hardware such as servers, processing power, storage, and networking. Therefore, a cloud provider could user security techniques to protect IaaS such as firewall, IDs and IPs. However, PaaS is an operating environment such as container-based virtualisation based on sharing one operation system that is the kernel OS. The isolation in containers is weaker because of sharing of this kernel OS. Finally, SaaS is a software that allows users to communicate with the system resources. Data encryption is one technique that is used to protect SaaS.

Another type of virtualisation is container-based virtualisation. Container virtualisation is a perfect way to separate user's application. Container virtualisation is based on Kernel OS that can virtually package and isolate applications for deployment without the need for traditional VMs. Container virtualisation has security features such as namespace, Cgroup and Docker daemon. Namespace isolates the user's containers from each other. Docker daemon allows the user to communicate with the containers. Cgroup manages the Docker resources such as CPU and memory.

Container virtualisation has weak isolation compared with traditional VMs which needs to be improved to make containers more secure in cloud computing. Moreover, containers do not isolate users within them. Existing orchestration applications and engine services could provide RBAC to manage the cluster but this does not provide any benefit if the underlying system is not properly secured. The next chapter is explored this point further as the basis for our novel design.

3. ANALYSIS OF ISOLATION IN VIRTUALISATION SYSTEMS

3.1 Introduction

This chapter provides information about the main hypervisor types and platforms and examines existing work that tries to isolate the user in cloud computing instead of VMS. The chapter focusses specifically on container-based virtualisation and platforms and examines the level of isolation offered in virtualisation to explain the different approaches used. The chapter then explains the basic concepts and describes the negative aspects of this approach related to the weaker isolation. This chapter will present some related work in VM and container-based virtualisation isolation to quantify the different levels of isolation of virtualisation systems and provides our work on isolation benchmarking in containers by using HTTperf to evaluate the isolation performance of Docker.

3.2 Isolation in Hypervisor-based Virtualisation

Isolating a user's OS, applications and libraries from other users is provided by virtualisation, which has different types such as Hypervisor type 1 [69]. In this case, the hypervisor runs directly on top of the host machine's physical hardware. Each guest shares the system's physical compute resources, such as processor cycles, memory space, and network bandwidth and so on has its own OS in the VM. Figure 27 below shows a type 1 Hypervisor.

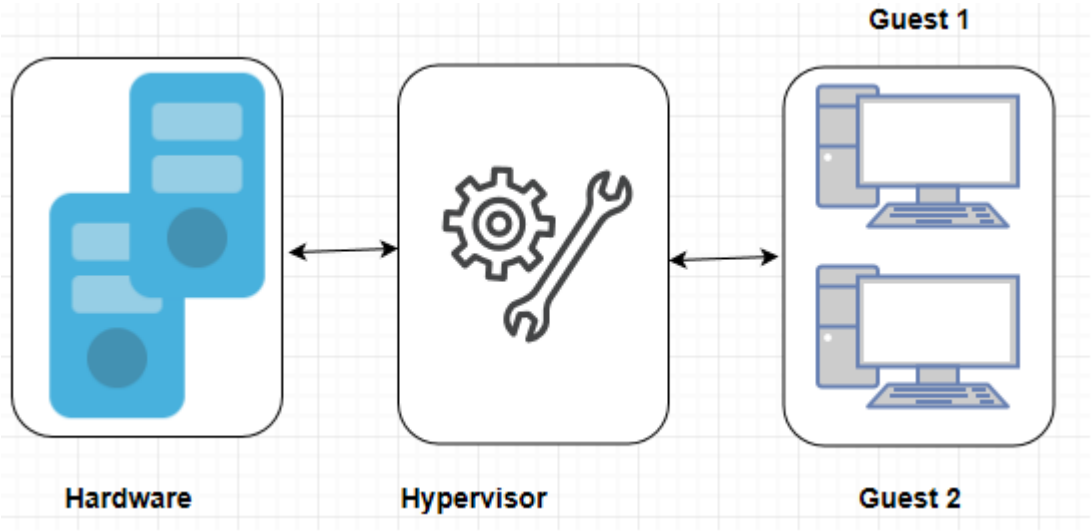

Figure 27. Hypervisor type 1

Hypervisor fully emulate another computer in the guest which allows them to emulate other types of device (for example a smartphone), other CPU architectures, or other OSs and this is also provided by cloud computing. Therefore, the developer does not need to physically have access to a target device because the hypervisor allows testing and developing applications by the developer on the development system, which is useful in some instances. Modern CPU capabilities have advantages and hypervisor has an advantage, which is that the CPU in an unprivileged mode could be directly accessed by virtual machine and its applications and resulting in performance improvements without sacrificing the security of the host system. Hypervisor are based on virtualising the hardware which allows each guest to deploy its own OS, libraries and application. Figure 28 below shows the scheme of hypervisor-based virtualisation where the hardware available to guests is usually emulated.

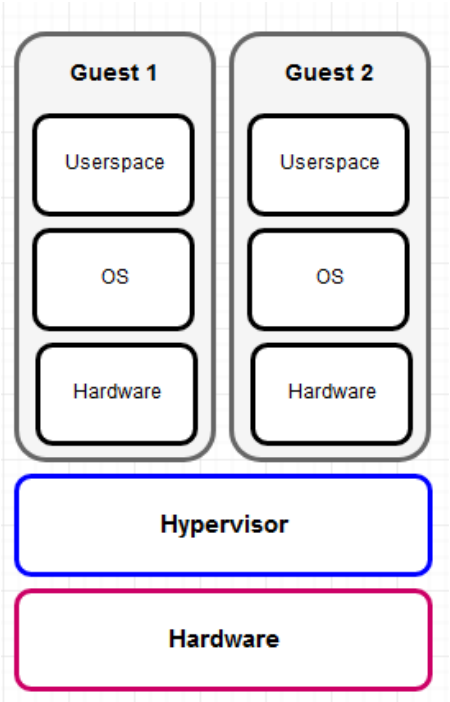

Figure 28. The scheme of hypervisor-based virtualisation. Hardware available to guests is usually emulated.

According to [70], the hardware resources/software resources of the host machine can be accessed by multiple applications or operations through the hypervisor Virtualisation that is layer between the hardware and the OS. For example, Virtual Machine Monitor (VMM), which is also known as the hypervisor. Virtual Machine Monitor (VMM) creates a path, which allows multiples of the same OS to run on the host machine, and the resources among the various OS hardware requirement are managed by the hypervisor. The Figure 29 below shows a host machine before virtualisation, which has a single OS and software and hardware resources per machine. It is an inflexible and expensive infrastructure. Moreover, conflicts of applications could happen because of running more than one instance of an application on the same machine.

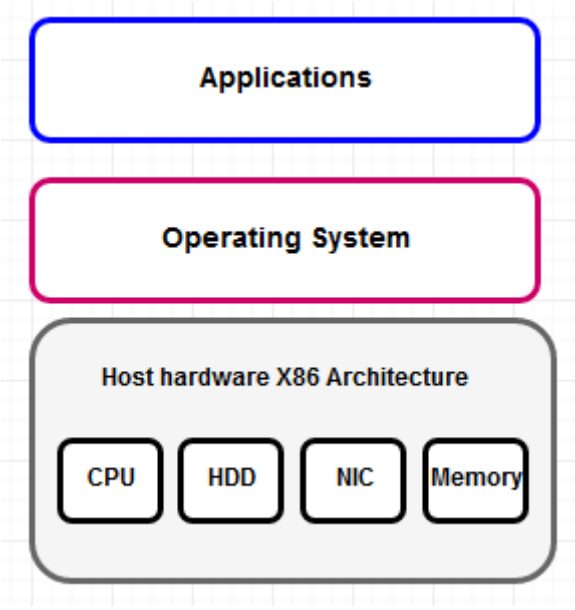

Figure 29. A machine before virtualisation.

3.2.1 Virtualisation platforms

Nowadays, the virtualisation technologies are available in a wide array of potential choices. The users often cannot identify which platform is best suited for their needs [71]. Therefore, the Figure 30 below provides a detailed comparison chart between Xen 3.1, KVM from RHEL5, VirtualBox 3.2 and VMware ESX.

	Xen	KVM	VirtualBox	VMWare
Para-virtualization	Yes	No	No	No
Full virtualization	Yes	Yes	Yes	Yes
Host CPU	x86, x86-64, IA-64	x86, x86-64,IA64,PPC	x86, x86-64	x86, x86-64
Guest CPU	x86, x86-64, IA-64	x86, x86-64,IA64,PPC	x86, x86-64	x86, x86-64
Host OS	Linux, UNIX	Linux	Windows, Linux, UNIX	Proprietary UNIX
Guest OS	Linux, Windows, UNIX	Linux, Windows, UNIX	Linux, Windows, UNIX	Linux, Windows, UNIX
VT-x / AMD-v	Opt	Req	Opt	Opt
Cores supported	128	16	32	8
Memory supported	4TB	4TB	16GB	64GB
3D Acceleration	Xen-GL	VMGL	Open-GL	Open-GL, DirectX
Live Migration	Yes	Yes	Yes	Yes
License	GPL	GPL	GPL/proprietary	Proprietary

Figure 30. A comparison chart between Xen, KVM, VirtualBox, and VMWare ESX [69].

Xen: The virtualisation method of each VM is investigated as the first point in this table. The Xen provides full virtualisation while the para-virtualisation is only provided by Xen. In the host and guest CPU, x86 and x86-64 are supported by Xen. Moreover, Itanium-64 architectures are supported by the Xen. The host environment for each system is investigated as the second point in this table. Xen supports Linux OS as a guest and host OS. Unix OS is supported by Xen as a guest and host OS. Guest OS support windows OS for Xen but Xen host OS does not support Windows. However, all variants of Linux, Windows, and UNIX can be supported by VT-X or AMD-V instructions. The HPC environment for each system is investigated as the third point in this table. Up to 128 vCPUs can be supported by Xen, which can address 4TB of main memory in 64-bit modes. Furthermore, 3D acceleration and live migration across homogeneous nodes are supported by Xen. The licence for each system is investigated as the final point in this table. Free Licence is for Xen, under the GNU Public Licence (GPL) version 2. Figure 31 below shows the Xen Architecture.

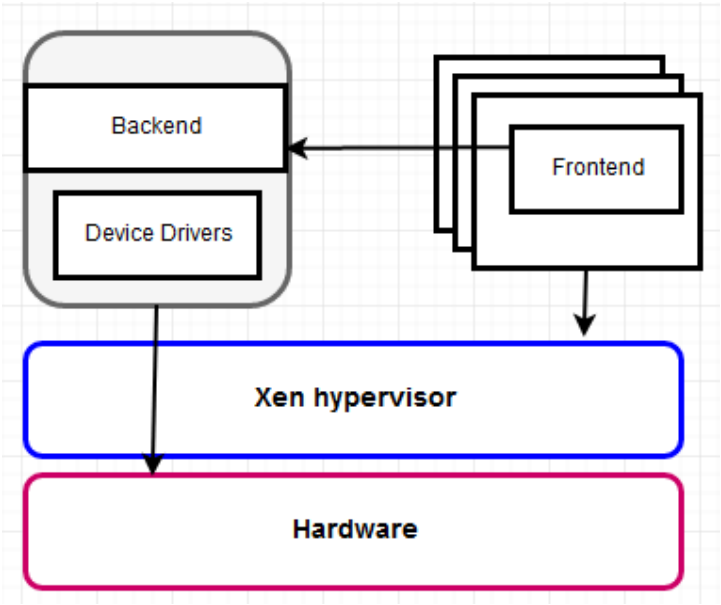

Figure 31. Xen Architecture.

KVM: The virtualisation method of each VM is investigated as the first point in this table [71]. KVM provides full virtualisation but does not provide the para-virtualisation. In the host and guest CPU, x86 and x86-64 are supported by KVM. Moreover, Itanium-64 architectures are supported by the KVM. Also, PowerPC is supported by KVM only. The host environment for each system is investigated as the second point in this table[71]. KVM supports Linux OS as a guest and host OS. Unix OS is supported by KVM as a guest but is not supported as a host OS. Guest OS supports Windows OS for KVM but host OS does not support Windows. However, all variants of Linux, Windows, and UNIX can be supported by VT-X or AMD-V instructions. The HPC environment for each system is investigated as the third point in this table [71]. Only 16 vCPUs can be supported by KVM that can address 4TB of main memory in 64-bit modes. Furthermore, 3D acceleration and live migration across homogeneous nodes are supported by KVM. The licence for each system is investigated as the final point in this table [71]. Free Licence is for KVM, under the GNU Public Licence (GPL) version 2. Figure 32 below shows the KVM Architecture.

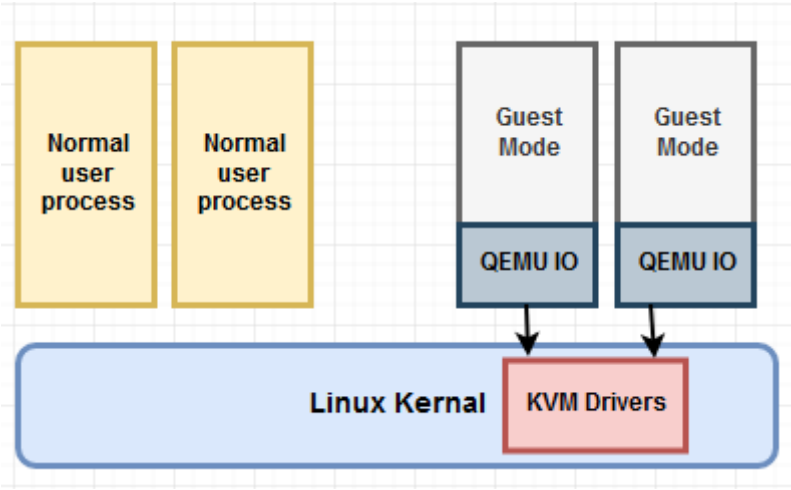

Figure 32. KVM Architecture.

VirtualBox: The virtualisation method of each VM is investigated as the first point in this table [71] .the VirtualBox provides full virtualisation but does not provide the para-virtualisation. In the host and guest CPU, x86 and x86-64 are supported by VirtualBox. The host environment for each system is investigated as the second point in this table. VirtualBox supports Linux OS as a guest and host OS. Unix OS is supported by VirtualBox as a guest and host. Guest and host OS support Windows OS for VirtualBox. However, all variants of Linux, Windows, and UNIX can be supported by VT-X or AMD-V instructions. The HPC environment for each system is investigated as the third point in this table. Only 32 vCPUs can be supported by VirtualBox and 16GB of addressable RAM per guest OS which could not be applicable for the large systems. Furthermore, 3D acceleration and live migration across homogeneous nodes are supported by VirtualBox. The licence for each system is investigated as the final point in this table. Free Licence is for VirtualBox, under the GNU Public Licence (GPL) version 2. However, additional features under a more proprietary licence dictated by Oracle since its acquirement from Sun last year is for VirtualBox is under GPL.

ESX: The virtualisation method of each VM is investigated as the first point in this table. The VMWare provides full virtualisation but does not provide the para-virtualisation. In the host and guest CPU, x86 and x86-64 are supported by VMWare. The host environment for each system is investigated as the second point in this table[71]. VMWare supports Linux OS as a guest but does not support as host OS. Unix OS is supported by VMWare as a guest and host OS. Guest OS support Windows OS for VMWare but host OS does not support Windows. However, all variants of Linux, Windows, and UNIX can be supported by VT-X or AMD-V instructions. The HPC environment for each system is investigated as the third point in this table. Less than any other, VMware supports 8 vCPUs but 64GB of addressable RAM per guest OS, more than VirtutalBox. Furthermore, 3D acceleration and live migration across homogeneous nodes are supported by KVM. The licence for each system is investigated as the final point in this table. Licence for VMWare is completely proprietary.

The figure 33 below shows that on type of the hypervisor, which is VMware. A host machine after virtualisation, has more than one OS and software and hardware resources [70]. Each guest has its own OS, libraries and application, therefore each guest does not rely on the OS or hardware of the host.

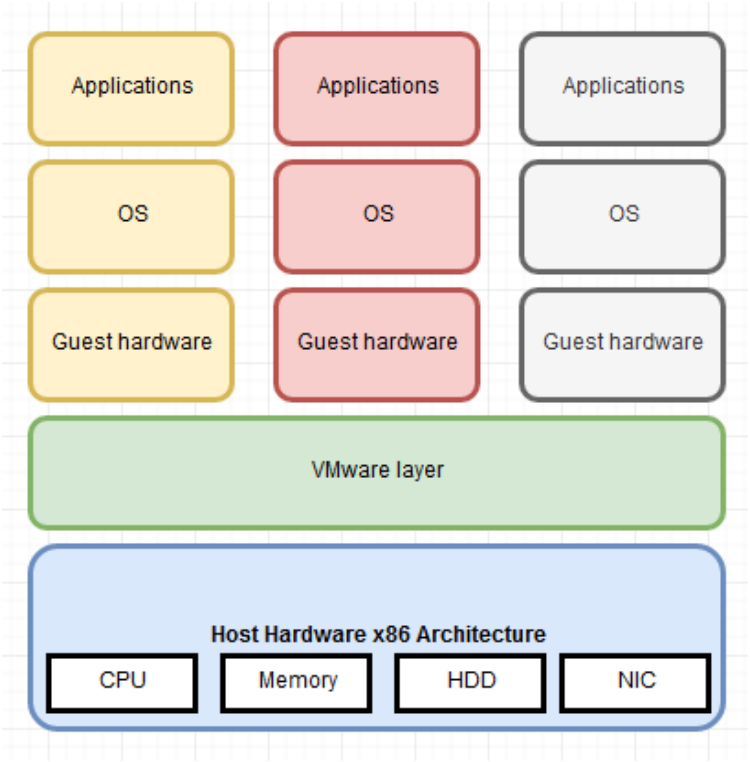

Figure 33. A machine after virtualisation.

VMware workstation is the first product of VMware, which has become successful for desktop virtualisation. VMware GSX Server was released as the first version of server virtualisation platform in late 2000 [72]. The first version of Elastic Sky X (ESX), which was released in later 2001, has a different approach to that of VMware workstation. The responsibility for regulating CPU affinity, memory allocation and oversubscription, network bandwidth throttling and I/O bandwidth control is the core of VMware ESX [72]. VMware ESX runs on the physical Host Server [72]. Hardware Capability List (HCL) has the main servers' brand on the market in the VMware. Using ESX on the servers reported on HCL could be convenient. VMkernel is another important module of VMware. The ESX host server can run VMkernel which is a high performance OS developed by VMware. In the VMkernel, a new device could be added without the need of recompiling it. Console Operating System (COS) is the third important module, which provides an executing environment for monitoring and administrating ESX. Figure 34 below shows the VMware ESX.

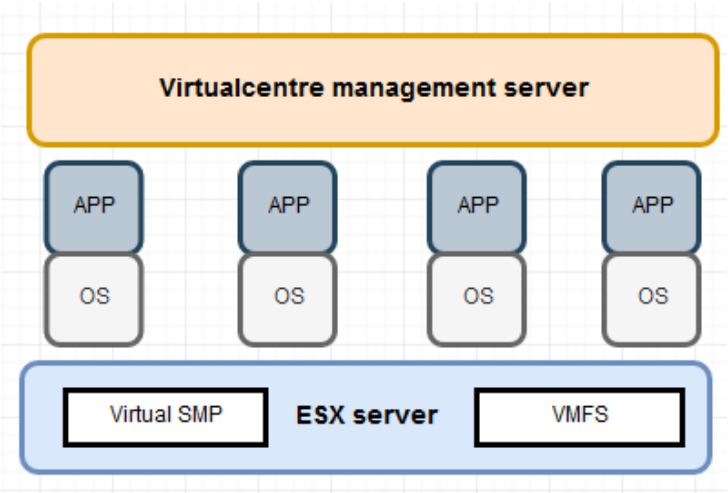

Figure 34. VMware ESX.

Virtual Machine File System (VMFS) and VirtualCenter are two other important components of VMware ESX [72]. Virtual Machine File System (VMFS) is a high performance cluster file system created by VMware. VirtualCenter is the management console used to control the virtualised enterprise environment.

In the hypervisor, the hardware and a virtual machine can communicate with each other through the hypervisor so that the virtualisation accomplishes with this abstraction layer (hypervisor) [73]. Type 1 and Type 2 are two types of the hypervisor, which are based on the different levels of implementation. Type 1 is sitting on the hardware that enables the hardware and virtual machine to communicate directly with each other. Type1 hypervisor runs directly on a physical machine, therefore, the host OS is not required in this type. It is sometimes called a 'bare metal hypervisor because it does not need the host OS and can run directly on a physical machine. This category identifies VMware vSphere/ESXi, Microsoft Windows Server 2012 Hyper-V, Citrix XenServer, Red Hat Enterprise Virtualisation (RHEV) and open-source Kernel-based Virtual Machine (KVM). In type 2, the virtual machine can be easily managed by the OS with the support of hardware configuration from OS. Type 1 is more efficient than type2 hypervisor because type 2 hypervisor has an extra layer between the hardware and virtual machine. This category has VirtualBox and VMware Workstation. Different roles are described by Host or Guest machine (or domain) in the hypervisor. The virtual machines are managed by the host machine and isolated and secure environment for each guest machine, which sits on a hosted machine. With these separated roles, resource boundaries to multiple virtual machines on the same physical machine could be offered by the hypervisor.

3.3 Related Work on Resource Isolation in Hypervisor-based Virtualisation

Recently, an isolation management process in cloud IaaS was proposed by the trusted virtual datacentre (TVDc) that addresses these multi-tenancy and management issues [28]. In trusted virtual datacentre (TVDc), trusted virtual domains (TVDs) group the TVDc virtual machines and their associated resources such as virtual bridge and virtual local access network (VLAN). An isolation policy is enforced towards its group members by each TVD. For instance, each hypervisor has colour and VMs can communicate with each other if they have the same colour and each VM can only run on a hypervisor that has the same colour as that VM. Isolating customer workloads from each other is the main goal of this process to reduce the threat of co-locating workloads from different customers.

Virtualisation technology is now widely used in modern data centres on cloud platforms to isolate the users in the cloud computing. Moreover, the authors suggest another approach to isolate the users in cloud computing instead of the virtualisation technology. Micro-architecture is used to overcome the limitation of using virtualisation technology as the sole approach to isolate workloads and users within a cloud. Furthermore, there is a set of heuristics for VM placement to improve the performance in the cloud platforms which considers security requirements.

In this paper, the author tries to find another approach to isolate workloads and users within a Cloud in order to limit the use of virtualisation technology as the sole approach [29]. In highly secured environments, unshared resource environments for two tasks with different security clearances ensure that strong isolation. However, the current public and private Clouds use virtualisation to improve the isolation in the cloud computing. Therefore, the strong isolation in such environments becomes critical because of the widespread usage of virtualisation.

3.3.1 Micro Architecture: where virtualisation failed

Creating isolation in the cloud computing is the current trend of security in Clouds which use virtualisation as the sole mechanism to improve the security for users and their VMs [74]. However, micro-architectural components do not have a proper isolation, which leads to the ability to create covert channels that can be used to extract sensitive information such as cryptographic keys between these VMs. However, cloud computing also has another issue, which is performance isolation that is also critical for Clouds. In order to slow a collocated VM down, a VM can exert such noisy-neighbour behaviour, which leads the collocated VM to release resources. Therefore, extracting more performance from the Cloud can be provided by the offending VM through releasing resources.

Covert Channel, The micro-architecture components can be used as covert channels to leak information between concurrent VMs because they are not properly virtualised in modern platforms. The authors suggest that there are three channels that are memory, disk and L2 cache to create covert channels of communication between virtual machines. In the cloud that uses the memory bus [75], the authors present a new inter-VM attack as a new channel, which allows them to not be limited to a set of cores sharing a cache, instead being able to reach the entire physical machine.

Noisy Neighbours: Performance, Interference. In cloud computing, isolating multiple tenants' performance could be provided by virtualisation [76]. Moreover, no performance interference between two VMs can exist with perfect performance isolation. However, only CPU time and memory capacity are focused by these algorithms without considering other resources.

The authors show that the behaviour of a collocated VM could result in the loss of up to 80 % of the performance [77]. Resource-Freeing Attacks (RFA) is a new class of attacks that could appear from interference issues. These observations to improve a VM's performance by forcing a competing VM to saturate some bottleneck resources are provided by a resource-freeing attack to create interference that would lead a VM to starve, due to its inability to access specific micro-architectural components and free other resources. The performance is improved up to 60% in the lab and 13% on Amazon EC2.

Amazon provides a service to continuously monitor memory access latencies and generate alarms when anomalies are detected [78]. However, there is a high rate of false positives in this approach and high price for this service. In [79], they suggest an approach that helps to improve the hypervisor to counter attacks on CPU

caches. However, this approach increases the memory overhead from 5.9% to 7.2% and loss of 3% of memory and cache.

3.3.2 Resource allocation with security requirements

Using constraints that reflect the goals expressed by the user and the entity that operates the infrastructure is the main approach to help guide the placement of virtual machines [74]. The placement problem could be resolve as Constraint Satisfaction Problems. However, solving large problems is time consuming while different goals could be represented by constraints such as energy saving. The search space could be reduced by the hierarchical representation to improve the performance of the VM placement algorithm. Physical Machines (PMs) and clusters could avoid cache and processor resource contention through scheduling algorithms to increase performance isolation between processes.

Efficient and scalable algorithms is the current issue that is able to work at the scale of multidata-centres Cloud infrastructure [29]. CSP or linear program approaches do not scale well. Indeed, a multi-dimensional bin packing can be formalised placing VMs on Clouds. Therefore, the problem could be complex to solve. Reducing complexity could be done by Heuristics that allow almost optimal solutions. In this paper, the authors use heuristics. Multi-data-centres Clouds or any infrastructures described through the distributed system model could have the isolation requirement for placing VMs by the heuristics approach. Increasing performance isolation between processes is the purpose of these approaches [29].

User Isolation Requirement. Enforcement of isolation between users is provided by the user isolation requirement [29]. The requirement has a scope where it applies. For example, PM is the requirement's scope, all the levels below the PM will be applied by requirement. The higher the quality of the isolation is the higher level. Alone, friends and enemies are three types of isolation requirement. Only user VMs can share the resources in the alone requirement while a list of users allowed to share resources with him in friends. Finally, a list of users that must never share resources with him in enemies. Two functions are to verify these requirements. Checks if a node and the VMs already allocated on it respects the user's requirements is the first function. Moreover, if all the nodes below (from a level perspective), the checked node also verifies the requirements that are verified by the first function. A new VM respecting the user's requirements of all VMs already allocated on a given PM is the second function.

Heuristics. First fit and best-fit heuristics that take into account the hierarchy are presented in this section [29]. In the both heuristics, the list of all PMs on the distributed system are first taken. Then, the first fit approach finds a node that fits the capacity and isolation requirements and iterates through it, while the Best Fit heuristic returns a list that contains all fitting PMs which can be sorted using one of the sorting algorithms. Figure 35 shows the different heuristics.

Short Name	Name	Hierarchy Aware	Complexity	Complexity EC2
FF	FirstFit	✗	N_{PM}	40320
FFS	FirstFitShuffle	✗	N_{PM}	40320
BF	BestFit	✗	N_{PM}	40320
$HABF$	HierarchicalAwareBestFit	✓	N	40873
$HABFP$	HierarchicalAwareBestFitwProperties	✓	N	40873
$HABFL$	HierarchicalAwareBestFitLarge	✓	N	40873
$HABFLS$	HierarchicalAwareBestFitLargeSort	✓	N	40873

Figure 35. Different heuristics and their complexity [30].

3.4 Container-based Virtualisation

Nowadays, cloud computing and distributed systems are increasingly being supported by Linux container virtualisation. Containers are OS virtualisation in that container-based virtualisation is based on sharing the OS and resource while the hypervisor is based on virtualising the hardware [80]. Container-based virtualisation is therefore a lightweight alternative to the hypervisor. In the container virtualisation should also be easy to share things between containers and the host because of the common kernel OS.

In container virtualisation, isolating a set of processes and resources such as memory, CPU and disk is easy from the host and any other containers. The physical host resources are divided to create multiple isolated user-space instances. Each process or resources inside the container cannot be seen by other processes or resources outside the containers. Therefore, the containers provide an isolated application environment. Control Groups (cgroup) is responsible for the resources management in container-based virtualisation systems that restricts the resource usage per process group. For example, CPU can be limited or prioritised by using cgroups as well as memory and I/O usage for different containers.

Container is one possible way to separate user's resources and environment [18]. Therefore, each resources within the container is completely isolated from other resources outside the container. Container virtualisation is based on one OS that is the kernel OS. Hypervisor provide better isolation compare with the container because each guest has its own OS. Furthermore, the container-based virtualisation is a good choice to isolate the application environment and resources. OpenVZ, VServer and LXC are container platforms examples.

3.4.1 Linux Container features

Linux containers (LXC) is a project which helps to isolate processes from other processes [69]. A secure environment (containers) is built by the LXC to be escape-proof for a process and its child processes and to protect the system even if an attacker manages to escape the containers; secure access to the kernel interfaces is provided by the container that also has fully virtualised network interfaces. Creating containers on Linux is done by the Linux containers (LXC) but it is possible to isolate processes or groups on the Linux by using other tools. However, the popular tools to achieve the isolated environment is LXC. Moreover, the LXC has some features such as Linux kernel namespaces, Control groups and Mandatory Access Control that help to create an isolated environment.

Linux kernel namespaces:

Isolated process groups are created by the Kernel namespaces feature [69]. In addition, subsystems such as the communication or the network subsystem of the kernel could be completed by the Kernel namespaces. Creating a different namespace is the Kernel mechanism for each process that need to be separated. The processes that have process ID (PID) that is already in use the host system or other containers could have namespace by using the kernel namespace features. In addition, this makes sharing common things between host and containers easy. Isolate security-related identifiers and attributes are created by user namespace. To summarise kernel name space features, build containers and isolate the process IDs from other namespace process IDs. However, the same process IDs could provide processes in different namespaces.

This allows each container to have its network artefacts like routing table, iptables, loopback Interface that are provided by net namespace. Moreover, isolation for various Inter-Process Communication (IPC) mechanisms namely semaphore, message queues and shared memory segments is provided by IPC namespace. Each container has its own mountpoint that is provided by the mnt namespace. Finally, viewing and changing hostnames for different containers could be provided by The UTS namespace. On the other hand, the containers can interact with each other through network interface through the host.

Control groups:

Control groups is called cgroups in order to isolate processes from other processes [69]. They track processes and process groups including forked processes is a mechanism which is provided by cgroups. Cgroups implement fine-grained resource control as well as provide hooks that allow other subsystems to extend those functionalities. However, the process isolation is not a responsibility for the cgroup or making the isolation stronger, they merely ensure fair allocation of CPU time, system memory, network bandwidth, or combinations of these resources. Therefore, certain resources allocated to the containers can be monitored, configured or accessed by using cgroup. Critically, the cgroups do not allow to a single container to bring the system down by exhausting one of the resources. Figure 36 below shows the kernel namespace and cgroup.

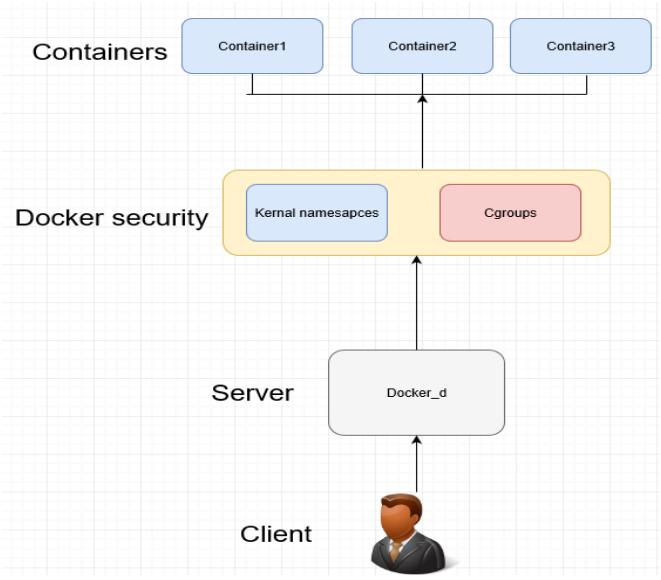

Figure 36. Kernel namespace and Cgroup.

3.4.2 Container Virtualisation Platforms

Numerous container-based virtualisation solutions exist. This section provides an overview of the existing implementations and how they work.

Linux-VServer

One of the oldest implementation of Linux container-based system is Linux-VServer [80]. Linux-VServer introduced (through a patch) its own capabilities in the Linux kernel to guarantee isolation instead of using namespaces such as process isolation, network isolation and CPU isolation. Traditional chroot system is used in the Linux-VServer to jail the file system inside the containers. Therefore, the scope of the file system for the

processes is limited by the Linux-VServer. A global PID space is responsible to carry out the processes isolation. All processes outside of a container's scope are hidden by global PID space and unwanted communications between processes of different containers are prohibited by the global PID space. The large number of containers is the main benefit of this approach. However, usual virtualisation techniques cannot be implemented by this system such as live migration, checkpoint and resume, due to the impossibility to re-instantiate processes with the same PID.

In Linux-VServer, Containers identify a tag in packets to ensure only the right container can receive them. However, this approach has drawbacks, binding sockets to a subset of host IPs and to change their own route table and IP tables rules is performed by the administrator not by the containers.

The Token Bucket Filter (TBF) is used by Linux-VServer to perform CPU isolation. Each container and server is connected with a token bucket to accumulate tokens at a certain rate. Therefore, the creation of a token is connected to every process running in a container. Xen Credit Scheduler is very similar to the token bucket scheme.

OpenVZ

OpenVZ provides similar functionality to Linux-VServer [80]. However, an isolated subset of a resource is for every container in the OpenVZ because it is built on top of kernel namespaces. Isolating the process between different containers is done by using PID namespace. Unique process IDs is for every container process. Furthermore, usual virtualisation techniques, such as live migration, checkpoint and resume can be used through the PID namespace and this feature is not in the Linux-VServer. The IPC kernel namespace capability means each container has its own shared memory segments, semaphores, and messages. In addition, there is a network stack for each container in the OpenVZ due to the network namespace.

Setting up standard UNIX per-user and per-group disk limits for containers is provided by the Disk Quota. Finally, I/O scheduling is a similar approach of CPU scheduling. Fair Queuing (CFQ) Scheduler is used by the second level scheduling. I/O priority is for each container and I/O bandwidth available according to priorities is distributed by the scheduler. The vzctl tool is responsible for controlling the OpenVZ containers.

LXC

Resource isolation among all containers is provided by kernel namespaces in the same way as OpenVZ, LXC [80]. The PID namespace, IPC namespace and file system namespace, respectively virtual PIDs, IPCs and mount points. Network namespaces is to communicate with the outside world and to allow the network isolation in the LXC. Route-based and Bridge-based are two configurations in order to configure the network namespaces. Only cgroups allow the resource management and this feature is unlike Linux-VServer and OpenVZ. Therefore, the configuration of network namespaces is defined by cgroups in LXC, which also accomplishes the process control that has the function of limiting the CPU usage and isolating containers and processes. CFQ scheduler is responsible for controlling I/O operations in the same way as OpenVZ and the lxc-tool areresponsible for managing LXC containers.

3.5 Isolation in containers

Various merits come with container-based technology which is related to speed and efficiency in the execution of operations as the level of abstraction mainly occurs at the OS level that is better than virtualizing the whole hardware stack[7]. Containers therefore share the same OS kernel while offering root authorisation in some cases while using the Linux OSs. Despite the merits, there are also downsides in using container-based technology. Weak isolation in containers is due to the sharing of components and OS kernel. In the case of an attack or a flaw in the container the magnitude is amplified because the attack can be propagated to the underlying OS and consequently to other containers. Container-based virtualisation relies on customizing an OS and providing isolation, hence sharing, of its resources. Thus, an application coupled with all its dependencies is deployed or installed in a container. However, weak isolation allows installation of applications by third parties that may have standard security measures and controls. If not checked well they may contain malware or other malicious code that would spread to other containers thus compromising a whole system [81].

For example, the same network bridge is still shared by all containers that could for example, increase the possibility of ARP poisoning attacks between containers on the same host. Here, an attacker could send ARP (Address Resolution Protocol) fake message that is a kind of security attack [82]. The MAC address will be linked with the IP address of a legitimate system on the network through this type of attack. Therefore, each bit of data from that specific IP address could be obtained by the attacker. The attacker could obtain any information of the bridge if the attacker can compromise a container. In addition, the malicious payload could be injected by the attacker into the network.

It is therefore important to understand the ways that isolation differs between traditional and container-based virtualisation as any flaws in the underlying system might negate the trust and security features outlined in the previous chapter. A recent study by IBM involved the comparison of Linux containers and full virtual machines, with the primary goal of achieving efficient methods of resource controls using the two different methodologies. The study stated that cloud computing has paved the way for the use of virtual environments and container technology. There was a comparison of traditional virtual machines and Linux containers regarding handling workloads that are CPU, memory, and network intensive which found that negligible overheads are introduced by both KVM and Docker for CPU and memory performance. However, KVM introduces a significant overhead impact for I/O-intensive workloads. Therefore, better performance and easier deployment could be provided by the container. Cloud-based solutions like Amazon provide virtual machines to their customers to be able to run various services like databases but Container-based technologies now seem to offer a better alternative for cloud-based solutions [83].

A similar study conducted by [84] provided evidence that no system achieves perfect efficiency and isolation and some level of trade-off is always involved. The usage scenario is the determining factor when it comes to determining the isolation, provided that it is achieved by trading off the efficiency. Isolation is critical in environments that are sensitive such as data centres, and the proper isolation is required for the proper functionality of the systems. The container design starts with the hardware and an OS is added to provide the required level of isolation. The virtual machines are logically independent of all others in the system. However, containers depend on one underlying kernel OS in order to make the containers lightweight. The study proves that hypervisor technologies such as VMware and Xen offer the best isolation because of their ability to support multiple kernels

and loading arbitrary modules. A container-based OS has a single kernel that does not allow loading the arbitrary modules perfectly.

The hypervisor is used for most commercial of the cloud computing [85]. However, some companies are now using containers to build a cloud computing system such as Google, IBM/Softlayer, and Joyent. The container clouds could provide better performance than hypervisor-based clouds. “Fast data” NewSQL system benchmark used to compare the performance in container and hypervisor. The performance in IBM Softlayer that using container is better than Amazon AWS using a hypervisor. In cloud computing, using container could not be a good idea because the isolation issue in the container. The isolation security could be solved by making each VM has one Docker only.

A similar study was conducted by [86] that compared between VM and container isolation. Virtual machines (VM) is based on the virtualised resources that could be seen as isolated execution contexts [86]. OS should be installed in each VM that completely isolated from others. Container-based virtualisation is based on use the kernel OS. The weak isolation is one of disadvantages of container-based virtualisation. Isolate resource process is provided by the namespace to support container implementation. This isolation could be performed by kernel namespace.

A similar study related to using containers to build PaaS and provide a fine-grained control over resource sharing [87]. However, security in the container platforms such as LXC needs to be improved to make the PaaS secure. As well as, each container could has its own OS to be Independence Container that is completely independent from other container system. For example, each VM has an independent container system. Independent container could improve the security in the container because sharing kernel OS could be fatal in some instance. For example, bugs and viruses could carry over all containers because sharing of the kernel OS.

A similar study shows that container virtualisation technologies could help to build cloud computing [88]. Recently, cloud providers could use containers to build cloud computing that would help to reduce the costing of building cloud platforms. However, raising security concerns could be increased by the attack surface in the containers. The attacker can be poisoning attacks between containers on the same host because sharing of the kernel OS. As well as, the user can communicate with the container through the Docker daemon. Therefore, the Docker daemon should be protected by building a subsystem to build a secure cloud environment.

3.5.1 User isolation in container-based virtualisation

In Docker, isolate processes, networks and devices are provided by the kernel namespaces. Limiting resource consumption is provided by cgroups, which is a part of the Linux kernel to isolate the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes. The container therefore has features to allow developers to create isolated environments such as isolating the resources, or application environments and an independent runtime environment is provided by using Control Groups (cgroups) and namespaces. Therefore, a logically separate network stack and process space, as well as a file system, are received by each container. On the other hand, user isolation is not provided by Docker because the container does not offer user id isolation.

Overall, containers provide limited user isolation for many reasons such as sharing the same kernel. That means all running containers and the host may be compromised when malware infiltrates just one instance. Moreover, all containers should use the same version of the same kernel module. For these and other reasons, the developers must design a security system to improve the user isolation in container-based virtualisation because

it is not enough which could be fatal in a multi-tenant environment because malicious tenants could attack other containers through the shared kernel OS.

Secure virtualised data centre resources, user privacy, and preserving data integrity are the three main objectives to protect cloud providers. This project will aim to implement a security system that forms a 'trust layer' for container-based virtualisation. While sharing the host OS kernel is the main idea for containers, developers could make them provide much better user space isolation. For example, anything running inside a container can only see resources that have been assigned to that container which can configure and run different applications and libraries. This system enforces the concept of role-based access control and security policy. In role-based access control, computer or network resources can only be accessed by users based on the roles of individual users within an organisation. In this context, 'access' is the ability of an individual user to perform a specific task, such as view, create, or modify a file.

3.6 Evaluation of VM and Container-based Virtualisation Isolation

In order to quantify the different levels of isolation offered by VM and container virtualisation, first present some related work on the topic before then conducting our own investigation using Docker.

3.6.1 Related Work on Virtualisation Isolation Performance

A study was conducted to investigate the degree to which a misbehaving virtual machine affects the performance of other machines [7]. The study involved six kinds of stress tests including memory, CPU, disk, and networking on three different types of virtualisation environments including both VMs (VMWare and Xen) and containers (OpenVZ). In all cases, the performance of the stress tests on separate virtual instances were measured using an Apache server response rate in SPECWeb.

In the memory intensive and 'fork bomb' tests, whereby a looping process simply creates new child processes, all the virtual machines continued to work perfectly but the Xen virtual machine displayed marginally degraded performance. This is significant because the misbehaving machines were completely overwhelmed but isolation was maintained. The CPU intensive tests involved arithmetic operations and all the virtualisation environments performed well in this test despite CPU load on the misbehaving virtual server rising to 100%. This leads to the conclusion that the scheduling algorithms used by the CPU are sufficient to allow the web server enough CPU time [7]. The tests were not repeated with more processor-intensive tasks. The disk-intensive test used IOzone and involved running ten threads of IOzone in an alternating write and read pattern. In the VMware case there was good overall performance despite performance degradation of around 40%. However, in the case of Xen, load on the misbehaving virtual server was only around 12% but led to a 2% impact on the other instances while in OpenVZ the degradation was the same across all instance at around 2.5%. This is a clear indication of weaker isolation in containers.

The last test was the network intensive test, which focussed on an examination of the server receiving and sending information. Here, there was degradation in all the virtual machines. Some network controls were enforced by using tools such as **ipqosconf.** There was, however, no difference realised as there was a continuous crashing and rebooting. On the server transmitting data, there were four threads that each sent 60k of data. There were mixed results. In the VMware workstation, the well-behaved machines reported a 100% response but the misbehaving machine had a 53% degradation in performance. In the Xen scenario, the well-behaved virtual

machines reported no degradation while the misbehaving machine reported a degradation that was 0.5% or less. In the OpenVZ scenario, the misbehaving machine reports a 29% degradation while the other well-behaved machines report a 21.3% thus again providing evidence of weak isolation. In the receiving tests the process was reversed with the server receiving data at the same rate. Here, there was little impact in both the misbehaving virtual server or the normal ones but, again, in this case OpenVZ performed badly. Here, none of the servers returned a result indicating that they were completely overwhelmed. Figure 37 below shows the percentages of degradation and a summary of the results.

	VMware Workstation		Xen		OpenVZ	
	Good	*Bad*	*Good*	*Bad*	*Good*	*Bad*
Memory	0	91.30	0.03	DNR	0	DNR
Fork	0	DNR	0.04	DNR	0.01	87.80
CPU	0	0	0.01	0	0	0
Disk Intensive	0	39.80	1.67	11.53	2.52	2.63
Network Server Transmits	0	52.9	0	0.33	21.3	28.97
Network Server Receives	0	0	0.04	0.03	DNR	DNR

Figure 37. Shows the percentages of degradation and a summary of the results [7].

In summary, the results show the various types of behaviours when evaluating the isolation properties of the virtualisation technologies. VMware operates perfectly in isolating the VMs in all the stress tests and para virtualisation, like in the case of Xen, offering efficient isolation regarding resource degradation. Only OS virtualisation (OpenVZ) displayed significant performance impacts, particularly where no resource-sharing controls are applied. The results also show that the networking tests resulted in the biggest impact, and therefore provided the weakest isolation, between virtual instances. This could be a result of the network-oriented measurements using SPECWeb. There was also some impact in the disk intensive tests, especially given the limited load the test introduced in the normal servers.

Another shortcoming of the test this that it only considers a single type of containerisation whereas there are now many different implementations and approaches that could be used. The type used here (openVZ) is perhaps the closest to traditional virtual machines but does not accurately represent the full scope of this field. For example, Docker provides a much more lightweight environment for dedicated containers and expect their performance to be very different as a result.

3.6.2 Container Isolation Performance Using Docker

To evaluate the isolation performance of Docker, this project has replicated the test above to evaluate the impact on a HTTP server in one container while the other is being used to run the above-mentioned isolation benchmarking tests. Our aim is to see if the isolation provided by Docker is sufficient to prevent the benchmarking container from affecting the operation of the HTTP server.

In this test, experiment have created two raspberry pi model B container hosts, one as a client and the other as a server; both are running Raspbian OS and Docker. Raspberry Pi is a good choice for testing and developments

because it provides a realistic system environment for low cost and has small size. Raspberry pi provides less power in comparison to full server systems, but can be configured with a similar platform, including container-based virtualisation. Raspian OS is the official Raspberry Pi OS that natively supports containers and they are resource constrained, which should highlight any issues with weak isolation.

In this experiment have used Httperf for our testing because it is a more open and flexible approach. The client is running Httperf in a single container while the server is configured with two containers, one with an apache2 webserver and another with the isolation-benchmarking suite. They are connected via a local Ethernet connection running at 1GBps. This configuration is shown in Figure 38.

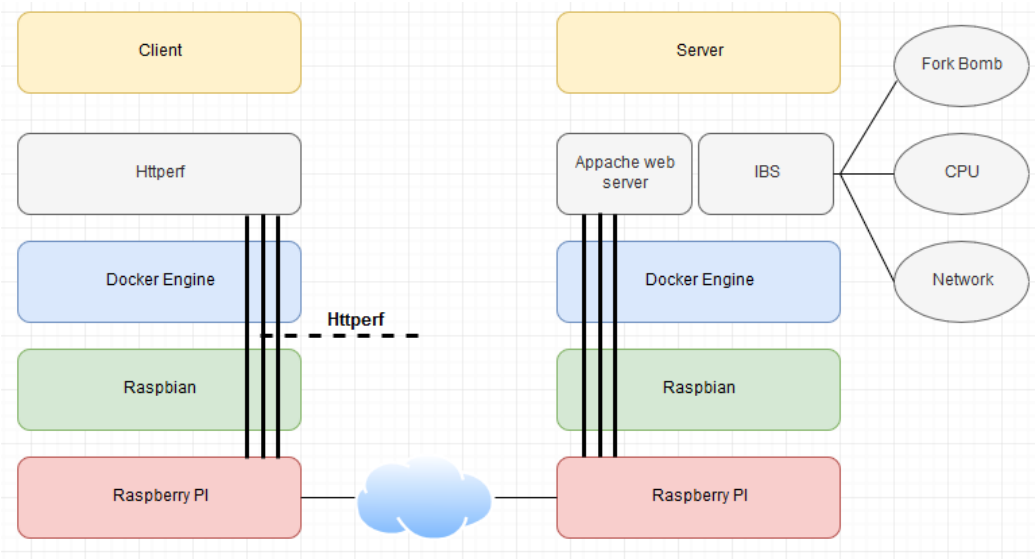

Figure 38. Test bed configuration

Httperf provides a very rich set of results, which can be used to understand the overheads to process incoming requests on a webserver and, therefore, should highlight any impact of the isolation testing. The Httperf tests were conducted with the following settings, the total number of connections was set at 27,000 and the request rate was set at 150 requests per second with a timeout of 5 seconds. First, a baseline was established by running Httperf on an unloaded system and the results are shown in the following Figure 39.

Connection rate: 150.0 conn/s (6.7 ms/conn, <=2 concurrent connections)
Connection time [ms]: min 2.6 avg 2.8 max 7.8 median 2.5 stddev 0.1
Connection time [ms]: connect 0.9
Connection length [replies/conn]:1.000
Request rate: 150.0 req/s (6.7 ms/req)
Request size [B]: 75.0
Reply rate [replies/s]: min 150.0 avg 150.0 max 150.0 stddev 0.0 (3 samples)
Reply time [ms]: response 1.9 transfer 0.0
Reply status: 1xx=0 2xx=0 3xx=0 4xx=2700 5xx=0
CPU time [s]: user 4.48 system 13.51 (user 24.9% system 75.1% total 100.0%)
Net I/O: 79.8 KB/s (0.7*10^6 bps)

Errors: total 0 client-timo 0 socket-timo 0 connrefused 0 connreset 0

Errors: fd-unavail 0 addrunavail 0 ftab-full 0 other 0

Figure 39. Base line Httperf results

According to the above result, Httperf running alone shows the number of TCP requests and replies is 2,700 for the test duration of 17.997s (150 connections per second). The lifetime of CPU connection is average 2.8ms with 0.9ms actual connection time. The reply rate was 150 per second and the time taken by the web server to respond to the request is 1.9ms. Total net I/O was 79.8 KB/s.

After that, the isolate benchmark suite tools were installed on the other Docker container in the server to test the performance by using Httperf while running the isolate benchmark suite. First, the CPU intensive tests were run on the server and Httperf test on the client. The results are the same as previously shown with no obvious impact. This was repeated with the network tests. Then the fork bomb test run and the results have been the different from previous results, which are shown in table 3 below.

Table 3 Testing Results.

	Httperf Only	**Httperf + CPU intensive test**	**Httperf +Network intensive test**	**Httperf+ Fork Bomb intensive test**
Conn. request/ replies	Total=2700 Requests: 2700 Replies: 2700	Total=2700 Requests: 2700 Replies: 2700	Total=2700 Requests: 2700 Replies: 2700	Total=2700 Requests: 193 Replies: 0
Conn. rate	150.0 conn/s (6.7 ms/conn, <=2 concurrent conns)	150.0 conn/s (6.7 ms/conn, <=2 concurrent conns)	150.0 conn/s (6.7 ms/conn, <=2 concurrent conns)	117.4 conn/s (8.5 ms/conn, <=818 concurrent conns)
CPU Conn. time	The lifetime should not be more than 100 PS. it is completely successful because is no more than 100 per second.	The lifetime should not be more than 100 PS. it is completely successful because is no more than 100 per second.	The lifetime should not be more than 100 PS. it is completely successful because is no more than 100 per second.	it is completely failed because is more than 100 per second.
Conn. time	Avarge connection time =0.9	Avarge connection time =0.9	Avarge connection time =0.9	Avarge connection time =1963.9
Reply rate	Reply rate: min 150.0 avg 150.0 max 150.0	Reply rate: min 150.0 avg 150.0 max 150.0	Reply rate: min 150.0 avg 150.0 max 150.0	Reply rate min 0.0 avg 0.0 max 0.0
Reply time	Response = 1.9 ms	Response = 1.9 ms	Response = 1.9 ms	Response = 0.0 ms
Request rate	Request rate = 150.0 conn/s	Request rate = 150.0 conn/s	Request rate = 150.0 conn/s	Request rate = 117.4 conn/s
Net I/O	The average output is 79.8KB/S	The average output is 79.8KB/S	The average output is 79.8KB/S	The average output is 0.6 KB/S
Test Duration	17.997s	17.997s	17.997s	22.994s

According to the above results, the Httperf performance with the Fork bomb test presented the main different results. The number of TCP connections is 2,700 requests as usual, but this experiment see just 193 replies and a test-duration of 22.994s. The connection speed is 117.4 per second, which is slower than the results that seen in the other tests and the lifetime of CPU connection is 0.0. The average number of replies each connection has received is 0.0, which indicates that than the results that seen no replies from the server. The request rate, which is the http, requests were issued by Httperf are 119.0 requests per second while the request size is 75.0 bytes and the request size is 470 bytes. The rate at which the server replied back to the requests is min 0.0 average 0.0 max 0.0 stddev 0.0 (4 samples). The time taken by the web server to respond to the request and the time taken to receive this reply is response 0.0 transfers 0.0.

In summary, the table shows the various isolation benchmark tests compared to the Httperf only test. The fork bomb intensive results are different from other types of results such as the CPU, network, and the Httperf only tests and demonstrate that Docker containers are also susceptible to the same issues of weak isolation and performance as compared to hypervisor-based virtualisation. As seen in [4], the VMware workstation operates perfectly in protecting the VMs in all the stress tests and para virtualisation, like in the case of Xen offers efficient isolation regarding resource degradation. The OS virtualisation as used in containers, however, needs additional configuration controls and even then may show considerable degradation hence is inefficient in some cases. It is therefore clear from this work that more still needs to be done to improve the isolation performance of containers and OS virtualisation.

Isolation in the container is very important to protect user data and make the cloud computing that using container has a secure environment. In the hypervisor, the user has a completely separate machine to ensure the user data privacy and integrity. However, in the container, each user depends on the host OS that could increase the possibility of attacks and loss of user's data. These experiments prove that containers have a weaker isolation performance so a solution to this issue is urgently needed to help build the user trust relationship in cloud computing based on containers.

3.7 Summary

Cloud computing offers several ways in which computing resources are virtualised and utilized depending on the type and level of service being offered. However, the cloud-computing interface makes the user depending on their provider for all these services. This dependency can be fatal in some instances, as the users do not have their data store with them and are forced to rely on the isolation provided by the virtualisation platform to separate their resources in shared tenant platforms. Therefore, customers are not sure whether they can identify a trustworthy cloud provider or not.

Container-based virtualisation offers many advantages over hypervisors in terms of speed, efficiency, workload, performance, and cost. However, the hypervisor still provides better isolation and security than containers. This chapter has provided two experiments that demonstrate the weak isolation provided by containers when compared with hypervisors. Container virtualisation has weak isolation compared with traditional VMs, which needs to be improved to make containers more secure in cloud computing. Moreover, containers do not isolate users inside the containers.

TAIC model aim to develop an isolation solution for container-based virtualisation that would help to create a trust-based relationship between cloud service providers and cloud users. The next chapter provide that system to add fine-grained access control policies and provide system isolation to container-based virtualisation platforms. This will give users additional confidence in the confidentiality, integrity, and availability of the data stored in the cloud. Further, due to the centralized nature of data stored in cloud infrastructures, my proposed design will minimize data leakage and improve monitoring.

4. TRUST ARCHITECTURE FOR ISOLATION IN CONTAINER (TAIC) DESIGN

4.1 Introduction

The critical aim of the design is that it must increase the user's level of trust in cloud services. The Trust Architecture for Isolation in Containers system (TAIC) allows the user to increase their level of trust in cloud services and build a trust relationship between providers and users. This will be done by providing trustworthy services that solve the isolation issue in container-based virtualisation. Further, due to the centralized nature of data stored in cloud infrastructures, the proposed design will minimize data leakage.

TAIC uses a specific approach to limiting container access to authorised users to enforcing isolation in container-based virtualisation. This will provide a higher level of container isolation to protect customers' data and make it secure and private. Our approach for limiting access to containers is based on policies that involve developing an enhanced authorisation plug-in and adding each user in the system, which enables them to perform specific tasks on the containers such as view, modify, create or delete files.

This chapter will explain how TAIC system works to enhance trustworthy authentication and privacy in container-based cloud systems. The first section will describe how the design decisions were evaluated to arrive at a final set of fundamental building blocks for TAIC model before describe the design approach and the system itself. Finally, this chapter presents a use case that illustrates the operation of the trust-based system.

4.2 TAIC Design Approach

This project proposes the development of a security system to address one of the key shortcomings in container-based virtualisation by using rules to isolate the guest VMs from each other. The concept of rules inside the policy file will be used as it relies on defining and assigning users to policies that provide permissions. This method enables the individual user to perform a specific task in a container such as view, create, or modify a file.

The mechanism have adopted to do this is through adding access control in the Docker daemon through an authorisation plug-in. Access control policies can be configured dynamically by the Docker administrator through using authorisation plug-ins for managing access to the Docker daemon. The developer can add a plug-in to the installed Docker daemon and does not need to restart the Docker daemon.

In the Docker, there are two types of plug-ins, which are authorisation and authentication plug-ins. The authentication plug-in is completely different from the authorisation plug-in. The authentication plug-in allows the user to make contact with the Docker daemon while the authorisation plug-in accepts or deny the user request to the Docker containers. After registering the authentication plug-in as a part of the Docker daemon [11], each request to the Docker daemon passes in order through this subsystem. According to the basic architecture for the authentication plug-ins, a HTTP request is made to the Docker daemon through the API, which contains the user and command context. The authentication plug-in approves or deny the request to the Docker daemon based on current authentication and command context. Figures 40 and 41 show the authentication allow and deny scenarios based on the existing Docker authorisation plug-in model [62].

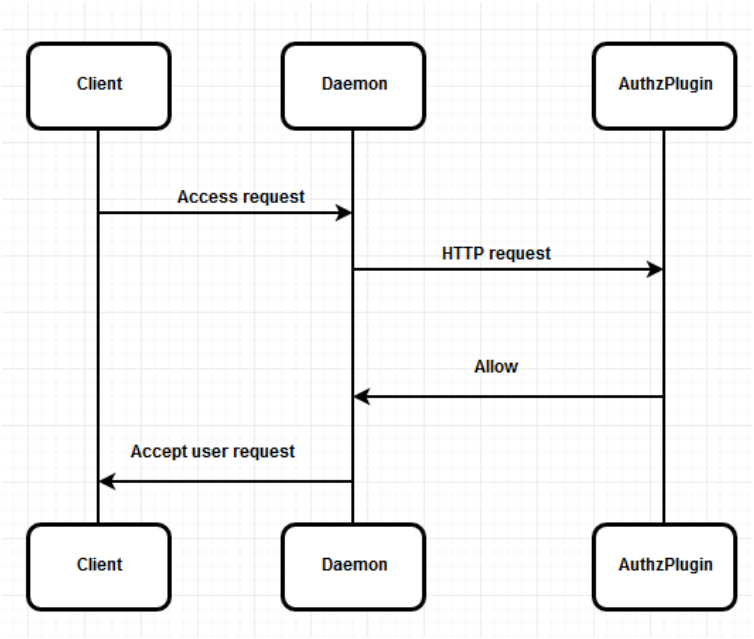

Figure 40. Authorisation allow scenario.

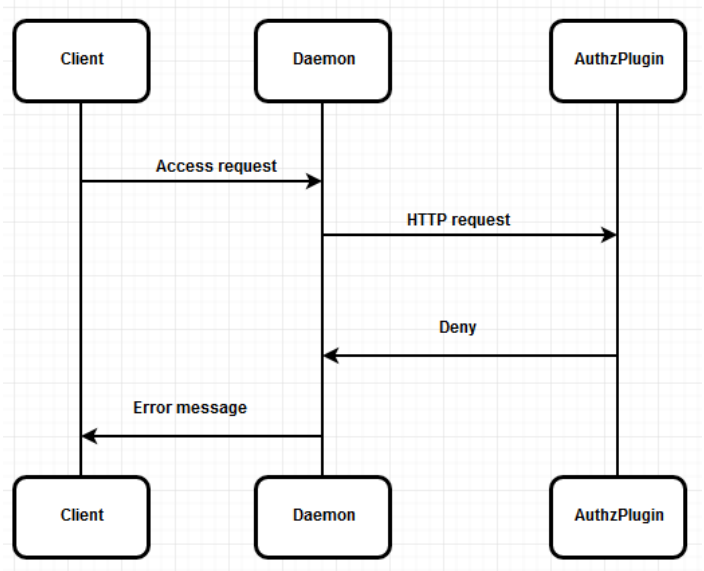

Figure 41. Authorisation deny scenario.

This security system will protect containers from malicious guests and improve the security of data that is stored in them through the addition of policy and provision of data protection. The system has groups of users and objects. Each group of users can perform specific tasks within the containers. User's permission will be provided by rules inside the policy file that are a collection of policies to allow the user to perform specific tasks within the container. User's resources are objects which are accessed by users through some permission. Our system has three levels of the granularity, which are the container, the image, and files within the image (user's resources). Each container has the Docker image which has a user's resources (objects) and the authorisation plug-in is used to check user rules and deny or accept access to the user object.

The most important element of the system will therefore be in designing the authorisation plug-in located in

the Docker daemon that can connect with the Docker container that will accept or deny requests based on the user's policies. However, there are a number of further design issues to be considered at this stage.

4.3 Authorisation plugins

The Docker authorisation plug-in runs directly on the host and is registered as part of the Docker daemon start-up. Before developing of our own plug-in implementation, first investigate and evaluate some existing related works that offer container security plug-ins, such as Twistlock, casbin/casbin-authz-plugin and open-policy-agent/opa-docker-authz. All these examples provide extra security for a Docker container but do not address the isolation issue within the container.

4.3.1 Twistlock

This authorisation plug-in is registered as a part of the Docker daemon and could be run within the host or within containers [89]. There is a single file for all policies that is /var/lib/authz-broker/policy.json and any change in the policy file does not require restarting the system. The policy rules just contain the object (container or image) and action because the Docker API currently does not know to identify the user. In order to set up the Twistlock environment, first one must install the go language with a specific particular configuration and set of godep tools.

The policy file in the Twistlock has the name of the policy, users, and actions. For example { "name":"policy_1", "users": ["user1","user2"],"actions": [container create"]} this means the user1 and user2 can create container. However, the current docker socket does not identify the users. Therefore running the Twistlock authorisation plug-in, should make the users empty in the policy file. For example { "name":"policy_1", "users": [""],"actions": [container create"]} this means any user can create containers only.

4.3.2 casbin/casbin-authz-plugin

The authorisation policy controls the access to the Docker commands [90]. Authentication is not provided by Docker because there is no user information when executing Docker commands. Therefore, there are some plug-in controls for controlling access to Docker. For example, Casbin is a powerful GO-based access control library, which enforces authorisation based on various access control models. In Casbin, a CONF file has an abstract for an access control model based on the PERM metamodel (Policy, Effect, Request, Matchers). Casbin allows the admin to get specific permissions for the users. For example, user1 should read data1 only and users 2 should write on data2 only. P, user1, data1, read that means user1 can read only data1.P, user2, data2, write means that user2 can write data2 only. Two sets of APIs are provided by Casbin for a policy management to manage permissions. Management API and RBAC API are two sets of APIs. Management API provides full support for Casbin policy management and RBAC is provided by a RBAC API. Figure 42 below shows the scenario to allow the user access a specific resource within the container.

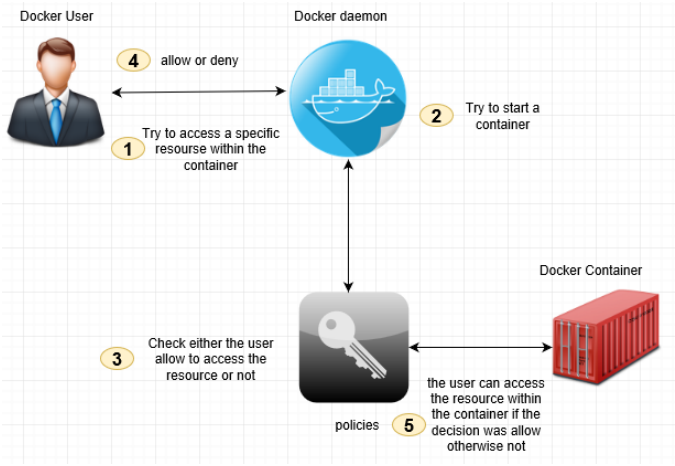

Figure 42. The scenario to allow the user to access a specific resource within the container.

4.3.3 Summary for each authorisation plugin:

This section provide examples of existing Docker authorisation plug-ins. Through our analysis, Twistlock is difficult to make it support any access control model in the future work because the model is not easy for developers to change. This section also investigated other plug-ins such as OPA but they were no longer supported and could not be installed on the latest versions of Docker, and so were not evaluated in this section. The developer can change the model easily by using the Go language and adding roles to the model. A particular file for the model could be changed; therefore, the developer needs to change this file only, instead of changing the whole code.

The existing authorisation plugins do not identify the users because the Docker client communicate with the containers through by Docker API through Docker socket. In Docker, user root must own the Docker socket and the daemon listens to this UNIX socket for requests. The users can run Docker commands without root permission through the Docker group. Each user in the system should be added to the Docker group to allow this user to perform Docker commands without root permission by using sudo. However, The Docker socket does not identify the user because the Docker daemon must be run as root, so root permissions are still provided to the Docker containers. Therefore, the TAIC scrip will be created to identify the users and update the policy file in the authorisation plugin according to the rules for the user that logged in.

4.4 Design Decisions and Technical Approach

The first section will discuss the TAIC design decisions that contain an authorisation plugin and two scripts. The authorisation plugin provides rules for the users to make each user can perform specific actions inside the containers. However, these rules are provided for all users. Therefore, all users can perform the same actions inside the containers. Therefore, TAIC is created to ensure each user can perform different actions inside containers. The TAIC script is created to identify the users who logged in the host and update the policy file in the authorisation plugin according to their identity. Therefore, the TAIC script allows each user to do different actions from others. The administrator could, for example, make each user perform a specific task inside the containers. Finally, a TAICSSH script is created to limiting remote access requests and ensure only authorised users to access the host.

4.4.1 Suggested system approach

First, TAIC model need to consider how and where to deploy the Trust Architecture for Isolation in Containers (TAIC) in the system. TAIC works within the Docker environment that has an authorisation plugin and two scripts to improve the isolation issue in container-based virtualisation. The authorisation plugin has policies to ensure the user can only perform specific actions inside containers. This architecture could be a help to improve the security in the container.

The Docker socket in the current version of Docker does not identify the user because the Docker daemon must be run as root, so root permissions are still provided to the Docker containers. All users should have the same policies that are provided by the authorisation plugin. However, in this project, each user could have the same or different policies from other users in the system. Therefore, a TAIC script is created to identify the users and update the policy file in the authorisation plugin according to the rules for the user that logged in. A TAICSSH script is also created to accept or deny the remote request connection.

User's policies could be configured through an authorisation plugin in the Docker daemon that is located in the host. The administrator can select rules to assign to the new user and edit or delete user rules through this authorisation plugin. This proposed system may be composed of multiple hosts, and each host has one authorisation plugin, as shown in Figure 43 below. The authorisation plugin will obtain the user request data and determine the decision for each container according to the user policy.

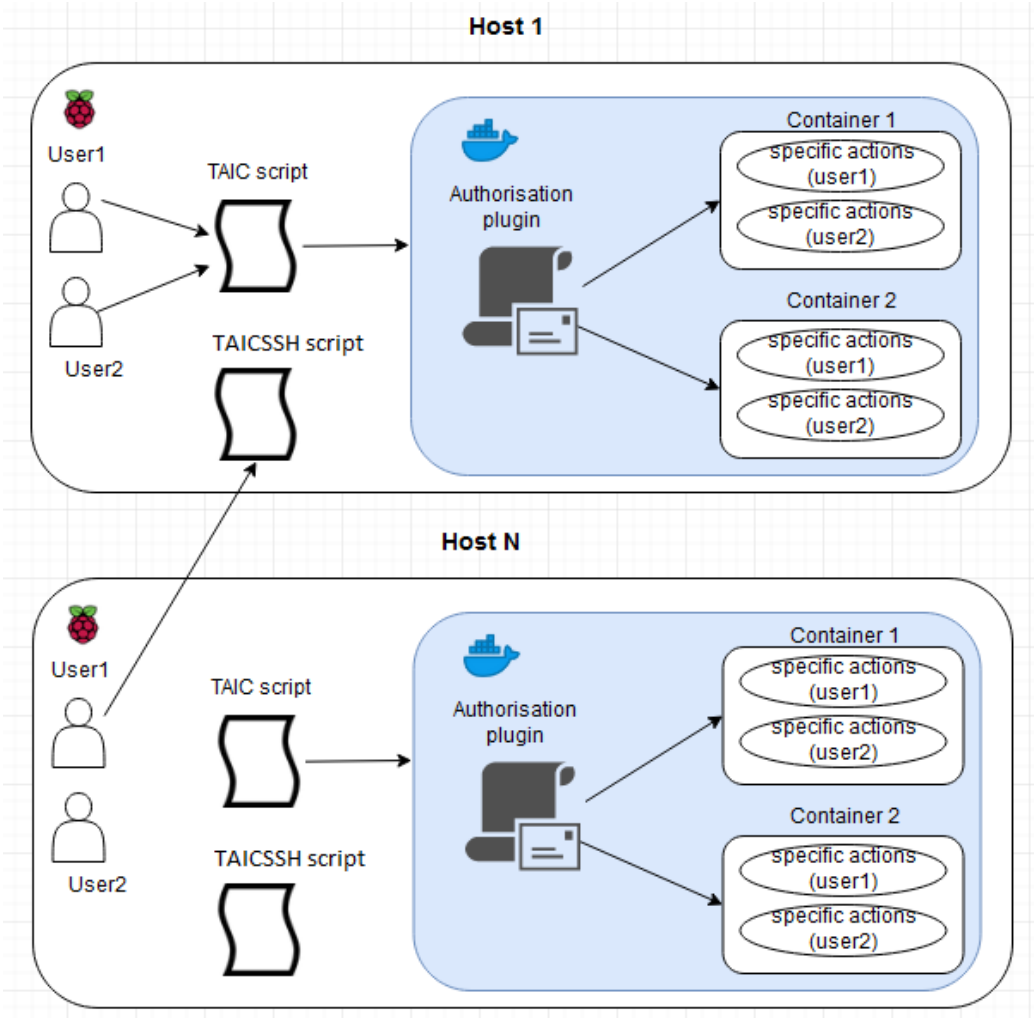

Figure 43. Proposed Trust Architecture for Isolation in Containers (TAIC) model

In this prototype, the containers rely on the host to make the access decision because the authorisation plugin runs in the host, and the container itself does not have any authorisation functionality. This approach needs to deploy just one security model in the host, and so is more scalable as it can accommodate any number of containers and users in each host. Each host could have the same or different policy files depending on the deployed containers. The authorisation plugin will be registered as part of the Docker daemon within the host. When the user requests access to a specific resource within a Docker container, the user should make contact with the Docker daemon beforehand. The Docker daemon obtains this request through the CLI or via the Engine API, which passes the request to the authorisation plugin. The user name, container ID, and action are contained in the user request, and the authorisation plugin will allow or deny the user request.

In the current version of Docker, The socket does not identify the user because the Docker daemon must be run as root. Therefore, TAIC script is created that executes at startup to identify the users. This script runs automatically to identify who logged in to the system and update the policy file according to the user automatically. Through the authorisation plugin, each user can then get rules that allows them to perform specific tasks within the containers. Moreover, this script is protected so only the administrator can access or edit it.

Moreover, TAICSSH script is created to remote connection to allow or deny the SSH access for particular users. This script also executes at startup and runs automatically when the user login. This script is protected. Therefore, only the administrator can access or modify this script.

The administrator should deploy one authorisation plugin and two scripts in the host that is more efficient if the system has a massive number of containers. Moreover, each user could not access any container because the authorisation plugin runs before the user accesses the container to decide if they can access the resources inside the container or not. However, the drawback would occur if the authorisation plugin is compromised or hacked because it means the malicious user could access all containers in the system. Moreover, the authorisation plugin in the host could affect the performance because it could lead to delays in the users' access decisions and increase the response time.

4.4.2 Data and policies

The Trust Architecture for Isolation in Containers (TAIC) model supports a range of user policies to control access to containers. These policies will be deployed by the administrator to control user behaviour in the Docker container. Figure 44 shows the Trust Architecture for Isolation in Containers (TAIC) overview.

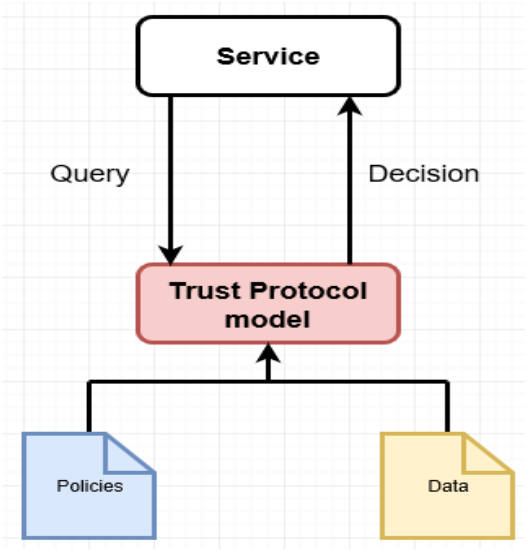

Figure 44. Trust Architecture for Isolation in Containers (TAIC) overview.

Each deployed image has objects (which could be database, files or applications), which are assigned to users, and the container will apply authentication on an action to control access. The administrator creates the actions and assigns them to users and objects. The host could have a number of different containers, each of which has different objects and actions to separate users. For example, the system has two containers and each container has two objects. Each object has a security level, which has some associated permissions.

Policies

A file will be created which contains users' policies. There are three main factors in the policy file, which are subject, object, and action. Subject is the user who logged in the system. The object is the container that has user resources. Finally, a request method that is read, delete and update in this project will be created as an action. Each user should have one or more rules inside the policy file. The authorisation plug-in check if the user has permission to access the container object or not according to the policy, which contains subject and action.

After these checks, the authorisation plug-in can decide according to the user policies, if the user with the given permission to access the object or not. If the check fails, the authorisation plug-in will deny the user access, otherwise it will authorise the user to execute this action on the container object.

Objects

Each container has objects and each user could perform specific tasks within the container. For example container system, that has multiple services or files. The files could be accessed by only users who have permission to access files and only users who have permission to delete or add files inside the containers can perform deleted or add action. ER system could has multiple service; therefore, the authorisation plugin could provide policies for the users to make each group of users can perform particular actions only within the container.

According for the usergroup1 policies, any users related to usergrouop1 policies can export the contents of a container as a tarball while other users cannot perform this action if they do not have permission to perform this action inside the container.

4.5 TAIC System Design

A Cloud Data Centre (CDC) is made up of a number of clusters, each of which contains a number of servers. In our model, each server has an authorisation plug-in which is added to the Docker daemon. The authorisation plug-in will register as a part of the Docker daemon and grant or deny access to the resources inside each container. User rule assignment will be provided by the administrator. Therefore, when a new container is deployed, the new container should be accessed only by particular users who have a permission to access this container. The authorisation plug-in will be updated to contain all user rules and the action could be performed by the users within it. Figure 45 shows the Container-based Cloud DC System Architecture where each node in the cluster is configured as a host that has a Docker daemon and can support containers.

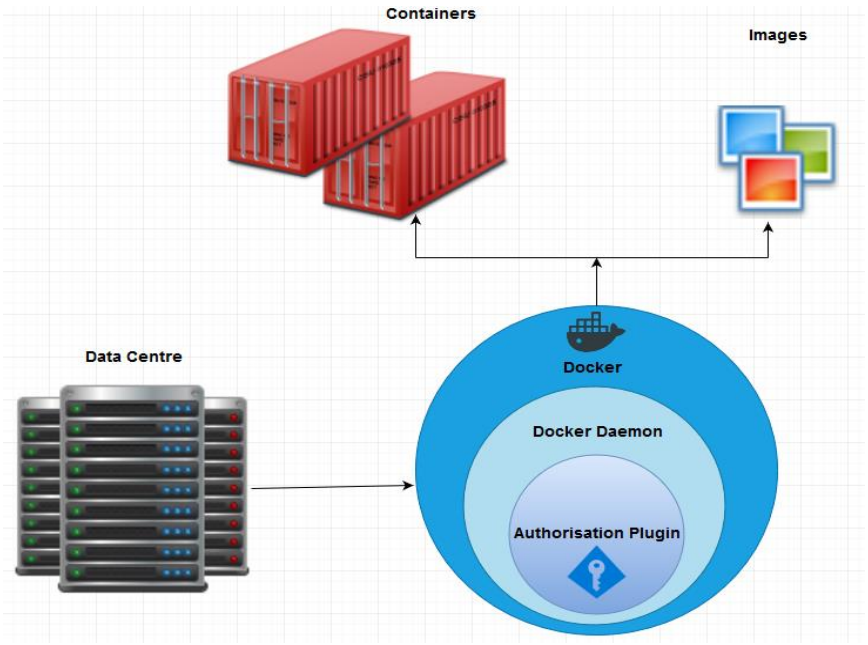

Figure 45. Container-based Cloud DC System Architecture.

All users are classified according to their policies. Therefore, any new user should be registered into one or more rules before providing them access to the cloud computing resources. Each rule has permissions, which are defined according to job functions of rule. In the container-based virtualisation, each server has one or more Docker containers and each container has one Docker image, which has resources for one or more users. The container will be created by the admin based on an image and they will create the trust layer via importing user policies into the authorisation plug-in to isolate the users within the containers. Each user will get access to the right container according to the user policy, all policies will have one or more users, and each user belongs to one or more policies. Figure 46 below shows the system design.

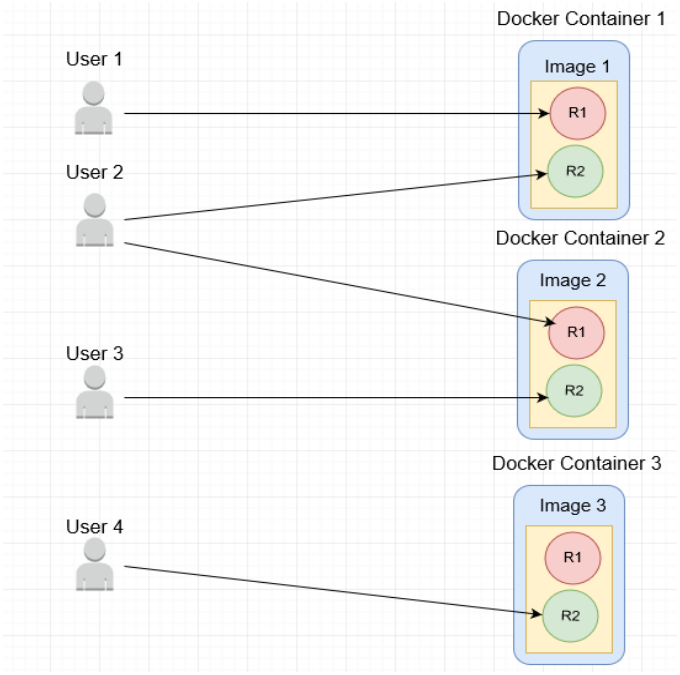

Figure 46. TAIC System Design.

Our approach has several advantages in terms of the granularity and flexibility it offers the cloud system. The existing model provides limited access control to the virtualisation hypervisor only, which helps to protect the guest container from the Linux host and the guests from each other. Moreover, the access control used in the container is based on the existing Linux mandatory access control (MAC) to protect a guest container from the host. The central authority makes the decision in the MAC, not the owner of the resource [47]. Moreover, MAC is still difficult to configure for the users and it has some disadvantages in terms of performance. The information is assigned a level of security in the MAC, which allows the user to access the resources for which they have permission. Only the administrator can make changes in the resource security label because he is not owner of the data.

4.5.1 Module explanation

1. TAIC script

Identify users is provided by the TAIC script. TAIC script is a file in each host and run automatically when the user logged in the host. TAIC script identify the user and open the policy file inside the authorisation plugin to update the user rules. Each user can perform specific actions inside containers otherwise; the user will get an error message from the Docker daemon. Figure 47 below shows the TAIC script idea.

This project has configured the TAIC script in each node to identify the users.

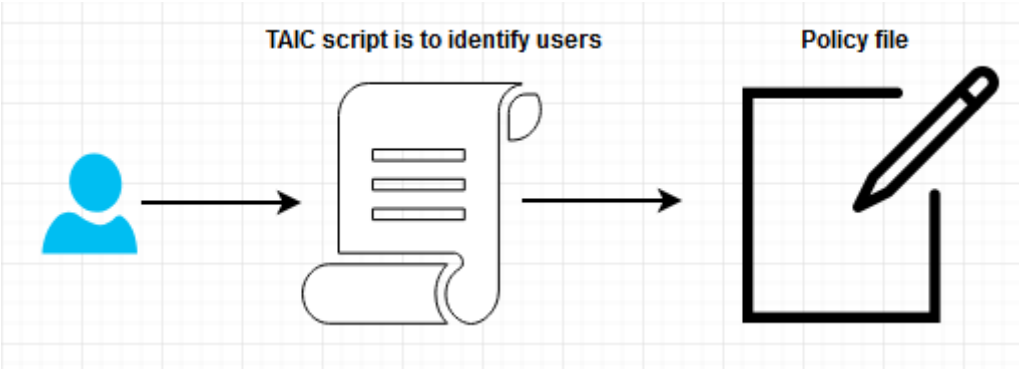

Figure 47. TAIC script

For example, there are two users (user1 and user2). Additionally, there are three containers (container1, 2, and 3). Each user should perform specific actions inside the container. When the user1 login the host, TAIC identifies the user and open the policy file and update the rules inside the policy file.

User 1 could have rules as follows:

- Rule 1 has a Docker PAI version, Container 1 ID, and action. For example, Rules 1 could allow the user 1 to run a specific service inside the container 1.
- Rule 2 has Docker PAI version, Container 2 ID, and action, for example, Rule2 could allow the user 1 to only add a file inside container 2.
- Rule 3 has a Docker PAI version, Container 3 ID, and action. For example, Rule 3 could allow the user 1 only to read the content of the container 3.

User 2 could have the rules as follows:

- Rule1 has Docker PAI version, Container 1 ID and action for example, Rule 1 could allow user 2 to only see the file that added in container1.
- Rule2 has Docker PAI version, Container 2 ID and action for example, Rule 2 could allow the user 2 only see the container 2 history.

2. Authorisation plugin

In our approach, the containers rely on the host to make the authentication decision because users should always make contact with the Docker daemon before access the containers. The authorisation plug-in is Go language programming code, which integrates cleanly with the Docker daemon. The developers can create the authorisation plug-in which is provided by the Docker engine. Trust Architecture for Isolation in Containers (TAIC) authorisation plug-in provides access control to access the resources within the containers through the Docker daemon. The authorisation plug-in accepts or deny the user access according to the user rules.

The authorisation plug-in will be deployed on the host. The API allows the Docker daemon to communicate with the Docker client. The API sends the network request to the Docker daemon. The Docker plug-in has an API that has rules that should be followed by an authorisation plug-in. The API is provided by Docker to interact with the Docker daemon. The Docker client can do everything through the API. The authorisation plug-in has users'

policies that have permissions to access the resources within the containers. The Docker daemon allows the Docker client to make contact with the containers. The Docker daemon configuration has the Trust Architecture for Isolation in Containers (TAIC) system authorisation plug-in which manages access to the Docker resources.

This authorisation plug-in is built by using Go language programming code, Docker API, Docker, Raspbian OS. The authorisation plug-in has the policy file, which contains all the users' policies. The Trust Architecture for Isolation in Containers (TAIC) model file has request definition, policy definition, Effect and Matchers. Subject, object, and action are the three main elements in this authorisation plug-in. The user request is accepted or denied based on these three elements.

The Docker services are managed by Docker daemon. Images, containers, networks, and volumes are Docker objects, which are managed by Docker daemon. The client uses Docker commands that use the API to allow the user to make contact with the Docker daemon. The authorisation plug-in is a part of the Docker daemon.

The user should have one or more rules that allow the user to perform actions on the resources within the container. The rule grant or revoke the user access according to user policy. Each user policy has an object, which is Docker container. When the user requests access to perform an action inside the container. The authorisation plug-in will allow the access if the user has permission to perform this action inside the container otherwise it will deny access. This method makes data more secure and privacy in the Docker container. Therefore, each user should apply the action on the resources to allow the user request access to the Docker container. Figure 48 below provides the authorisation plug-in.

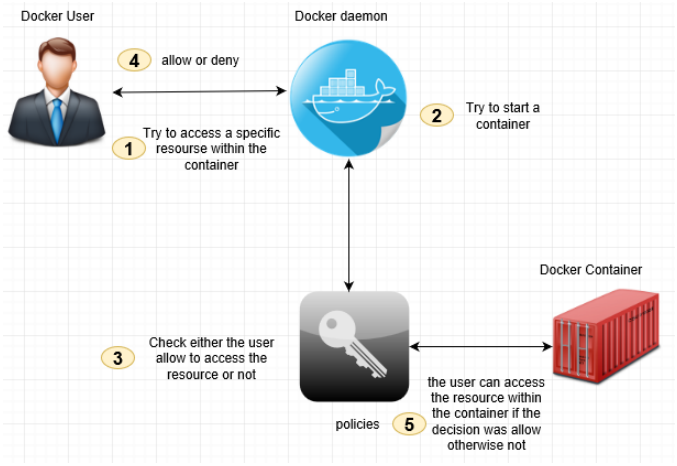

Figure 48. Authorisation plugin

The framework depends on Docker authentication plug-in support. The Docker daemon contains the authorisation plug-in, which is registered as part of the Docker daemon start-up. The authorisation plug-in is responsible for granting or denying the user access to the resources. In this project, each server has Docker daemon, which has an authorisation plug-in.

3. TAICSSH script

SSH allows the user to get remote access. Therefore, enabling SSH could enable unauthorised users to gain remote access, and this feature is not a good idea for security. Therefore, this project is to create TAICSSH script for limiting the remote access connection for authorised users only to improve the security in the system and

protect the container from malicious guests or unauthorised users. The security could be enhanced by limiting remote access to the system that has a large number of users. Figure 49 shows the operation of the TAICSSH script.

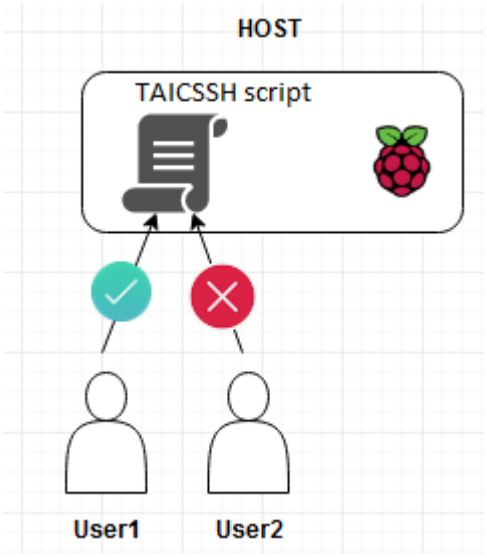

Figure 49 TAICSSH Script

For example, user Bob accesses the host. Two users are user1 and user2 try to get remote access to the host that accessed by Bob. When Bob login the system, TAICSSH identifies the user, who logged in the system and limited remote access for specific users only. The TAICSSH script allows user1 to get remote access and deny user2 request.

User Bob logged in the system:

- Allow user1 to get a remote access.
- Allow administrator to get a remote access.
- Otherwise, deny all other users requests.

User Tom logged in the system:

- Allow user2 to get a remote access.
- Allow administrator to get a remote access.
- Otherwise deny all other users requests.

4.5.2 Proposed system:

The following Figure 50 below shows the steps for accessing the objects within Docker containers.

Step 1: The user logged in the system.

Step 2: TAIC script identify the user who logged in.

Step 3: TAIC script provide rules to the user who logged in.

Step 4: TAIC script update the policy file in the authorisation plugin.

Step 5: TAICSSH script configure the /etc/ssh/sshd_config file and limit the remote access for specific users.

Step 6: The user sends an access request to the Docker client or API for access to a specific object within the Docker container.

Step 7: API allows the user to interact with the Docker daemon.

Step 8: Docker daemon has an authorisation plug-in, which has some policies to provide a secure access to the Docker container objects.

Step 9: The authorisation plug-in checks if the user has policies to perform particular tasks within the containers.

Step 10: The authorisation plug-in makes the decision on whether to accept or deny the user request.

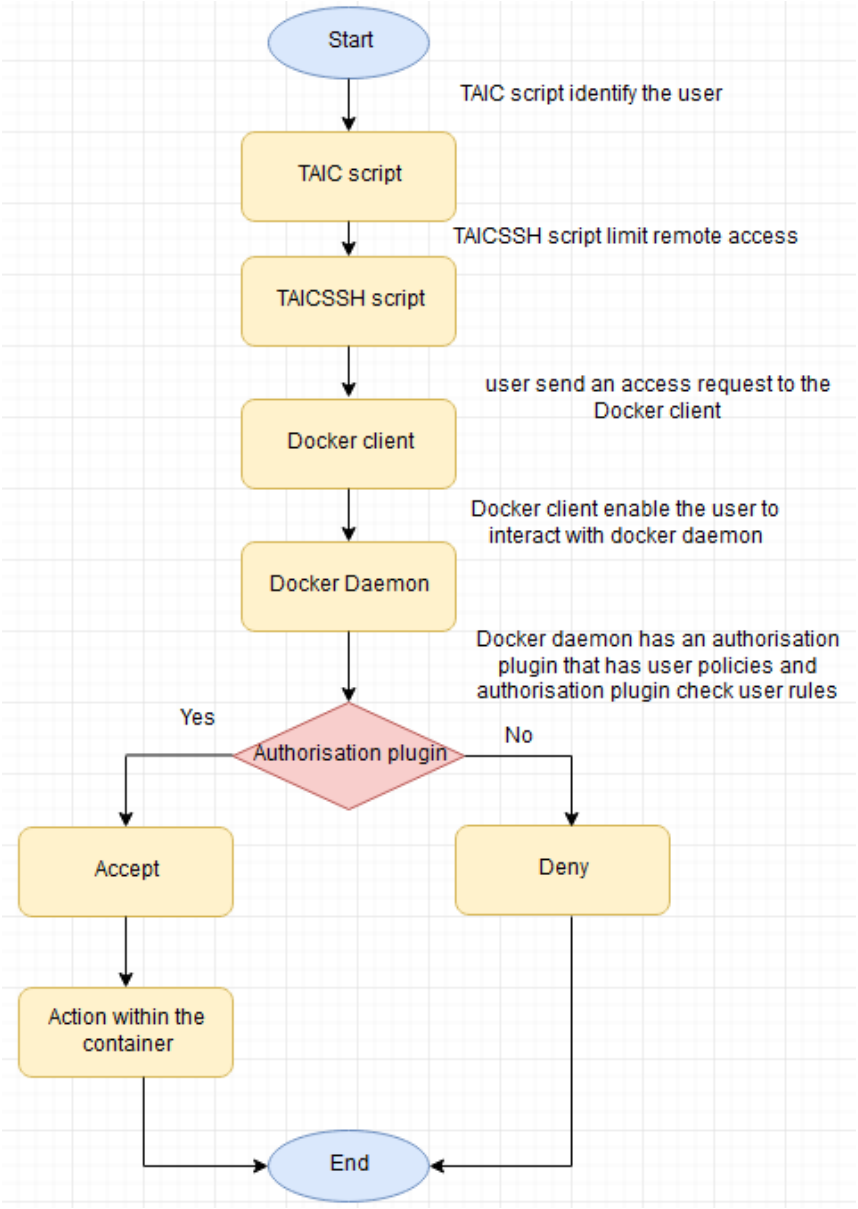

Figure 50. Trust Architecture for Isolation in Containers (TAIC) flowchart

4.5.3 TAIC policies examples:

A user request should contain information on the username, policy, container ID, the object, and action. Then, the authorisation plug-in will make a decision whether to accept or deny the user request. For example, user

Bob. Bob wants to access the container that has the ID 495ad09fc530 and see the low-level information about this container. A typical request in this case would include the following information:

Subject: = "Bob" // the user that wants to access an employee database.
Object: = "495ad09fc530" // the container that is going to be accessed.
Action: = “inspect” // the action that the Bob can see low level information about containers

The authorisation plug-in runs directly on the Docker framework and makes use of the intrinsic plugin support offered by the daemon. The authorisation plug-in is registered as part of the Docker daemon at start-up and contains a user policy file, which allows the administrator to set specific permissions for the users. Policy should be define to make each user can perform specific task within the containers.

Bob could has P, /v1.38/containers/ID/ top, GET policy.
Pi could has P, /v1.38/containers/ID/exec, POST policy.
Tom could has P, /v1.38/containers/ID/inspect, POST policy.

The policy file contains rules that are specified according to the following format. P is the policy type that is the first field in each line. This project has one policy type, which is P (policy definition) that contains object and action) but it is possible to add more than one policy type in the model such as P, P1and P2. For example:

[Policy definition]
P = object, action
The policy definition is matched by policy type so, for the following policy definition:
P, v1.38/ /container/495ad09fc530 /start, POST

P is the policy type and v1.38 is the Docker API version. The object is the container that has ID 495ad09fc530. All rules in the policy file should follow the Docker API references. For example, /containers/{id}/start, POST is to start a particular container. Request data from containers is provided by GET. Send data to server to stop, start or attach containers is provided by POST.

The policy definitions are based on the three factors explained in the previous section, so a policy file in the authorisation plug-in might typically comprise of the following policies:

Bob
P, /v1.38 /container/495ad09fc530 /json, GET
P, /v1.38 /container/495ad09fc530 /start, POST
P, /v1.38 /container/495ad09fc530 /stop, POST
P, /v1.38 /container/495ad09fc530 /attach, POST
P, /v1.38 /container/495ad09fc530 /update, POST
P, /v1.38 /container/495ad09fc530, DELETE

Tom
P, /v1.38 /container/321at06gh667 /json, GET
P, /v1.38 /container/321at06gh667 /start, POST
P, /v1.38 /container/321at06gh667 /stop, POST
P, /v1.38 /container/321at06gh667 /attach, POST
P, /v1.38 /container/321at06gh667 /update, POST
P, /v1.38 /container/321at06gh667, DELETE

Alice
P, /v1.38 /container/987re66yt098 /json, GET
P, /v1.38 /container/987re66yt098 /start, POST
P, /v1.38 /container/987re66yt098 /stop, POST
P, /v1.38 /container/987re66yt098 /attach, POST
P, /v1.38 /container/987re66yt098 /update, POST
P, /v1.38 /container/987re66yt098, DELETE

Docker clients can communicate with the Docker container through the API. The API only sees action and object in the users' requests sent from Docker daemon. Therefore, the TAIC script identifies the user.

4.6 TAIC use case model

The interaction between the user and the TAIC system is described by the use case model. Five main components in the TAIC system are users, TAIC script, TAICSSH script, an authorisation plugin, and containers. The user could be a local user or remote user. The use case is an external vision of the system so that a review some of the actions that can be performed by the user to complete the task can be carried out. In the TAIC model, the user should be log in the system, and then TAIC and TAICSSH scripts will be run automatically. The policy file inside the authorisation plugin will be updated by the TAIC script according to the use rules. Finally, the user can perform specific tasks inside the containers according to the user rules. TAICSSH responsible for accepting or denying the remote user access request. If the remote user gets access to the host, the authorisation plugin allows the remote user to perform specific actions inside the containers. The TAIC use case model shown in Figure 51.

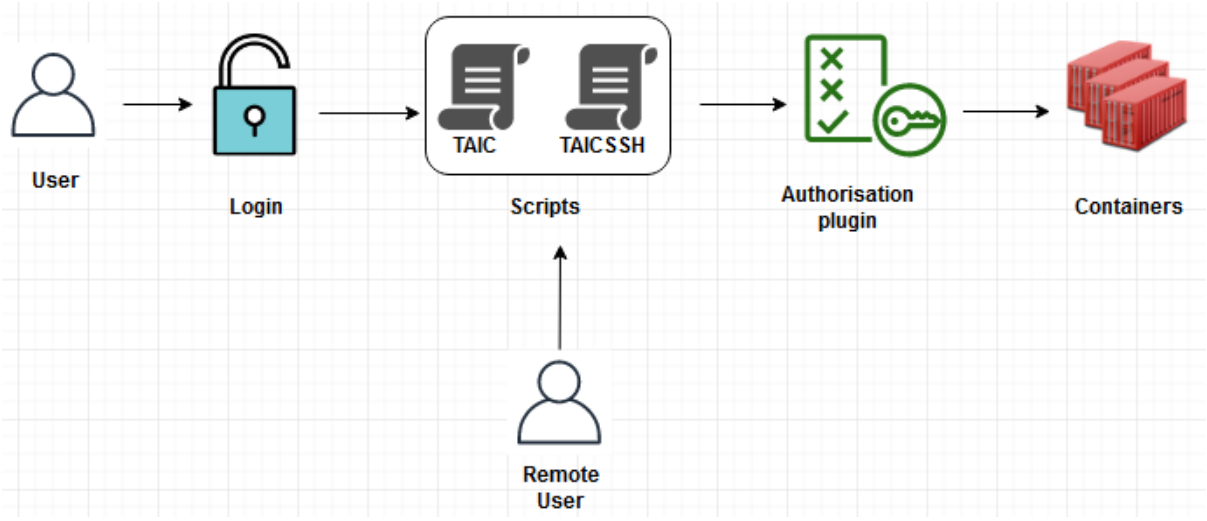

Figure 51 TAIC Use Case Model

4.7 TAIC requirements (Verification)

The trust architecture for isolation container needs hardware and software the hardware used in TAIC mode is raspberry pi devices and SD cards for each Raspberry Pi. The software is Raspbian OS, Docker, Docker swarm, TAIC script, TAICSSH script, GO language, godeep tools, and Authorisation plugin.

4.7.1 Raspberry Pi Cloud (Picloud)

The cost of building a cloud infrastructure can reach up to millions of dollars, which makes it difficult for researchers or academic institutions to afford [38]. Apart from the cost of construction, data centres are also very expensive for any form of big institution to run or manage. Hence, there exists a need for creating different tools that will allow researchers to experiment and study cloud systems. According to [39], Open Cirrus is a global cloud infrastructure that was created by big brands such as HP and Yahoo to enable cloud researchers to study and experiment with cloud related issues. However, one of the key issues identified is the difficulty for researchers from different locations and organizations to access this infrastructure. Therefore, this limits accessibility and is not feasible for everyone. Also, big public cloud providers like Amazon and Microsoft provide free or reduced-cost access to their infrastructure for education and research purposes, but this is usually only limited to using their processing or storage resources without providing access to the underlying infrastructure. As a result, this has led to researchers identifying and creating Cloud environments that will be able to create different simulations for cloud operation and management. Some of the existing simulations include Green Cloud [40] and CloudSim [41]. The Green Cloud simulator has been noted as being one of the most energy conserving and closest models that simulates the communication between a datacentre and network for cloud applications. According to [4], the CloudSim application recreates the cloud environment and the interaction that often takes place between the cloud service provider at the Information as a Service (IaaS) level and the cloud users. Moreover, the study that was conducted by [4] also concluded that CloudSim was able to calculate the level of trust according to individual nodes to identify and evaluate only reliable and trustable nodes for users. Therefore, it was proven that the application was quite reliable in cloud-based study. Figure 52 shows an example of the Raspberry Pi Model B.

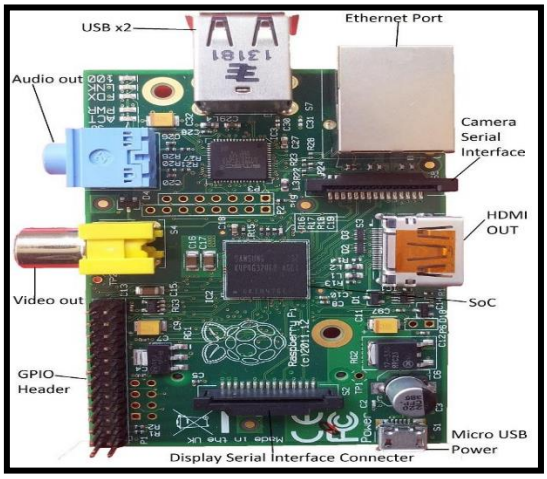

Figure 52. An example of the Raspberry Pi [91].

However, none of these simulators are able to accurately model complicated tasks or applications since there is a limitation in the level of interaction. A cross-layer interaction is required to increase the reality of the cloud simulations, and therefore the Raspberry Pi Cloud approach is explored in our research. The Raspberry Pi is a cheap device that is efficient in terms of power consumption, which increases the affordability for researchers to use. It also comes in a miniature form, which makes it easier and more convenient for users to build in large collections [8]. The Glasgow Raspberry Pi Cloud was one of the first such systems that contains fifty-six Pi devices to fully recreate the cloud's functions, which can be a useful tool for cloud researchers to study different

aspects of data centres' configuration and operation. This approach allows Cloud based services and applications to be fully reproduced right down to the virtualisation manager, which then allows the full set of operations and traffic in Clouds to be modelled.

A recent study conducted by [8] used the Pi Cloud configuration shown in Figure 53 below to simulate a cloud data centre. To simulate a data centre network model, the Raspberry devices were connected using multi topologies using an Open Flow switch. Furthermore, the Raspbian OS was used as it contains more than thirty-five thousand software packages that help to support the requirements for the Pi's hardware requirements. For each node used in the study, three containers was found to be sufficient to model the operation of an ordinary server while still providing reasonable operational performance.

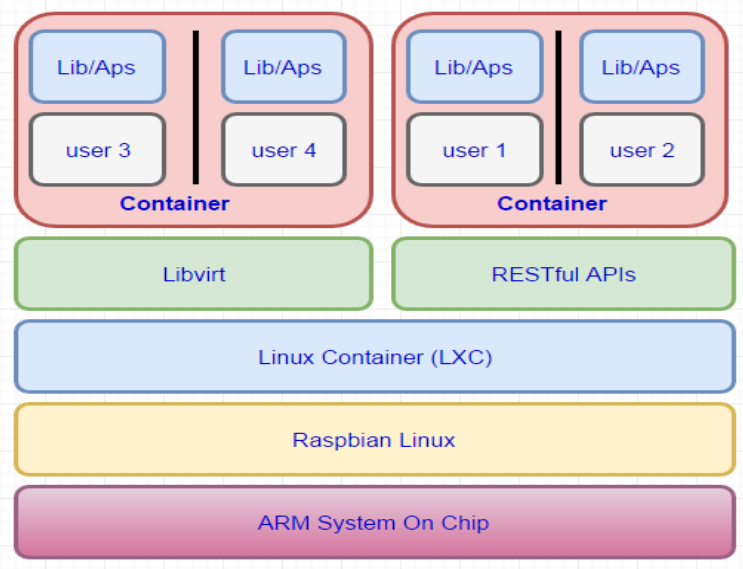

Figure 53. Software Stack for Pi Cloud.

4.7.2 The Project Testbed.

A prototype of our TAIC will be created using a Raspberry Pi Cloud to demonstrate a trustworthy, secure, and scalable measure for cloud services in a practical environment. This project aims to create a limited 'data centre' using four Raspberry Pi nodes running Docker. This will provide an excellent testbed for our work because, despite the limited performance of these devices, they will work in the same logical manner as a fully specified data centre and will demonstrate the same issues regarding the isolation of users and services. This project can, therefore, use this Pi Cloud to evaluate the TAIC that are developing and measure the impact on performance as a result.

The Picloud is created by using Raspberry Pi devices that run Raspbian from a Sandisk 16GB SD card storage. In each host, two scripts and an authorisation plugin as that has [policy file for the users. The method enables the individual user to perform a specific task such as view, create, or modify a file. Therefore, this project provides some security techniques that could help to solve the isolation issue in the container-based virtualisation.

4.8 Summary

The TAIC model is explained in this chapter. TAIC is created by using an authorisation plugin and two scripts. The existing authorisation plugin does not identify the users because, in the current version of Docker, the Docker socket does not identify the users, and Docker daemon must run as root. Therefore, the authorisation plugin is to provide rules for the users and make all users can perform the same actions inside the containers. Thus, the TAIC script is created to identify the user who logged in the system and update the user rules inside the authorisation plugin. Besides, TAICSSH is designed to protect the cluster from unauthorised remote access and make only authorised users to get remote access.

This chapter described existing authorisation plugins and evaluated them for use in our implementation. The authorisation plugin can be configured to allow the user to access a particular area from the user's own containers. The authorisation plugin defines policies for containers to ensure the users can access only the containers which are specifically added to the policy file and which they have permission to access. Twistlock is an authorisation plugin. Twistlock has All policy is under file json /var/lib/authz-broker/policy.json. The user section should be empty to run the plugin. Casbin is the second authorisation plugin. All policy is under filebasic_policy.csv.

Moreover, this chapter provides the positive and negative aspects of the design approach. TAIC provides rules for each user to access the containers. TAIC model has TAIC script, TAICSSH script, and authorisation plugin. The authorisation plugin is created to accept or reject the user request. However, the authorisation plugin does not identify the users. Therefore, TAIC script is designed to identify the users and update the policy file in the authorisation plugin and make each user has the same or different policy from others. TAICSSH script is created to limit remote access requests from the users, and only specific users could have permission to get remote access to the host and perform particular actions inside the containers.

5. TRUST ARCHITECTURE FOR ISOLATION IN CONTAINER (TAIC) IMPLEMENTATION

5.1 Introduction

In this project, each user can execute specific actions within the containers. Therefore, in the implementation chapter, the TAIC model is created that uses two scripts and the authorisation plugin to isolate user containers and add policies for the containers. This model could be deployed now on the currently deployed instances of Docker-based systems. In this model, the TAIC script is to identify the user and update the policy file according to who is logged in. That means that each user or group of users could have rules that allow the users to perform specific actions inside the containers. Besides, the TAICSSH script is created to protect the host from remote access and enable limiting the remote access for particular users only. Therefore, the TAIC and TAICSSH scripts could help to improve the security in the system by protecting user's containers from malicious guests or other users. Then, the authorisation plugin will be used on the host, as outlined in chapter 4. Containers can then be protected by the authorisation plugin that allows the user to perform specific tasks on particular containers only. This will further improve the security since the user can only access and do particular tasks on particular containers for which they have permission. The API allows the Docker daemon to communicate with the Docker client. The API sends the network request to the Docker daemon. The Docker plugin has an API that has rules that should be followed by an authorisation plugin. The plugin API is provided by Docker to interact with the Docker daemon.

In the second section of this chapter outline, the implementation approach has adopted in order to develop a working proof of concept of the TAIC system. Section 5.3 introduces system components and architecture. Section 5.4 provides details about the host implementation and pre-requirement for each node in the host. Therefore, each user in the system can perform specific actions only inside containers. Sections 5.5 and 5.6 provide the technical details about an authorisation plugin, TAIC script, and TAICSSH scripts. Section 5.7 provides the workflow analysis model for the TAIC model. Finally, a summary in Section 5.8.

5.2 TAIC Implementation Approach

Solving the isolation issue within the container is the novelty for this project. However, the communication between Docker client and Docker containers is performed through the Docker daemon that listens to the Docker socket. In the current version of Docker, the Docker socket does not identify the users and the user in the Docker daemon must have root privilege to run. Therefore, this project creates TAIC to identify the users so the authorisation plugin can provide rules that allow users to perform specific tasks within the containers.

Firstly, this model has authorisation plugin that should be configured in each node in the system. Each authorisation plugin has a policy file that provides rules for the users in the system. The authorisation plugin could help to limit the user request to the containers and could improve the security in the system and protect the user's containers. The authorisation plugin could help to protect particular containers by adding the policy to the authorisation plugin. Therefore, the user can do a specific task in this container according to the policy he has. However, the authorisation plugin des not identify the users. Therefore, all users should have the same rules and perform the same action inside the containers.

Secondly, this model has a TAIC script that should be configured in each node in the system. TAIC script is created to identify the user and update the policy file in the authorisation plugin according to the user who logged

in. Moreover, TAIC script makes the policy file empty when the unknown user logged in the host and deleted all rules inside the policy file in the authorisation plugin. Therefore, the authorisation plugin will deny all the request provided by the malicious guest. Therefore, TAIC script could make each user can perform specific actions inside the container and improve the security in the container because any user outside the system cannot achieve anything inside the container.

Finally, the TAICSSH script is created to limit remote access to the host; therefore, only a particular user can get remote access to the host. The TAICSSH script could help to protect the host from the remote request connection. Figure 54 below shows the three-stage approach used by Trust Architecture for Isolation in Containers (TAIC).

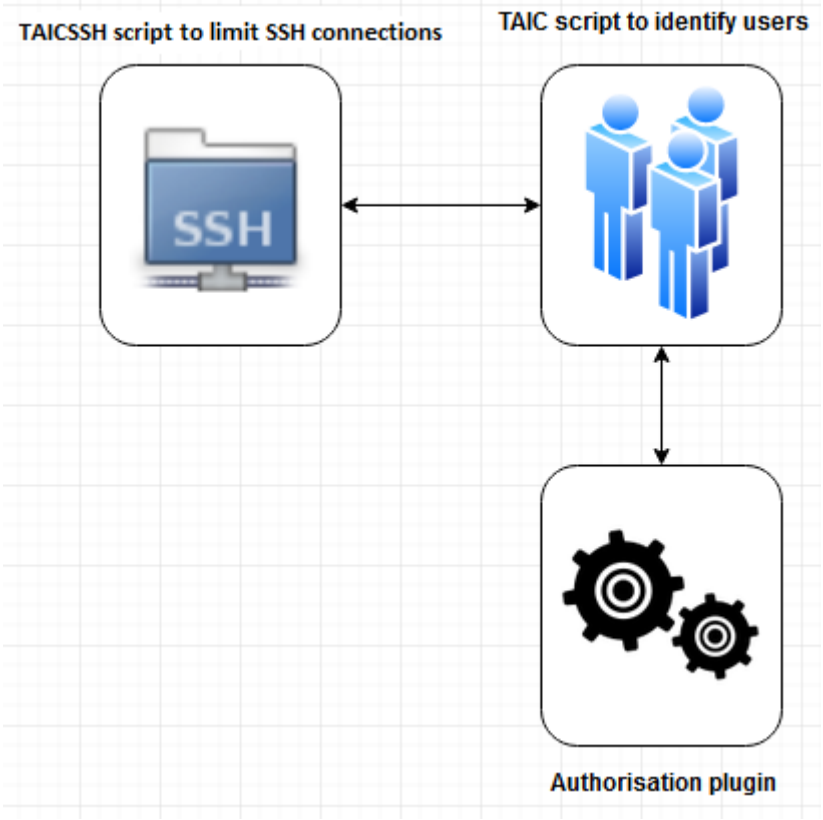

Figure 54. TIAC three-stage implementation (two scripts and authorisation)

TAIC is a system that provides a reliable link between cloud clients and containers. Clients can access the containers via the TAIC system. Current Docker versions utilise sockets to facilitate communication between the daemon and the containers. The Docker socket in the current versions does not identify the user. Therefore, the TAIC model Creates a TAIC script to isolate users inside containers and allow each user to perform specific actions on a particular container only.

5.3 Configuration project testbed

5.3.1 System Components.

In the Picloud, each Docker node is configured with the authorisation plug-in and TAIC, TAICSSH scripts such that any container that it is deployed is subject to the same process. Now, users who utilise the datacentre can specify user authorisation policies and associate them with any container images that they configure on the system. This will provide a consistent model of access that determines which users can access which resources within that specific image. Thereafter, any time an image is deployed into a container on any node with the datacentre, the associated policies will be deployed into the authorisation plug-in alongside the image to control access, as shown in Figure 55 below. This system provides a scalable point of control by providing rules to each users. Once a container is removed, the associated policies are also simply deleted from the authorisation plug-in on the host.

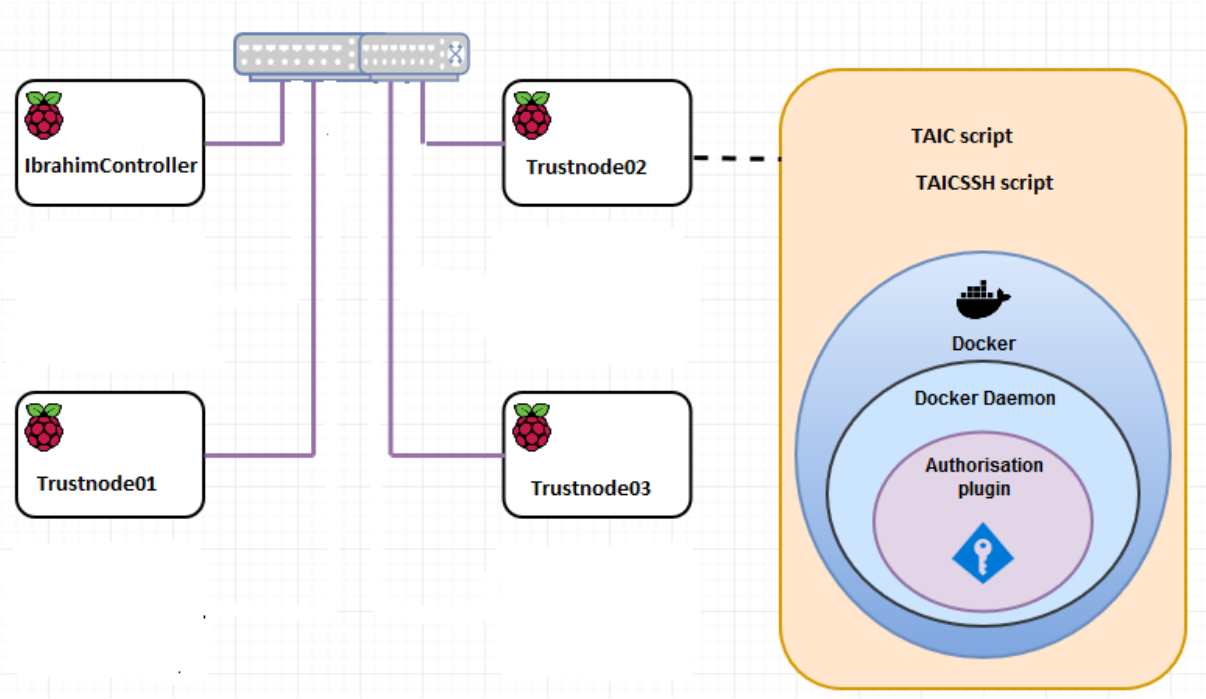

Figure 55. System Architecture

5.3.2 Implemented System Architecture.

The trust architecture is designed to be run in a Cloud Data Centre (CDC) cluster, which may be comprised of a large cluster of servers. As such, the first stage of implementing our work was to build a realistic data centre cluster by using Raspberry Pi devices. Each Raspberry Pi device has the Raspbian OS and Docker installed and is assigned a DHCP IP address. This allows us to develop our solution in a realistic, scalable, and cost-effective environment. The Raspberry Pi nodes communicate with each other through MPI (Messaging Passing Interface) library. Moreover, SSH enables the nodes to communicate with each other and remotely execute commands over an encrypted link between the systems. The three main capabilities provided by secure SSH are secure command-shell, secure file transfer and Port forwarding. The Picloud has a master node that has IP addresses for all cluster nodes / Docker hosts which can run containers as shown in Figure 56 below. Each Docker host is configured with our authorisation plug-in as part of the daemon, which has policies for each deployed container. All containers in the system should be accessed by users through the master node.

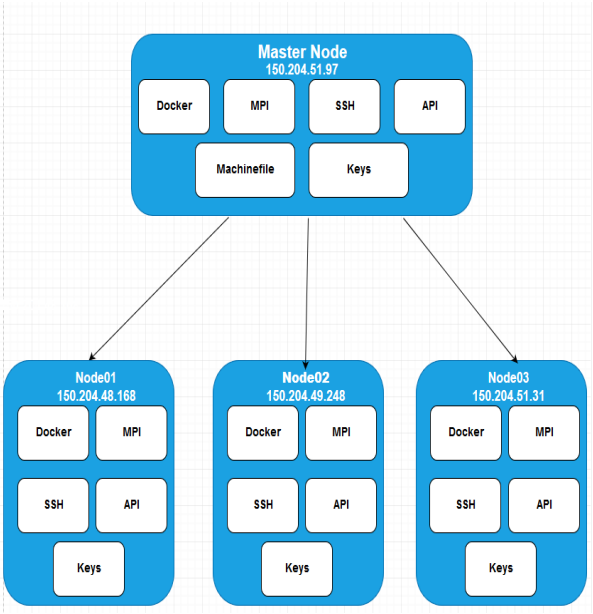

Figure 56. Trusted container Picloud implementation

Docker swarm is used to manage multiple container hosts. TAIC has four nodes; therefore, it could be easy to manage the four node without Docker swarm. However, in the system that has large number of hosts, the Docker swarm could be one of the tools that manges the cluster to solve the managing several containers issue.

The authorisation plug-in has been created using the Go language because this was used by Google in the development of Docker. Each host has an authorisation plug-in, which may be running a number of containers from many users. The administrator assigns the policies for each user.

Each host should be configured by the TAIC to identify the users, which means the Docker users could access particular containers and only perform specific task within these containers. The Docker daemon contains the configuration of the authorisation plugin to accept or deny the user access to the containers. TAICSSH script to limiting remote access request for particular users to improve the security in the cluster.

5.3.3 Raspberry Pi cloud.

This first stage of this project is to build a cluster computer by using raspberry pi devices. This stage allows the users to access any nodes in the system through a master node. All containers in the system could be accessed by users through the master node. In addition, this system has user isolation by using the Docker namespace feature and authorisation plug-in that has policies for users' containers.

The Picloud is created by using MPI (Messaging Passing Interface) library [92]. Parallel applications in a distributed environment could be built by using the message-passing interface (MPI). The most popular one is MPICH-G2. MPI is a communication mechanism used in parallel computing environments to enable all nodes to contact each other.

This project has several resources that are used to build it. Four Raspberry Pi devices that have four SD cards. Four USB cables to connect the Raspberry Pis to the power supply. Switch and five Ethernet cables that connect the Raspberry Pis together.

After installing Ibrahim_config image, the Raspberry Pi configuration should be done by using rasp-config command. Firstly, the hostname is changed to IbrahimController through typing raspi-config in command

terminal and then select network option, which allows the admin to change the hostname. Secondly, enable the SSH through the raspi-config and then select interfacing option and then SSH and enable. Then, the memory should be set to 16 via advanced option that allows the admin to expand file system as well.

The next step is to install MPICH3 and MPI4PY to make each node in the system communicate with others.

Installing MPI4PY as a Python package.

Next, the MPI4PY package needs to be installed, after this step; the project is cloned in Ibrahim-pis-supercomputer image that contains Raspberry Pi OS, Docker Engine 18.06.1-ce, API version 1.38, MPI, MPI4PY, Raspberry Pi configuration and namespace configuration in the Docker daemon. This Picloud has four nodes IbrahimController, Trustnode01, Trustnode02 and Trustnode03. IbrahimController is a master node that has Machinefile that contain IP addresses for all cluster nodes. The IbrahimController IP address is 150.204.51.97, the Trustnode01 IP address is 150.204.48.168, the Trustnode02 IP address is 150.204.49.248, and the Trustnode03 IP address is 150.204.51.30.

5.3.4 Configuring SSH keys for each Raspberry pi nodes.

Secure network communications are provided by Secure Shell (SSH) [93], which provides an SSH private key for each node in the system. This enables each Raspberry Pi to communicate with others without requiring credentials. Therefore SSH key need to be in each node in the system.

IbrahimController node should run the following commands:

```
ssh-keygen
cd ~
cd .ssh
cp id_rsa.pub IbrahimController
```

Configure SSH key for Trustnode01:

```
ssh pi@150.204.48.168
ssh-keygen
cd .ssh
cp id_rsa.pub Trustnode01
scp 150.204.51.97:/home/pi/.ssh/IbrahimController .
cat IbrahimController >> authorized_keys
exit
```

Then, the previous step need to repeat on Trustnode02 and trustnode03. Figure 57 below shows how the Picloud is configured with the SSH keys.

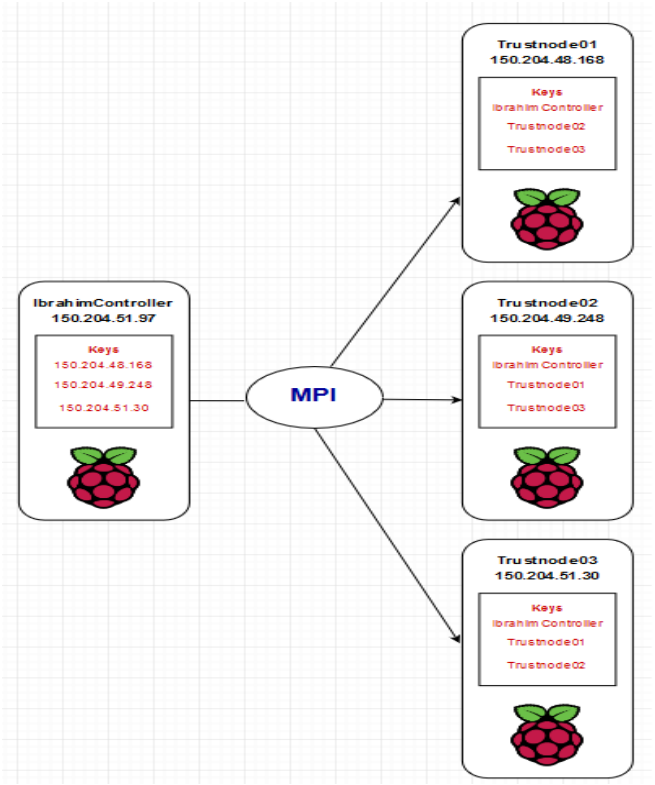

Figure 57. Picloud with SSH keys.

The following steps are then run in the master node to generate the authorised keys on the master node.

```
cd ~
cd .ssh
scp 150.204.48.168:/home/pi/.ssh/Trustnode01 .
cat Trustnode01 >> authorized_keys
scp 150.204.49.248:/home/pi/.ssh/Trustnode02 .
cat Trustnode02 >> authorized_keys
scp 150.204.51.30:/home/pi/.ssh/Trustnode03 .
cat Trustnode03 >> authorized_keys
```

1- The following steps run in the Trustnode01 to generate the authorised keys on the Trustnode01.

```
cd ~
cd .ssh
scp 150.204.49.248:/home/pi/.ssh/Trustnode02 .
cat Trustnode02 >> authorized_keys
scp 150.204.51.30:/home/pi/.ssh/Trustnode03 .
cat Trustnode03 >> authorized_keys
```

2- The following steps run in the Trustnode02 to generate the authorised keys on the Trustnode02

```
cd ~
cd .ssh
scp 150.204.48.168:/home/pi/.ssh/Trustnode01 .
cat Trustnode01 >> authorized_keys
scp 150.204.51.30:/home/pi/.ssh/Trustnode03 .
cat Trustnode03 >> authorized_keys
```

3- The following steps run in the Trustnode03 to generate the authorised keys on the Trustnode03

```
cd ~
cd .ssh
scp 150.204.48.168:/home/pi/.ssh/Trustnode01 .
cat Trustnode01 >> authorized_keys
scp 150.204.49.248:/home/pi/.ssh/Trustnode02 .
cat Trustnode02 >> authorized_keys
```

Once these steps are completed, the Picloud configuration is complete and ready for us to begin our testing.

5.3.5 Create Docker Swarm.

Docker swarm cluster consist of four nodes that are IbrahimController, Trustnode01, Trustnode02 and Trustnode03. IbrahimController is a manager node and others are configured as workers nodes. The following command run on the manager node.

```
sudo docker swarm init --advertise-addr 150.204.51.97
```

Then, run the following command in each worker node to join the swarm cluster.

```
Sudo docker swarm join --token SWMTKN-1-3sjdt2v3f8dqlvn3n0nch3mzw09nisrfrxanw83qviunbn5eqs-e126y7gw0st46e5lkgzak7yge 150.204.51.97:2377
```

The following figures 58 show that the swarm cluster.

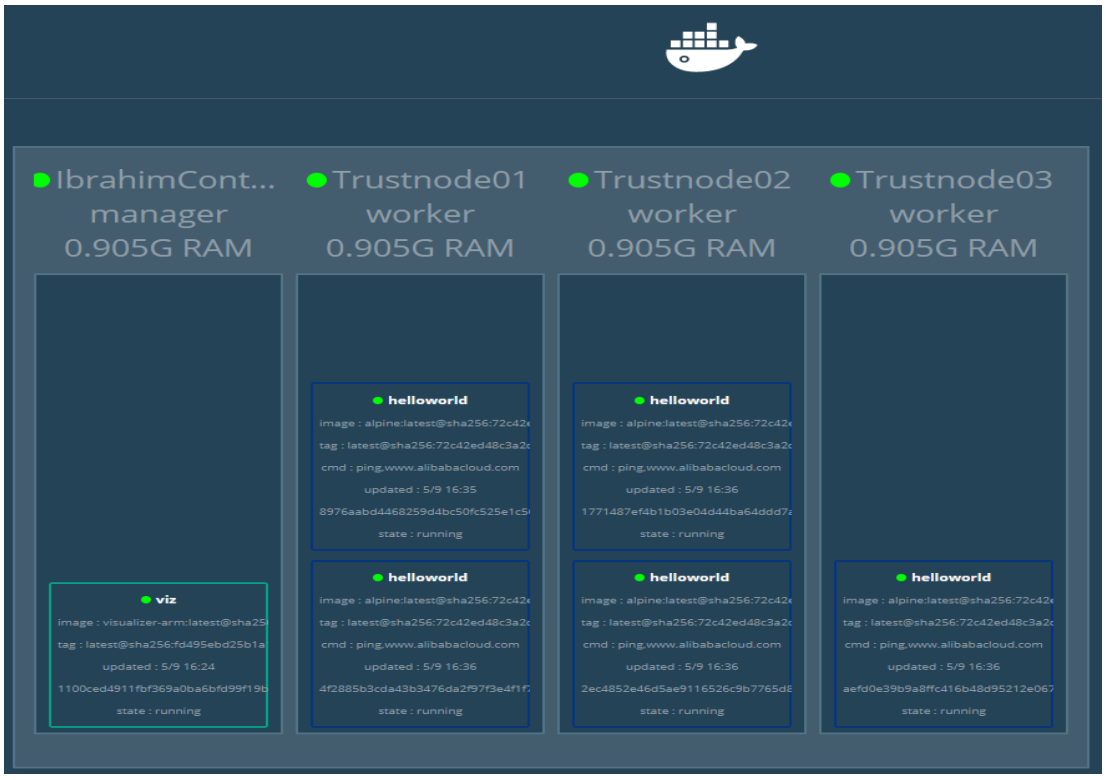

Figure 58 Docker Swarm Cluster

5.4 Host implementation of TAIC model.

This section provides the implementation of TAIC, which contains two scripts and the authorisation plugin.

5.4.1 Host Implementation Overview

Clients can perform specific tasks within the containers by using the TAIC model. Authorisation plugins are a security system running on the host to add policies on the client's containers. Besides, each server in Picloud has an authorisation, TAIC script, and TAICSSH script. Each authorisation plugin should have a policy file that is named basic_policy.csv. The basic_policy.csv file stores containers' policies such as Docker images, containers, version and info. This policy file adds a rule for the containers. Any action on the containers could be performed based on the authorisation plugin policy file. Each authorisation plugin has policy file. Upon a client trying to start a particular container, the TAIC script identifies the users and authorisation plugin verifies containers that could be started by this client according to the client rules in the policy file. The authorisation plugin checks the policy file, which has containers' ids, which could be started by users. Available containers can be started through the authorisation plugin by the user. In the case of this container not being added to the policy file, the authorisation plugin will deny the user request and send an error message to the user. To access the system, the user must register through the admin on the system before trying to access any resources within the containers. If the user does not register on the system, then the user will not be able to start or access any containers. The four main factors to build a TAIC are Docker, Go programming language, Godep tools and Raspbian OS..

Install Docker and Raspbian OS.

Docker installation on Raspbian or Debian OS is quite different from Ubuntu or other OSs. Set up the Docker repository is different from Debian and Ubuntu. As well as the Docker's official GPG key is different from Ubuntu and Debian. However, installing the Docker engine is the same in Debian and Ubuntu. Moreover, Docker EE is not supported on Debian while is supported on Ubuntu. Firstly, apt package should be updated by using "sudo apt-get update". Then, a repository over HTTPS should be used by apt through install packages. The Ibrahim_config image was created to have Raspbian OS, Docker engine Docker Engine 18.06.1-ce and API version 1.38. Therefore, this image has already been set up to have the previous steps.

Install go language

Simple, reliable, and efficient software should be easy to build by using Go language, which is an open source programming language. Raspbian OS need particular configuration to install Go language.

```
wget https://dl.google.com/go/go1.10.2.linux-armv6l.tar.gz
sudo tar -C /usr/local -xvf go1.10.2.linux-armv6l.tar.gz
cat >> ~/.bashrc << 'EOF'
export GOPATH=$HOME/go
export PATH=/usr/local/go/bin:$PATH:$GOPATH/bin
EOF
source ~/.bashrc
go version
```

Install godep tools.

The first step is to install the dependencies package for the Go language (godep)

```
$ go get github .com/tools/godep
$ godep version
```

The TAIC system is implemented using Picloud and Docker containers. Docker images could be provided by using **sudo docker images**, which contains image id, size, name, created, and tag. For example, the system has Ubuntu: latest image. For example, the sudo docker images command provides the following results:

```
REPOSITORY    TAG       IMAGE ID        CREATED       VIRTUAL SIZE
Ubuntu        latest    d0955f21bfbf24   3 weeks ago    188.3 MB
```

The TAIC script, which could help to identify the users and update the policy file in the authorisation plugin according to the user who logged in. TAIC has rules for each user. TAIC and authorisation plugin make the Docker users able to perform particular actions inside containers. Each user can perform specific actions within the container according to the user rules; otherwise, the user will get an error message from the Docker daemon.

5.5 Authorisation Plug-in implantation

The Docker authorisation plug-in is created to allow the users to access only particular containers and perform particular tasks inside these containers. Each authorisation plug-in has a socket and service file [94]. Each authorisation plug-in should have Service and Socket files. The Docker socket file allow the daemon to listen for engine API requests. The Socket file allows the users to communicate with the containers, whereas the service file allows the daemon to accept the connection. Therefore, the Docker demon requires both files to allow the users to communicate with the Docker containers. These two files could be activated by using systemd. For example, the service file (for example /lib/systemd/system/ Trust-protocol-plugin. Service):

```
[Unit]
Description= Trust-protocol-plugin
Before=docker.service
After=network.target Trust-protocol-plugin.socket
Requires=your-plugin.socket docker.service

[Service]
ExecStart=/usr/lib/docker/ Trust-protocol-plugin

[Install]
WantedBy=multi-user.target
```

The socket file (for example /lib/systemd/system/Trust-protocol-plugin.socket):

```
[Unit]
Description= Trust-protocol-plugin

[Socket]
ListenStream=/run/docker/plugins/ Trust-protocol-plugin.sock

[Install]
WantedBy=sockets.target
```

Each plugin has a policy file, which consist of policies to run Docker commands. The Docker daemon obtains the user request and sends it to the authorisation plug-in, which makes the decision according to the policies that

are located in the policy file. This uses a policy file that contains request definition, policy definition, policy effect and matchers.

```
[request_definition]
r = obj, act

[policy_definition]
p = obj, act

[policy_effect]
e = some(where (p.eft == allow))

[matchers]
m = r.obj == p.obj && r.act == p.act
```

R is request definition that consist of object that is a container and action that could be start, access, stop and attach. *P* is policy definition that consist of object that is a container and action that could be start, access, stop and attach. Policy effect is to allow the user to perform a specific task on the container if the result is true. Finally, the matcher will compare the policy rule against the request based on the object or action. Specifically, the matcher will compare *R.obj* (request definition object) to *P.obj* (policy definition object) and so on for the action. A match will be found only when there is an exact correlation between each of the requests and policy parameters. The following sections provide more details for our model that has two scripts and the authorisation plugin.

1. Prerequisites

Each of the authorisation plug-ins consider requires a number of prerequisites to be met prior to installation on a standard Raspberry pi device. These include the Docker engine, Docker API and root or sudo access. This project used Docker Engine 18.06.1-ce and API version 1.38. A specific image, called Ibrahim_config, was created to have Raspbian OS, Docker engine Docker Engine 18.06.1-ce and API version 1.38, as outlined in the Figure 62 below:

```
pi@raspberrypi: ~$ sudo docker version
Client:
Version:      18.6.1-ce
API version:  1.38
Go version:   go1.10.3
Git commit:   e68fc7a
Built:        Tue Aug 21 17:30:49 2018
OS/Arch:      linux/arm
Experimental: false

Server:
Version:      18.6.1-ce
API version:  1.38 (minimum version 1.12)
Go version:   go1.10.3
Git commit:   e68fc7a
Built:        Tue Aug 21 17:30:49 2018
OS/Arch:      linux/arm
Experimental: false
```

This configuration provides a basis on which can install and test the different security plug-ins.

```
Pi@raspberrypi:~$go get github.com/casbin/casbin-authz-plugin
```

```
Pi@raspberrypi:~$ cd $GOPATH/src/github.com/casbin/casbin-authz-plugin
Pi@raspberrypi:~/go/src/github.com/casbin/casbin-authz-plugin $ make

go build –o casbin-authz-plugin

Pi@raspberrypi:~/go/src/github.com/casbin/casbin-authz-plugin $ sudo make install

mkdir –p /lib/systemd/system
onstall –m 644 systemd/casbin-authz-plugin.service/lib/systemd/system
onstall –m 644 systemd/casbin-authz-plugin.socket/lib/systemd/system
onstall –m 755 casbin-authz-plugin.service/usr/lib/docker
onstall –m 644 examples/basic_model.conf /usr/lib/docker
onstall –m 644 examples/basic_policy.csv /usr/lib/docker
```

Now the authorisation plug-in has been built. The next step is to run it.

```
$ cd /usr/lib/docker
$ sudo mkdir examples
$ sudo cp basic_model.conf examples/.
```

This is the model file that contains:

```
[request_definition]
r = obj, act

[policy_definition]
p = obj, act

[policy_effect]
e = some(where (p.eft == allow))

[matchers]
m = r.obj == p.obj && r.act == p.act
```

The users have not been added to the model file because the current versions of Docker provide sockets that do not support users, and the daemon must run as root.

```
$ sudo cp basic_policy.csv examples/.
```
This is the policies file that contains Docker command policy.
```
$ sudo ./casbin-authz-plugin
```

The working directory of the plug-ins should be added to the systems service by using:

```
$ sudo vi /lib/systemd/system/casbin-authz-plugin.service

 [Service]
ExecStart=/usr/lib/docker/casbin-authz-plugin
Workingdirectory=/usr/lib/docker
```

Then, reload the Docker daemon and start the plug-in.

```
$sudo  systemctl daemon-reload
$sudo  systemctl enable casbin-authz-plugin
$ sudo systemctl start casbin-authz-plugin
```

The Docker service file should be edited to run the authorisation plugin.

```
$systemctl edit docker
[Service]
ExecStart=
```

```
ExecStart=/usr/bin/dockerd --authorization-plugin=casbin-authz-plugin
```

After editing the Docker service file, the Docker daemon should be reloaded. Then add the policies in /usr/lib/docker/examples/basic_policy.csv file.

Each plug-in has a policy file, which consist of policies to run Docker commands. The Docker daemon obtains the user request and sends it to the authorisation plug-in, which makes the decision according to the policies that are located in the policy file. This uses a policy file that contains request definition, policy definition, policy effect and matchers.

```
[request_definition]
r = obj, act

[policy_definition]
p = obj, act

[policy_effect]
e = some(where (p.eft == allow))

[matchers]
m = r.obj == p.obj && r.act == p.act
```

R is request definition that consist of object that is a container and action that could be start, access, stop and attach. *P* is policy definition that consist of object that is a container and action that could be start, access, stop and attach. Policy effect is to allow the user to perform a specific task on the container if the result is true. Finally, the matcher will compare the policy rule against the request based on the object or action. Specifically, the matcher will compare *R.obj* (request definition object) to *P.obj* (policy definition object) and so on for the action. A match will be found only when there is an exact correlation between each of the requests and policy parameters.

5.6 TAIC and TAICSSH scripts implementation

5.6.1 TAIC script

This script is created to identify the users because the user does not identify by the Docker socket in the current version of Docker. This script executes at startup automatically. The user will be determined by this script. This script connects to the policy file inside the authorisation plugin. Therefore, after the execution of the script, the user will be identified, and the policy file will be updated. This script should be configured in each node in the cluster.

```
sudo nano TAIC

#! /bin/bash

if [ "$(logname)" == "User1" ]

then

sudo vi /usr/lib/docker/examples/basic_policy.csv -c ':1,$d' -c ':s/$/

(Rule1) \r

(Rule2) \r
```

```
(Rule3)  \r
(RuleN)  '  -c ':wq'
elif [ "$( logname)" == "User2" ]
then
sudo vi /usr/lib/docker/examples/basic_policy.csv -c ':1,$d' -c ':s/$/
(Rule1) \r
(Rule2) \r
(Rule3)  \r
(RuleN)  '  -c ':wq'
elif [ "$( logname)" == "UserN" ]
then
sudo vi /usr/lib/docker/examples/basic_policy.csv -c ':1,$d' -c ':s/$/
(Rule1) \r
(Rule2) \r
(Rule3)  \r
(RuleN)  '  -c ':wq'
else
sudo vi /usr/lib/docker/examples/basic_policy.csv -c ':1,$d' -c ':wq'
```

The script is created to identify the users for example if (logname = user1) then {

1- Open the policy file. (sudo vi /usr/lib/docker/examples/basic_policy.csv)
2- Delete the all rules in the policy file. (':1,$d').
3- Add rule1 \r , rule2 \r …., ruleN.
4- Save the file (:wq)
   }

The last line in the script, which is (sudo vi /usr/lib/docker/examples/basic_policy.csv -c ':1,$d' -c ':wq') means that the script will delete all rules in the policy file for any users does not add to the system. Therefore, any user outside the system cannot get permission to perform anything inside Docker.

5.6.2 TAICSSH script

TAICSSH script is created to ensure that the SSH connection request is secure and has permission to access the containers. The configuration of the file '/etc/ssh/sshd_config' allows the developers to make the SSH access for particular users only. The developer only needs to add the parameter 'AllowUsers' to the '/etc/ssh/sshd_config file.

```
sudo nano TAICSSH
#!/bin/bash
if [ "$( logname)" == "User1 " ]
```

```
then
sudo sed -ie '124,130d' /etc/ssh/sshd_config
echo "AllowUsers UserX  UserY admin" >> /etc/ssh/sshd_config
elif [ "$( logname)" == "User2 " ]
then
sudo sed -ie '124,130d' /etc/ssh/sshd_config
echo "AllowUsers UserZ  UserF admin" >> /etc/ssh/sshd_config
 else
sudo sed -ie '124,130d' /etc/ssh/sshd_config
AllowUsers admin
```

The script is created to limiting remote access requests if (logname = User1) then Add the parameter 'AllowUsers' at the end of /etc/ssh/sshd_config file following by the users that have permission to get the remote connection.

The last line in the script, which is (AllowUsers admin), means that the script will accept admin requests only and reject any request for any users does not add to the system. Therefore, any user outside the system cannot get permission to remote access for the hosts.

When the user logged out of the host, the AllowUsers configuration in the SSH configuration should be clear to make other users could access the host. Therefore, this project created a script that executes before reboot or shutdown to remove AllowUsers configuration and make the host ready to the next user.

```
#!/bin/bash
sudo sed -ie '124,130d' /etc/ssh/sshd_config
sudo vi /etc/systemd/system/sshclear.service
[Unit]
Description=Run Scripts at Stop
[Service]
Type=oneshot
RemainAfterExit=true
ExecStop=/usr/lib/docker/examples/sshclear
[Install]
WantedBy=multi-user.target
```

5.7 TAIC workflow model.

This section is a model is designed to describe the overall shape of the TAIC model. The main objective of this section is to describe the scenario that will be done starting from the user who logged in the system access the containers.

Firstly, the user logged in the system. Then the TAIC script identifies the user and update the policy file in the authorisation plugin according to the user rules that are provided by the administrator. Then the authorisation plugin accepts or deny the user request to the container according to his rules in the policy file. Finally, the user can perform an action inside the container if the authorisation plugin accepts his request. TAICSSH script runs automatically when the user logged in the system and identify the user and allow the remote access for specific users only. Figure 59 below show that the TAIC workflow model.

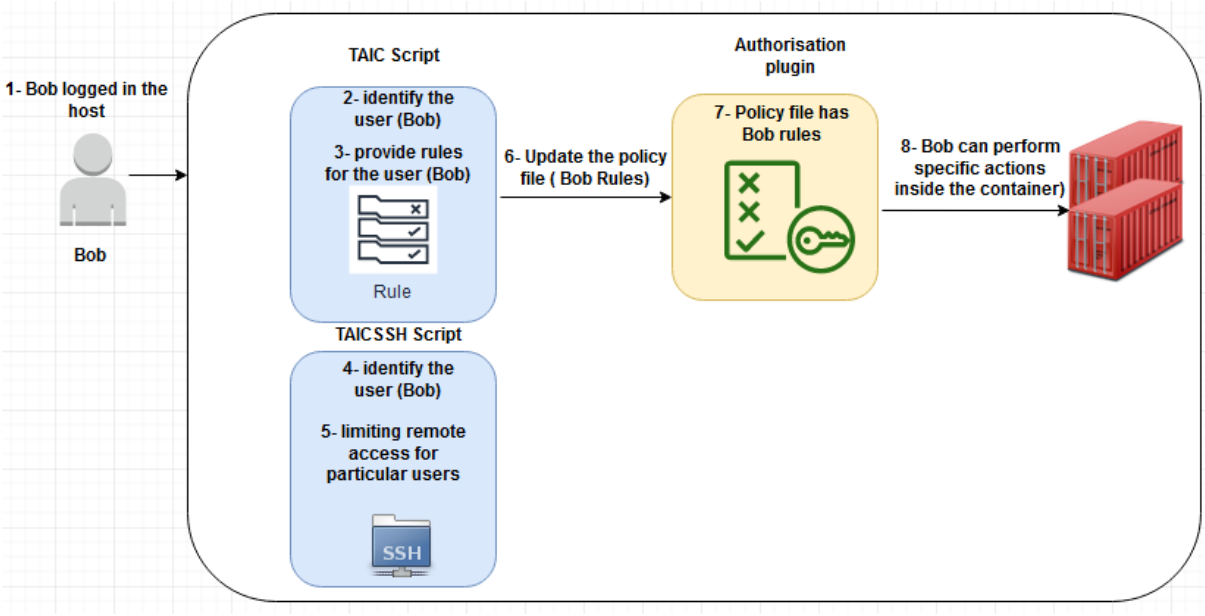

Figure 59 the TAIC workflow model

5.8 Summary

The novelty in this project is to address the isolation issue within the containers. However, the client can only communicate with the container through the Docker daemon that listens to the Docker socket. The Docker socket in the current version does not identify the user. Therefore, the TAIC script is created to identify the users on the host and then update the policy file in the authorisation plugin.

This model could help to improve the security in cloud computing based on container virtualisation. This model has two scripts that are TAIC and TAICSSH scripts. The TAIC script will identify the users and the TAICSSH script will limit SSH access for the remote users. Additionally, the authorisation plugin will protect the Docker container and ensure the user only performs a specific task within the container according to the policy file.

In this project, each node has an authorisation plugin that has a policy file and TAIC and TAICSSH scripts should be configured in each node to isolate the user from each other. The nodes are implemented by using raspberry pi devices and connected by MPI. Docker swarm in implemented to manage the cluster.

6. EVALUATION

6.1 Introduction

The TAIC implementation required five main components, which are the Picloud, Docker, TAIC script, TAICSSH script, and the authorisation plugin. The evaluation of our proof of concept implementation will involve four stages. Firstly, provide a security comparison for three cases that are a Docker cluster with Docker swarm only, a cluster with Docker swarm and the authorisation plugin and the last case is cluster with Docker swarm and TAIC model. This section will also describe the testing objectives. Secondly, case 1 will be evaluated in section 6.3 and case 2 will be evaluated in section 6.4. Then, evaluate the TAIC model (case 3) in section 6.5. TAIC ensures that each user can perform specific actions inside the containers and only access the permitted containers. Then, this section provides testing for the TAIC and TAICSSH scripts to make sure that users can get the same or different rules from each other and to limit remote access requests for particular users only. Finally, the analysis of our results is provided at the end of this chapter.

6.2 Testing Overview (Validation)

6.2.1 Security comparison

This section provides a security comparison for three cases. The first case is a basic deployment that is just a cluster of Raspberry Pi nodes with Docker swarm. In this case, any user inside or outside the host can access any container in the system. Moreover, any user can get remote access to the host and can perform any action on the container without restriction. The testing results for the first case are provided in section 6.3.

The second case consists of the same Raspberry Pi cluster with Docker swarm and an authorisation plugin installed in the hosts. In this case, only very granular policies will allow/deny access to any user [95]. Any user inside or outside the host can still access any container in the system according to the policies in the authorisation plugin. All users have the same policies; therefore, all users can perform the same actions on the containers. Moreover, any user can get remote access to the host and can perform specific actions on the container. The testing results for the second case are provided in section 6.4.

The last case is the TAIC model that consists of the Raspberry Pi cluster with Docker Swarm, the authorisation plugin, and the TAIC script and TAICSSH script. In this case, the TAIC script identifies the user and updates the policy file according to who is logged in to the host. In this case, each user could have different policies from the others. Therefore, each user can perform specific actions on the container, which could be the same or different from other users. Any user inside or outside the host can access any container in the system according to his policies in the authorisation plugin. Moreover, particular users can get remote access to the host and can perform specific actions on the container according to his policies in the authorisation plugin. The testing results for the third case are provided in section 6.5.

Figure 60 and Table 4 shows a security comparison of the three cases.

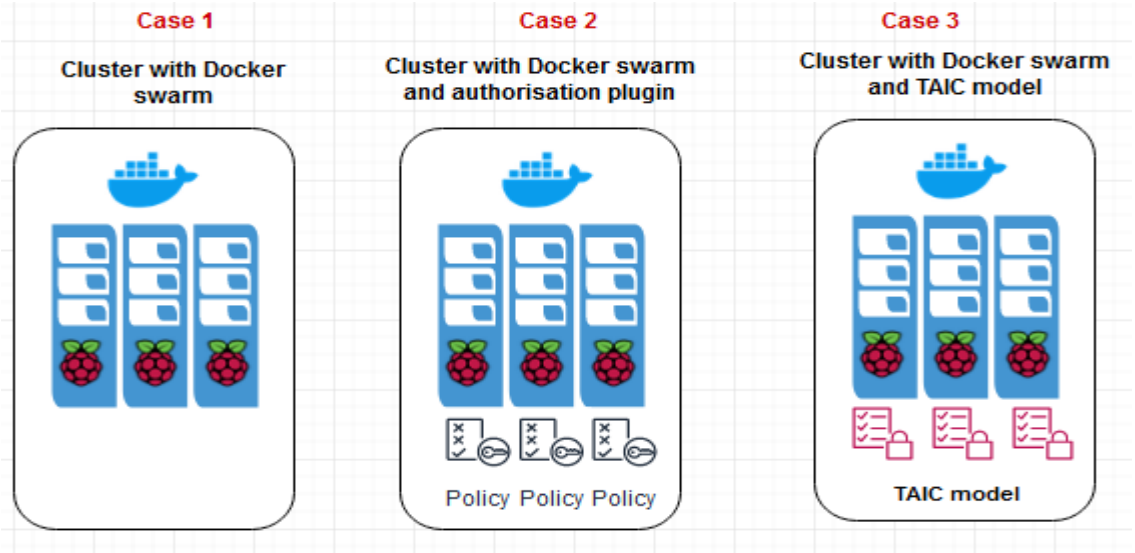

Figure 60 three cases for security comparison

Table 4 three cases security matrics

	Case 1: Cluster with Docker swarm	Case 2: Cluster with Docker swarm and authorisation plugin	Case 3: Cluster with Docker swarm and TAIC model
Any user inside the host can access any container in the system	Yes	No	No
Any user outside the host can access any container in the system	Yes	No	No
Any user can get remote access	Yes	Yes	No
The same policies for all users to allow access	No	Yes	No
The same policies for all users to deny access	No	Yes	No
Identify the users	No	No	Yes
Limiting remote access	No	No	Yes
Each user has specific policies to perform specific tasks inside containers	No	No	Yes
Each user could has the same or different policies	No	No	Yes
Particular users cannot perform specific actions on the containers.	No	No	Yes
Any user not added to the system cannot perform any action inside the containers	No	No	Yes

6.2.2 Testing objectives

This section provides information about the security testing objectives, which combines several combinations, repetitions, and experiment results. Our testing objectives are as follows:

- Each user can get the same or different policies from others. Therefore, each user can do a specific task inside the containers according to the policy the user has. In existing deployments, all the users can only have the same policies, but in the TAIC mode, each user can have specific policies. Moreover, a user cannot perform any tasks not added to the policy file.
- Configure each node as a host to improve the security in the Picloud. That means if one node fails, other nodes are not affected. Moreover, the developer can protect a number of nodes and make them for particular

users only. For example, trustnode02 has sensitive data. Thus, the developer could allow only specific users to access trustnode02 or deny all requests for this node.

The Raspberry Pi nodes on my network are IbrahimController, trustnode01, trustnode02 and trustnode03 and four users are in this system who are Pi, Bob, Tom, and Alice. Pi is the administrator that can access or modify the policy file, TAIC and TAICSSH scripts. Bob, Tom, and Alice are users in the system. Table 5 shows the security testing table.

Table 5. Propoesd Tests

Number	Testing
Testing 1	Each user has different rules from others.
Testing 2	Each user could perform different actions inside the container from others.
Testing 3	User has permission should perform a specific action inside container (inside or outside the host).
Testing 4	User does not have permission should not perform a specific action inside container (inside or outside the host).
Testing 5	User is not added to the system should not access the container.
Testing 6	User does not have remote access permission should not get remote access to the host.
Testing 7	User has remote access permission should get remote access to the host.
Testing 8	User should not edit or access TAICSSH script, TAIC script or policy file (inside or outside the host).
Testing 9	Admin should edit or access TAICSSH script, TAIC script or policy file.

To evaluate the three cases outlined above, two containers are created in each host with three users. Moreover, one additional user named Leo, is created as an outside user that tries to access the container. This project has done the security testing in six containers, as shown in table 6 below.

Table 6 six containers to validate three cases.

Container ID	Trust node number
94903b6c7b9	Trust node 1
ad5168051372	Trust node 1
ffd5603da220	Trust node 2
1590f1e7bb96	Trust node 2
3694024b1ea3	Trust node 3
612e4f6965dd	Trust node 3

6.3 Case 1: Cluster with Docker swarm only

To evaluate Case 1 that is a cluster with Docker swarm only, all users' containers are first deployed in the Trust nodes (Trustnode01, Trustnode02 and Trustnode03). In each trust node, there are two containers. Moreover, all users should be able to perform any action in the containers because the Docker daemon does not have an authorisation plugin to limit user requests to the containers. Figure 61 below shows the container in each node.

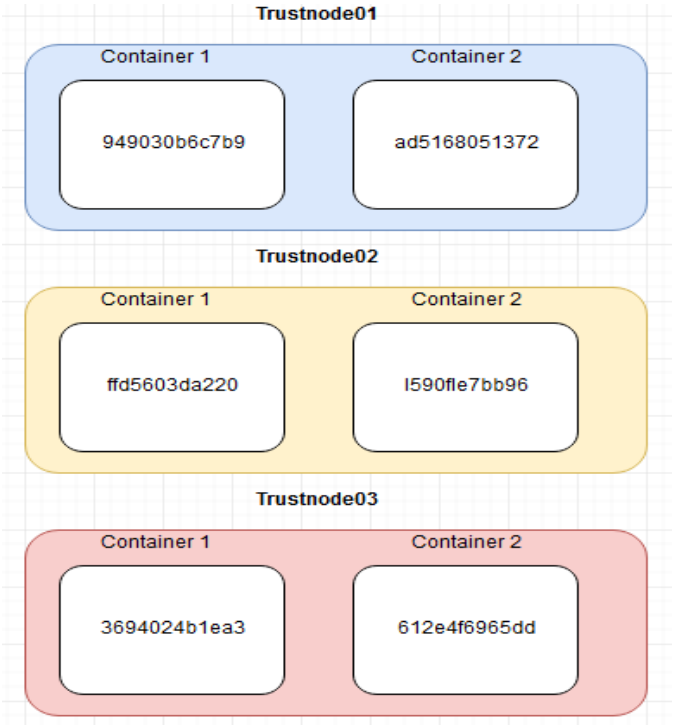

Figure 61 Containers in each node

In all trust nodes, all users can perform any action inside all containers in the system. Any user inside or outside the host can access any container in the system without any restriction. The Docker daemon allows all user's requests to perform any action inside the container. Docker Swarm is used as the cluster management tool, which means that the multiple systems that host containers should be managed and launch as a single body. Docker Swarm could deploy various containers automatically inside the host machines but it does not provide isolation for users inside the container. Figure 62 below shows the actions that could be performed by three users inside the containers in all nodes in the system.

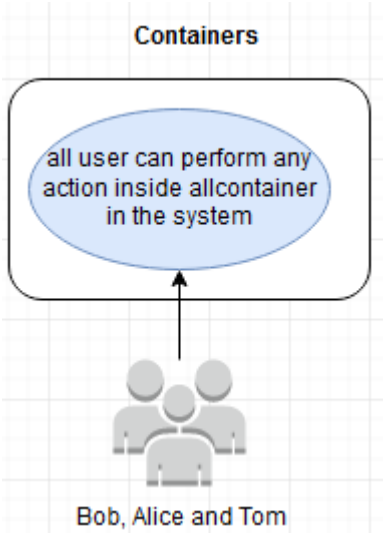

Figure 62 the actions that could be perform by three users inside the containers in all nodes in the system.

Case 1 does not have an authorization plugin; therefore, none of the users need rules to access the containers. Moreover, any user can get remote access to any node in the system unless the /etc/ssh/sshd_config file is configured for particular users only. Figure 63 below shows the testing results for the case 1 cluster with Docker swarm only which demonstrates that only one of the tests passes successfully.

Number	Testing	Result
Testing 1	Each user has different rules from others.	No
Testing 2	Each user could perform different actions inside the container from others.	No
Testing 3	User has permission should perform a specific action inside container (inside or outside the host)	No
Testing 4	user does not have permission should not perform a specific action inside container (inside or outside the host)	No
Testing 5	User is not added to the system should not access the container	No
Testing 6	User does not have remote access permission should not get a remote access to the host	No
Testing 7	User has remote access permission should get a remote access to the host	Yes
Testing 8	User should not edit or access access TAICSSH script, TAIC script or policy file (inside or outside the host)	No
Testing 9	Admin should can edit or access TAICSSH script, TAIC script or policy file.	No

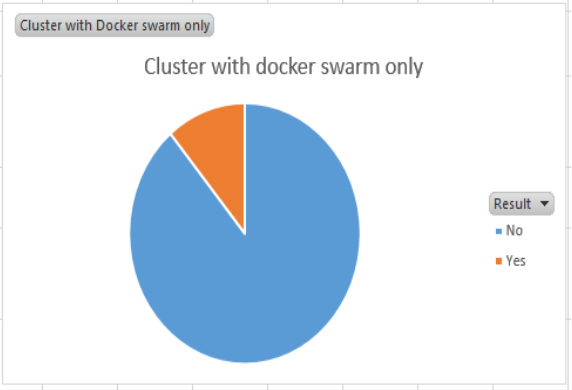

Figure 63 Testing results for case 1 cluster with Docker swarm only

6.4 Case 2: Cluster with Docker swarm and an authorisation plugin

To evaluate Case 2 (a cluster with Docker swarm and an authorisation plugin in each host) first deploy the containers in each node as described in Figure 60 above. In this case, all users should have the same rules to perform the same tasks within these containers because the Docker daemon does not identify the user.

A. *Trustnode01:*

Trustnode01 has two containers, 94903b6c7b9 and 5168051372. In this Trustnode, all users get the same rules to perform the same action inside the two containers. For example, all users can start container 94903b6c7b9 and see the files that are added or deleted in container 5168051372. Table 7 provides the rules that are applied in this case for all users.

Table 7 Users rules in Trustnode01

	Container 1 (94903b6c7b9)	Container 2 (5168051372)
Rules	P,/V1.38/containers/94903b6c7b9/ start, POST	P,/V1.38/containers/5168051372/ changes, GET

Figure 64 below shows the actions that could be perform by the three users inside the containers in trustnode01.

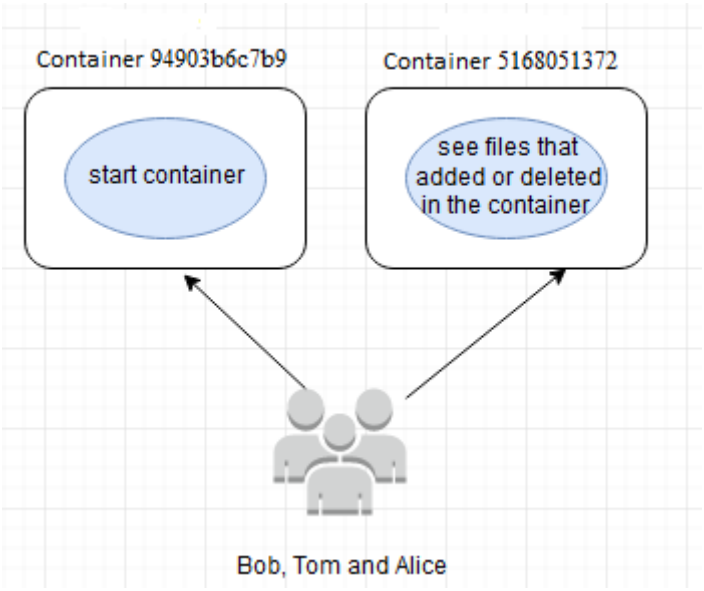

Figure 64 Actions for all users in Trustnode01

The Docker daemon returns an error message to the users if they try to perform any action not specified in the policy file. Therefore, all users can only start container1 otherwise; the Docker daemon will send an error message to the users. In container 2, all users can only see the files that are added or deleted to the container otherwise the user will again get an error message from the Docker daemon.

B. *Trustnode02*

Trustnode02 has two containers ffd5603da220 and 1590f1e7bb96. In Trustnode02, all users can start and attach to container ffd5603da220 only. All user can only export the contents of container 1590f1e7bb96 as a tarball. Table 8 provides the rules that are applied in this case.

Table 8 Users rules in Trustnode02

	Container 1 (ffd5603da220)	Container 2 (1590f1e7bb96)
Rules	P,/V1.38/containers/ ffd5603da220/ start, POST P,/V1.38/containers/ ffd5603da220/ attach, POST	P,/V1.38/containers/1590f1e7bb96/ export, GET

Figure 65 shows that the actions that could be perform by three users inside the containers in trustnode02.

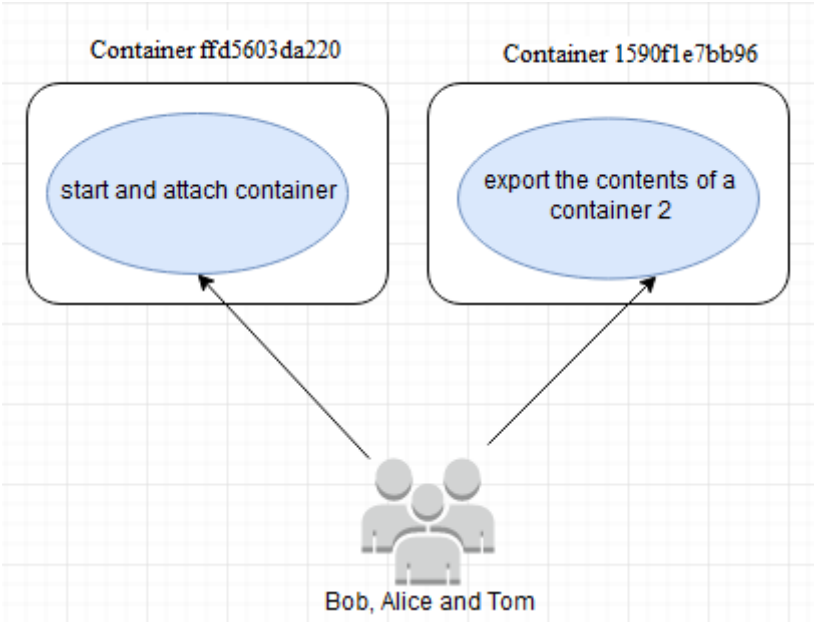

Figure 65 the actions that could be perform by three users inside the containers in trustnode02.

C. Trustnode03

Trustnode03 has two containers (3694024b1ea3 and 612e4f6965dd). In Trustnode03, all users can start and update container 3694024b1ea3 only. No user can perform any action inside container 612e4f6965dd. Table 9 provides the rules that applied in this case for all users.

Table 9 Users rules in Trustnode03

	Container 1 (3694024b1ea3)	Container 2 (612e4f6965dd)
Rules	P,/V1.38/containers/ 3694024b1ea3/ start, POST P,/V1.38/containers/3694024b1ea3/ update, POST	None

Figure 66 shows the actions that could be performed by the users inside the containers in trustnode03.

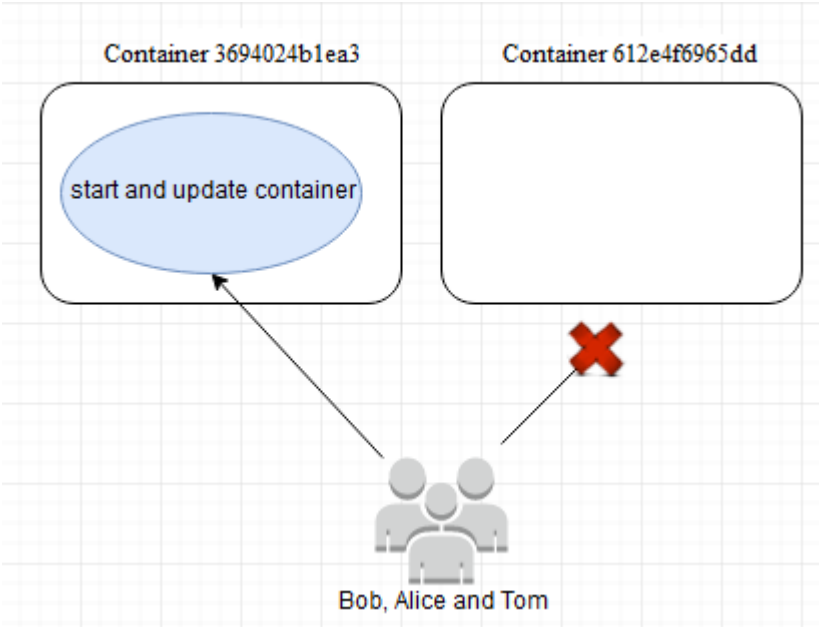

Figure 66 the actions that could be perform by three users inside the containers in trustnode03.

D. Analysis

In all Trustnodes, all users should have the same rules and make the same actions inside the containers. This authorisation plugin therefore provides limited access to the users' containers. This improves security in the system and maintains the user containers' privacy. However, this case does enable each user can perform specific actions inside the container and cannot identify the user to make each has different rules from others. In Case 2, any user can get remote access to any node in the system unless the /etc/**ss**h/sshd_config file is configured for particular users only. Figure 67 below shows the testing results for case 2 cluster with Docker swarm and authorization plugin. In this case , three out of the nine tests are successful.

Number	Testing	Result
Testing 1	Each user has different rules from others.	No
Testing 2	Each user could perform different actions inside the container from others.	No
Testing 3	User has permission should perform a specific action inside container (inside or outside the host)	Yes
Testing 4	user does not have permission should not perform a specific action inside container (inside or outside the host)	Yes
Testing 5	User is not added to the system should not access the container	No
Testing 6	User does not have remote access permission should not get a remote access to the host	No
Testing 7	User has remote access permission should get a remote access to the host	Yes
Testing 8	User should not edit or access access TAICSSH script, TAIC script or policy file (inside or outside the host)	No
Testing 9	Admin should can edit or access TAICSSH script, TAIC script or policy file.	No

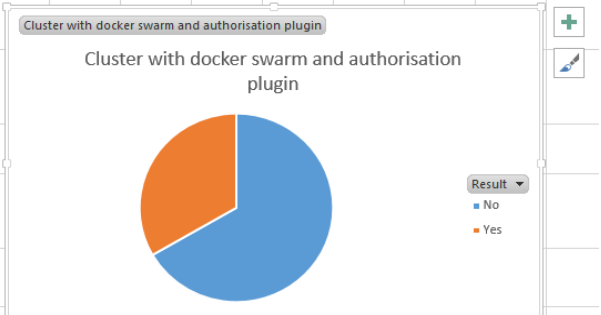

Figure 67Testing results for case 2 cluster with Docker swarm and authorisation plugin

6.5 Case 3: Cluster with Docker swarm and TAIC model

6.5.1 TAIC model testing for users that added to the system.

This evaluation aims to demonstrate that each user can only perform specific tasks inside the containers, as indicated in the policy file. The users will attempt to access each trustnode and try to perform particular actions inside the containers. This test is again evaluated using three trust nodes and six containers.

A. Trustnode01:

Trustnode01 has two containers as before (94903b6c7b9 and ad5168051372) and in this Trustnode, each user got rules to perform specific actions inside the two containers. For example, Bob can add or delete files inside two containers and start the containers. Tom can only see the files added in the two containers. Alice can only see low-level information about the two containers. Table 10 below shows the rules that are applied in this case for each user.

Table 10 Users rules in Trustnode01

	Container 1	Container 2
Bob	P,/V1.38/containers/94903b6c7b9/ start, POST P,/V1.38/containers/94903b6c7b9/ exec, POST	P,/V1.38/containers/ ad5168051372/ start, POST P,/V1.38/containers/ ad5168051372/ exec, POST
Tom	P,/V1.38/containers/94903b6c7b9/ changes, GET	P,/V1.38/containers/ ad5168051372/ changes, GET
Alice	P,/V1.38/containers/94903b6c7b9/ json, GET	P,/V1.38/containers/ ad5168051372/ json, GET

Bob can only start two containers and delete or add files inside the containers; otherwise bob gets an error message from the authorisation plugin because he only has these four rules. Tom can only see the files that are added or deleted to the containers; otherwise he got an error message. Finally, Alice can only see low-level information about two containers otherwise she receives the error message. Figure 68 below shows that the actions that could be performed by three users inside the containers in trustnode01.

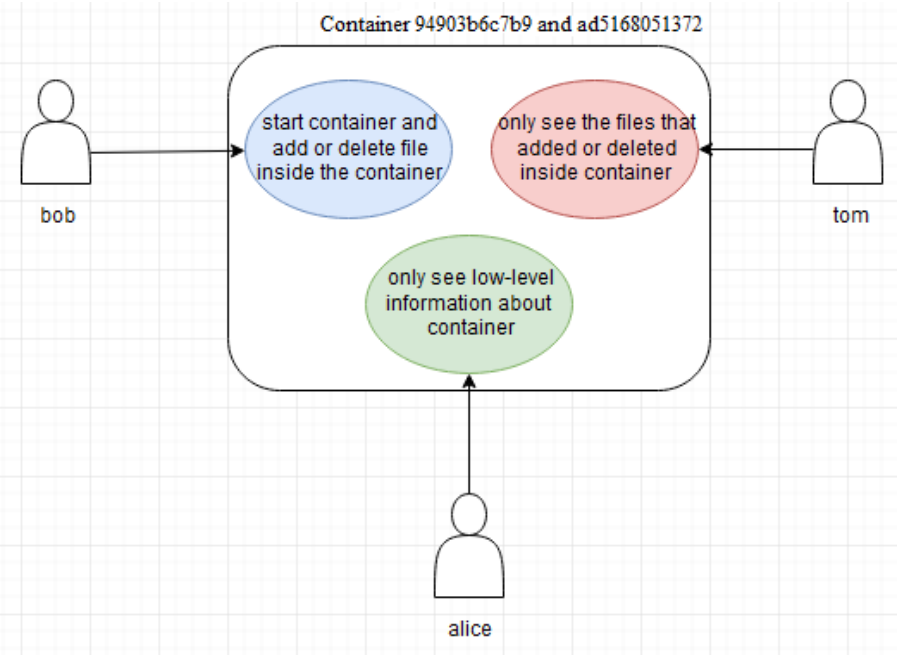

Figure 68 Trustnode01 users' permissions

B. Trustnode02:

Trustnode02 again has the same two containers (ffd5603da220 and 1590f1e7bb96) but now with user-specific rules. Bob can export the contents of container1 as a tarball and can start container2. Tom can only see the files added in container1 and stop container2. Alice can see low-level information about containers1 and start and add or delete files inside container2. Table 11 below shows the rules that are applied in this case for each users.

Table 11 Users rules in Trustnode02

	Container 1	Container 2
Bob	P,/V1.38/containers/ffd5603da220/ export, GET	P,/V1.38/containers/1590f1e7bb96/ start, POST
Tom	P,/V1.38/containers/ fd5603da220/ changes, GET	P,/V1.38/containers/1590f1e7bb96/ stop, POST
Alice	P,/V1.38/containers/ fd5603da220/ json, GET	P,/V1.38/containers/1590f1e7bb96/ start, POST P,/V1.38/containers/1590f1e7bb96/ exec, POST

Figure 69 below shows that the actions that could be performed by each of the three users inside the containers in trustnode02.

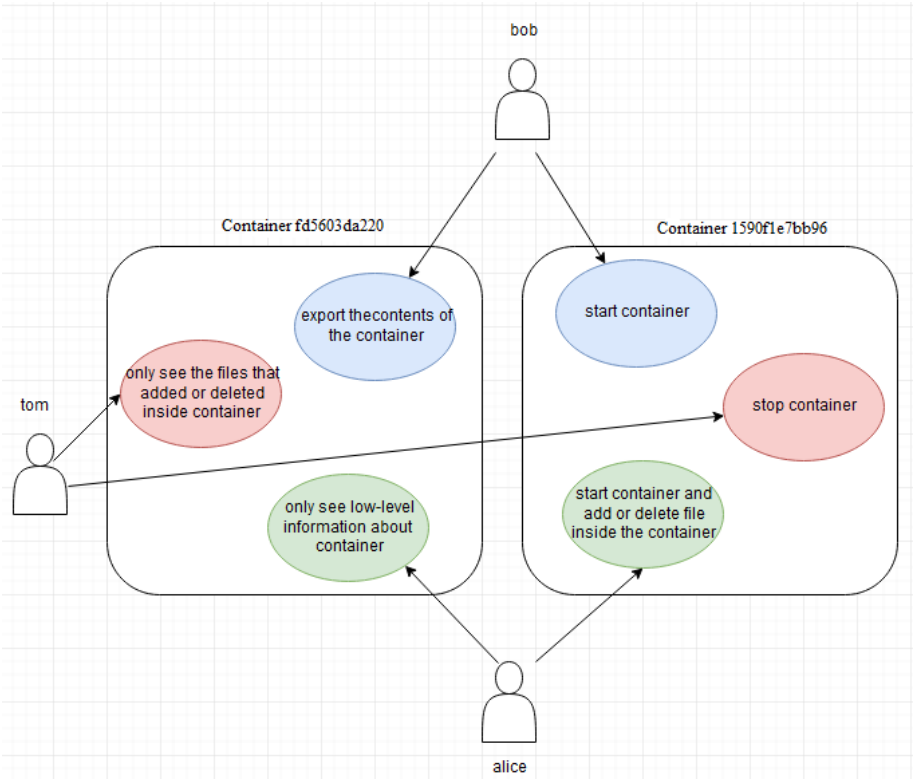

Figure 69 Trustnode02 user's permissions

C. Trustnode03:

Trustnode03 has two containers (3694024b1ea3 and 612e4f6965dd) and, in this Trustnode, only one user (Tom) can see the history of the images that help to create two containers in this node. Bob and Alice cannot perform any action in this Trustnode. Container 1 was created through an image that has ID ab6e4b0f51f8. Container 2 was created through an image that has ID 02e84d0157af. Therefore, Tom can see the history of these two images but perform no other action. Table 12 below shows the rules that are applied in this case for all users.

Table 12 the rules that applied in this case study for all users

	Container 1	Container 2
Bob	None	None
Tom	P,/V1.38/images/ ab6e4b0f51f8/ history, GET	P,/V1.38/images/02e84d0157af / history, GET
Alice	None	None

As described above, Bob and Alice cannot perform any action in Trustnode03 because they do not have permission. Tom can only see the history of the images that created two containers otherwise; tom got the error message from the authorisation plugin. Figure 70 below shows that the actions that could be performed by three users inside the containers in trustnode02.

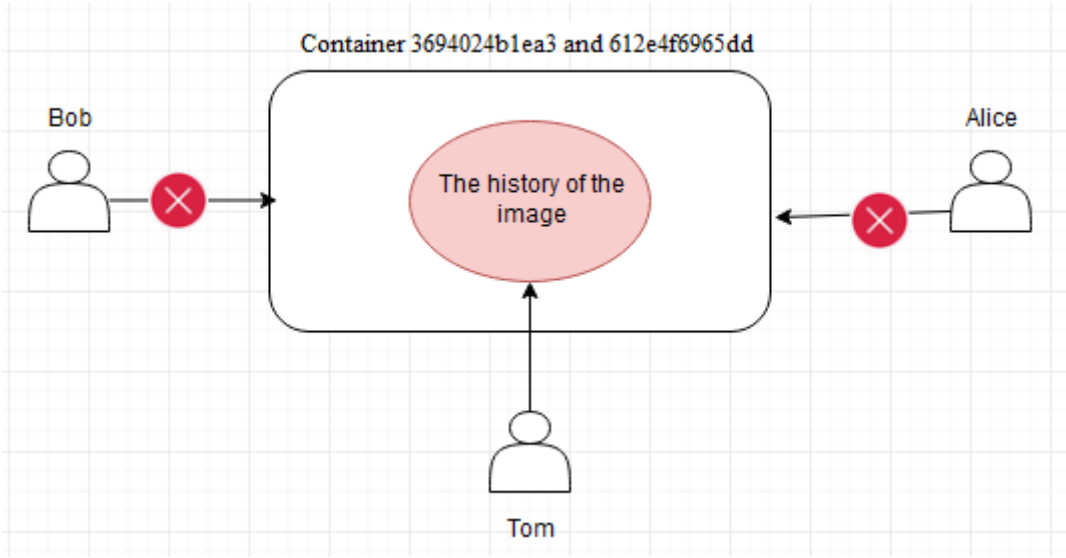

Figure 70 Trustnode03 user's permissions

6.5.2 TAIC model testing for users not added to the system.

Each user in the system should be registered in the TAIC script to get rules to access the containers. TAIC has a feature that will delete all rules in the policy file for any users not added to the system. Therefore, any user outside the system cannot get permission to perform anything inside Docker. The Authorisation plugin will deny the request provided by this user because the policy file is empty and does not have any rules that allow the user perform the specific task inside the containers.

For example, user Leo is created as an outside users that tries to access the containers inside the Trust nodes. In this case, the TAIC script will delete all rules in the policy file inside the authorisation plugin and so the authorisation plugin will deny all requests provided by Leo because he does not have any rules in the policy file. Therefore user Leo will receive an error message from the authorisation plugin in each node because the policy file will be empty. Figure 71 below shows the user Leo request result.

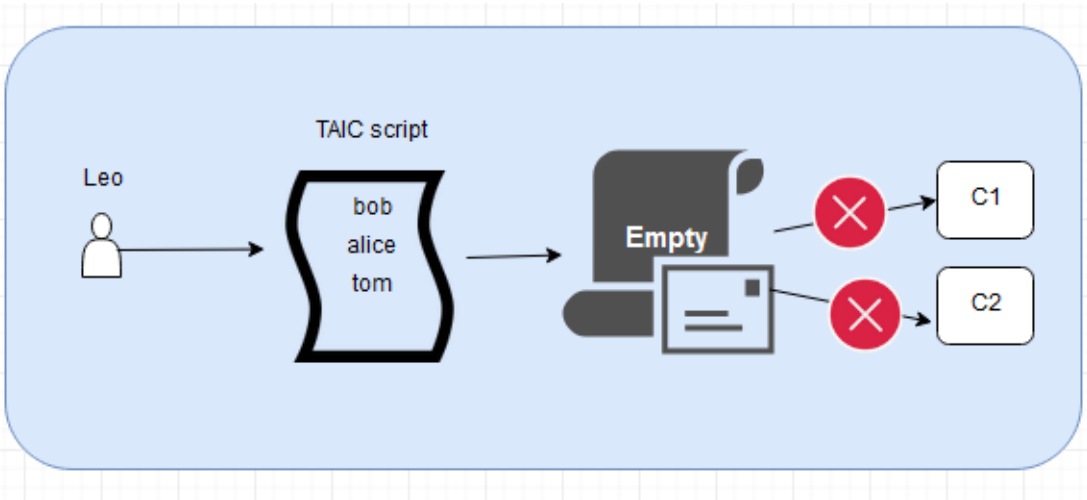

Figure 71 User Leo request result

From this test, this project has achieved the objective that ensures outside users cannot perform any actions in the containers in the system. Table 13 below shows the actual results for the TAIC model testing for the users who do not add to the system and the objectives that are achieved from this test.

Table 13 the actual results for the authorization plugin testing for user outside hosts,

User	Trustnode	Actual results
Leo	Trustnode01	Leo cannot perform any actions in the containers in the system and always got error message from the authorisation plugin in each node because the policy file is empty.
	Trustnode02	
	Trustnode03	

6.5.3 TAIC script testing:

Each authorised system user should have rules in the TAIC system. The TAIC script opens the policy file and updates the policy file according to the user who is logged in to the system. The TAIC script should operate as follows. First, identify who is logged in to the system. Second, open the policy file inside the authorisation plugin. Third, the TAIC script deletes all rules inside the policy file and provides rules for the logged in user. The administrator assigns rules for the users. Finally, TAIC should save the policy file.

For example, bob has rules such as P,/V1.38/containers/94903b6c7b9/ start, POST and P,/V1.38/containers/94903b6c7b9/ exec, POST. When Bob logs into the system, the TAIC script deletes all rules inside the policy file and add these two rules in the policy file to allow Bob to perform specific actions inside the container. Tom and Alice have the same scenario when they logged in to the system. For any user not added to the system, the TAIC script opens the policy file and delete all rules inside the policy file to make sure the user cannot perform any action in the system. Figure 72 below shows that TAIC script and policy file.

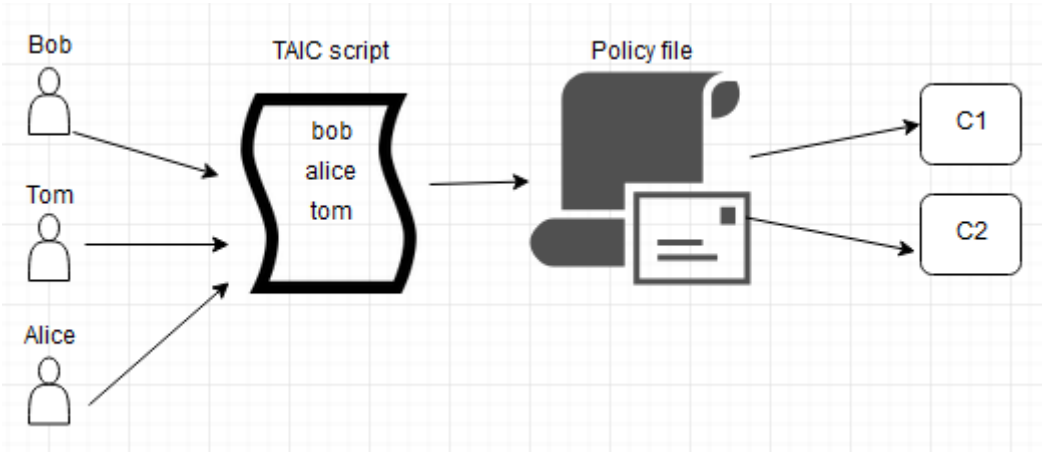

Figure 72 Figure below show that TAIC script and policy file.

6.5.4 TAICSSH script testing:

This script allows the group of users that have the same rules to access the host at the same time. For example, userA1, userA2, and userA3 are related to groupA that have the same rules in the policy file. Moreover, userB1, userB2, and userB3 are users related to groupB have the same rules as well. When userA1 logged in the host, TAICSSH accepts the remote connection from userA2 and userA3 only. Otherwise, it will reject all remote

requests. When userB1 access the host, the TAICSSH will accept userB2 and userB3 request only to remote connection. Figures 73 and 74 below shows how TAIC will operate for groupA and groupB.

.

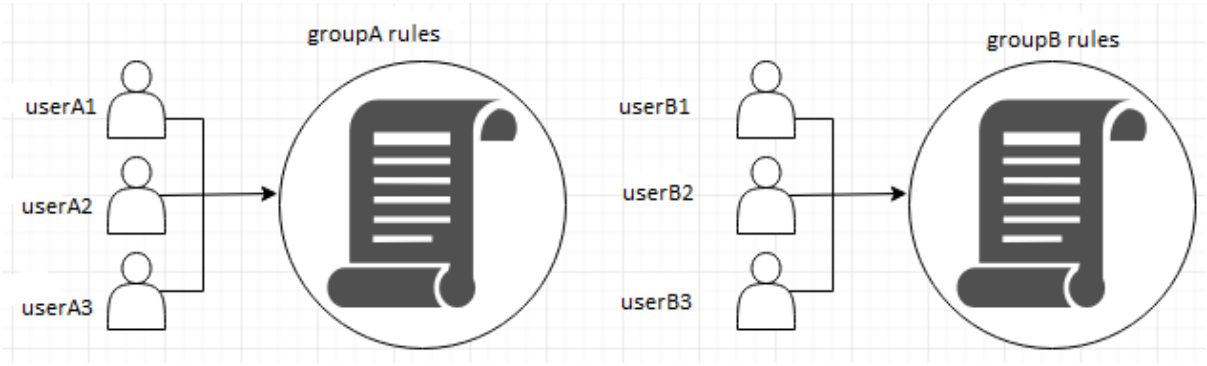

Figure 73 Bob and tom group.

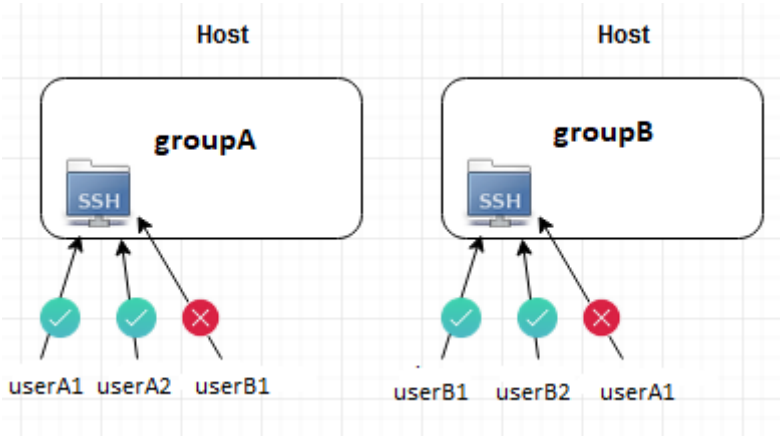

Figure 74 SSH configuration for user's groupA and groupB

6.5.5 Protect the policy file, TAIC, TAICSSH scripts:

In Docker, each user can get root privileges to access the Docker daemon. Therefore, the policy file, TAIC, and TAICSSH scripts could be accessed by any users in the system. This project has changed the owner and permission for the policy file, TAIC, and TAICSSH scripts to ensure only one user (the administrator) can access or modify these assets. In this system, the Pi user is created as the admin user. Therefore, Pi is the owner for the policy file, TAIC, and TAICSSH scripts and has full permissions (read, write, and execute):

```
sudo chown pi:pi  -R   /usr/lib/docker/examples/basic_policy.csv
sudo chgrp -R pi /usr/lib/docker/examples/basic_policy.csv
sudo chmod -R 700 /usr/lib/docker/examples/basic_policy.csv

sudo chown pi:pi  -R   /usr/lib/docker/examples/TAIC
sudo chgrp -R pi /usr/lib/docker/examples/TAIC
sudo chmod -R 700 /usr/lib/docker/examples/TAIC

sudo chown pi:pi  -R   /usr/lib/docker/examples/ TAICSSH
sudo chgrp -R pi /usr/lib/docker/examples/ TAICSSH
sudo chmod -R 700 /usr/lib/docker/examples/TAICSSH
```

6.5.6 Analysis

For the above tests, this section provides the TAIC model results for users inside and outside the host. As expected, each user can perform specific actions inside the containers in each host. The TAIC script identifies the users and update the policy file inside the authorisation plugin based on who is logged in. This authorisation plugin then provides limited access to the users' containers that could improve security in the system and maintain the user containers' privacy. Each user should have permissions located in the policy file to perform specific tasks inside the container. In case 3, remote access is limited to particular users only. Moreover, only the user Pi can access or edit the TAIC, TAICSSH files and policy file. Figure 75 below shows that the testing results for case 3 and the TAIC model.

Number	Testing	Result
Testing 1	Each user has different rules from others.	Yes
Testing 2	Each user could perform different actions inside the container from others.	Yes
Testing 3	User has permission should perform a specific action inside container (inside or outside the host)	Yes
Testing 4	user does not have permission should not perform a specific action inside container (inside or outside the host)	Yes
Testing 5	User is not added to the system should not access the container	Yes
Testing 6	User does not have remote access permission should not get a remote access to the host	Yes
Testing 7	User has remote access permission should get a remote access to the host	Yes
Testing 8	User should not edit or access access TAICSSH script, TAIC script or policy file (inside or outside the host)	Yes
Testing 9	Admin should can edit or access TAICSSH script, TAIC script or policy file.	Yes

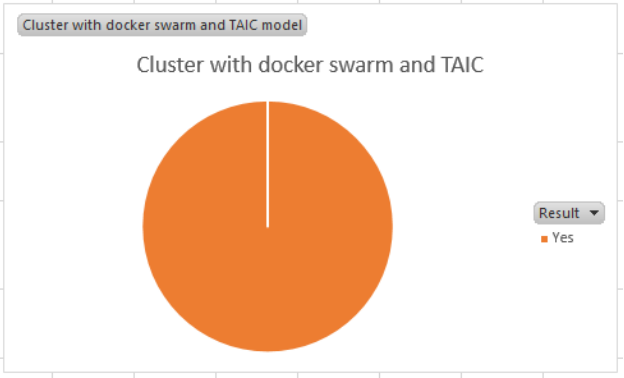

Figure 75 testing results for case three: Cluster with Docker swarm and TAIC model.

6.6 Analysis of Results

Table 14 below shows the expected behaviour for the three Cluster with Docker swarm and TAIC model

Table 14 Validate the expected users behavior

	Containers	Expected behaviour
1	Case 1 : Cluster with Docker Swarm only	In all trust nodes, all users can perform any action inside all containers in the host. Any user inside or outside the host can access any container in the system without any restriction.
2	Case 2 : Cluster with Docker Swarm and authorisation plugin	This authorisation plug-in provides limited access to the users' containers. In all Trust nodes, all users should have the same rules and make the same actions inside the containers. However, this case does not make each user can perform specific action inside the container and cannot identify the user and make each user has different rules from others.
3	Case 3 : Cluster with Docker Swarm and TAIC model	This authorisation plug-in provides limited access to the users' containers. In all Trustnode, each user can perform specific tasks inside the container. The user could have the same rules or different rules from others. TAIC model identifies the users and makes each user has different or the same rules from others.

In the first case, any user inside or outside the host can access any container in the system. Moreover, any user can get remote access to the host and can perform any action on the container without restriction. Docker Swarm does not offer isolation of users within a container and each user can perform specific actions inside the containers. In case 2, the users can only perform specific actions inside the containers because the authorisation plugin provides limited access to the containers. However, all users have the rules and making each user have different rules is not supported because the Docker daemon does not identify the user.

The last case is proposed as the TAIC model and is based on three main elements which are the TAIC script, the TAICSSH script, and an authorisation plugin. In particular, the TAIC script is created to identify the users because the socket API does not support users in the current version of Docker. As such, this project has developed and evaluated a proof of concept implementation to provide container isolation. The authorisation plugin in the TAIC model provides a policy file to protect containers by specifying rules that allow the user to perform specific tasks inside particular containers. The authorisation plugin helps the Docker daemon to provide secure access to the Docker containers. Therefore, the malicious user could not pass through the Docker containers. The TAIC model also leverages the TAICSSH script to improve privacy in the user containers. The TAICSSH script is for limiting remote connections to the host and allow particular users to get permission for the remote connection.

Each host in the cloud data centre has a Docker authorisation plugin, which is populated with policies based on the container images deployed. These points are also less prone to errors because each user can access his own data only, but he cannot access other users' data. The two scripts and authorisation plugins are evidence shown by the provider to the cloud user to prove that the environment is secure and user containers are protected. This design helps to improve security in the containers by limiting the system access to authorised users. As a result, a trust relationship between the cloud provider and users is achieved through TAIC, which ensures the customer can perform specific tasks inside the containers that have been added to the policy file in the authorisation plugin and carry out particular tasks on the container such as start, attach and stop.

The overall shape of the TAIC model is defining a list of rules, which has permission to limit system access for malicious users to protect a container from other guests. Initially, the user wants to access the particular resource on the containers. The user has one or more rules, which have permissions. The Docker daemon obtains the user request, processes it, and sends it to the authorisation plugin. Then the authorisation plugin determines whether the user request is accepted or denied according to the user rules. When the authorisation plugin generates the request result, the system will return the results to the Docker daemon to grant or deny the request. If the user has permission to access the requested container, then the user can perform specific tasks on the container according to the permission he has. Otherwise, the system will deny the user request.

However, TAIC has disadvantages such as the system does not protect the Docker daemon that has an authorisation plugin. Therefore, the attacker could attack the Docker daemon and get access to all containers on the system. Therefore, a trustworthy administrator should control the Docker daemon. If the Docker daemon is attacked by malicious guests, the attacker can access and start any containers on the system. Moreover, each authorisation plugin should have a socket and service file. If one of these files is attacked, this means that the authorisation plugin will not provide the policy for the containers. Therefore, protecting the Docker daemon is essential to make the system in this project more efficient and will be investigated as part of future work.

6.7 Summary

This chapter provides the result for three cases of security testing. The first case is cluster with Docker swarm only; in this case, the users can perform any action inside any containers. In case 2 that has an authorisation plugin, all users can get the same rules to perform any action inside the container. However, the Docker daemon does not identify the users; therefore, the second case cannot make each user has different rules from other users. In case three, which is cluster with Docker swarm and TAIC model, each user can perform specific tasks inside the container according to his rules that are added to the policy file. Therefore, the TAIC script identifies the users and update the policy file in the authorisation plugin. TAICSSH script is a limit remote connection request for authorised users only. As well as, the authorisation plugin improved the security by adding the policy to particular containers.

This evaluation has multiple hosts, and each host has a TAIC model. This chapter evaluates the TAIC for users inside the host and outside the host. Moreover, evaluate the two scripts is provided in this chapter. Then, this chapter explains how to add a new container, image users, and node in section automated Deployment. Finally, the overall shape of the TAIC model provided in the last section of this chapter.

7. CONCLUSION AND FUTURE WORK

My contributions and summary of the work are provided in this chapter. The conclusion chapter discusses the models that are implemented and the results for each model. This chapter describes future work that could be developed in this area. This project has focussed on the concept of trust and presented how this is tightly linked to the provision of isolation in Cloud Computing. Then explored how different levels of isolation can be offered depending on the type of virtualisation being used and identified container-based virtualisation as a potential problem. The provider services should protect the cloud environment and make it secure to build a trust relationship with the cloud users.

7.1 Isolation in Container Based Virtualisation

Isolation is an important issue in cloud computing. Building trust in cloud computing that using container virtualisation need to solve security vulnerabilities that include isolation. Therefore, address the isolation issue could help to build trust relationships between the cloud service provider and users.

As a result of our literature review and subsequent analysis, this project found that container-based virtualisation does not provide a high level of isolation because it shares aspects of the OS kernel and components between all instances. This means that a user instance (and their data) is never truly secure and trustable, and bugs, malware, or other intrusions could carry over from one container to the other. The performance isolation of various virtualisation systems was quantified based on existing work, and extended this to include current lightweight virtualisation techniques using Docker.

Since this level isolation is, to some extent, intrinsic to the approach, the issue of isolation in container virtualisation could instead be addressed by creating a trust model that used two scripts and authorisation plugin. This project attempts to enforce isolation within each instance, as an individual user can only perform a specific task in a container according to rules, authority, and responsibility assigned to that cloud user. The user can access particular containers through the authorisation plugin that should follow the API rule that does not allow the user to modify the image. As well as the privacy and isolation of customers will precede any other need for building trust in cloud computing systems. In that regard, every process that regards clients will ensure the achievement of their needs, and that will include data privacy, integrity, and availability. The customer will be required to send a request to access the security system before they get access to any resource. After that the security system can check the user's rules and permissions to ensure the access is acceptable. The security system can also establish a stronger trust relationship that benefits the customers when they use cloud computing.

7.2 Objectives and Contributions

This study mainly focuses on researching and identifying solutions that would help to create a trust-based relationship between cloud service providers and cloud users. Containers are the perfect way to isolate the application environment or isolate resources — container based on the use of a single OS that is kernel OS that could help to reduce the cost. However, the first object of this project is to compare isolation in container-based virtualisation and traditional VMs and conduct experiments to identify the differences between isolation in container-based and hypervisor-based virtualisation technologies. The hypervisor provides better isolation than the container. Container-based virtualisation needs to improve the security to use it in cloud computing. The

project presented a security system to provide better isolation for containers from malicious guests to boost the security of data that is stored in them. This will give confidence in the confidentiality, integrity and availability of the data stored by the user. In this project, aimed to solve the isolation issue in container-based virtualisation to build trust relationships between cloud service providers and users.

Evaluate the approach that could help to address the isolation issue in container-based virtualisation and critically analyse the positive and negative aspects of the approach is one objective of this project. This project has suggested an approach in the design chapter and provides the positive and negative aspects of this approach. This approach has a script that is created to identify the users and make each user can perform different action inside the container from other users.

Design an architecture that has security characteristics that could help to address the isolation issue in the container and implement a prototype based on the resultant architectural design are two objectives of this project. The TAIC model has two scripts and authorisation plugin. This model provides isolation for the user containers and the authorisation plugin make the users can perform a particular task on the containers. This model could help to improve security in the container-based virtualisation. According to the previous studies, using containers in the cloud is not a good idea because container has weak isolation. Therefore, this model provides isolate for the user containers and give the policy for the containers as well. This could help to build the trust relationship between the cloud offers and cloud users when the provider using container-based virtualisation.

The last objective for this project is to evaluate the prototype using a Raspberry Pi-based datacentre testbed. The Implementation chapter describes how the system was built using scripts and the authorisation plugin. Also, use this chapter to describe our realistic development and test environment, which uses a cluster of Raspberry Pis to simulate a cloud data centre. The final section is evaluation and testing which evaluates the Trust Architecture for Isolation in Containers (TAIC) results.

The TAIC and TAICSSH script in Trust Architecture for Isolation in Containers (TAIC) provides could help to build the trust relationship between cloud providers and users because they help to make each user can perform specific tasks inside the containers. Secondly, the authorisation plugin protects the container from other users by adding rules to the policy file, which makes the environment more secure. Finally, The Picloud with TAIC mode provides a secure environment that would help to create a trust relationship between the cloud service provider and users.

The novelty in this project addresses the isolation issue within the containers. However, the client can communicate with the container through the Docker daemon that listens to the Docker socket. The Docker socket in the current version does not identify the user. Chapter 5 provides the TAIC model that could help to improve the security in cloud computing that using container virtualisation. This model has TAIC scripts that identify the users and update the policy file inside the authorisation plugin. TAICSSH script that limits remote access connection for authorised users only. Finally, the authorisation plugin that protects the Docker container and makes the user perform a specific task within the container according to the policy file.

7.3 Concluding Remarks

This thesis has presented three approaches that would help to build a trust relationship between cloud providers and users in the container-based cloud computing. Container virtualisation is not as secure as traditional hypervisor-based virtual machines due to the issue of weaker isolation. However, TAIC could improve security

in the container and would help to improve the level of trust in this form of cloud computing because the user can guarantee that their containers are not visible to others and can only be accessed by authenticated users.

TAIC provides a layer of isolation for user containers. This layer of isolation could improve security in the container and make the user data more private. The developer could increase the security in particular containers by using the authorisation plugin. This feature allows the user only to perform specific tasks in the container. Therefore, the cloud provider could demonstrate to the cloud users show that the environment is secure, and the user's data is protected. This thesis has shown how this functionality could be implemented in currently deployed instances of Docker-based cloud data centres, and also demonstrated how it could be built more efficiently in the future with some modifications to the main API.

7.4 Further Work

One major limitation encountered during our implementation is that the current version of Docker provides a socket API that does not identify the users. Isolation of users within the containers will be provided by the authorisation plugin only when Docker updates the current version and enables this feature. In the last version of Docker, Docker provide an experimental rootless mode that is make the user access the Docker daemon and containers without root privileges. The benefit to this is, even if it is compromised the attacker will not be able to gain root access to the host. However, rootless mode completely isolate the users from each other. Because each user has its own space to run, the containers and the users cannot shred the containers. Moreover, rootless mode tested on Ubuntu 18.04 only. Therefore, the users cannot share the same containers. The user can access particular resources within the containers not all resources. Each user should have permission to access specific resources within the container according to the policy file in the authorisation plugin. Therefore, one aim of our immediate future works will be to address this by extending the API to include user details.

Next, our future work will make the authorisation plugin isolate users within the containers by improving the Docker socket. The future Trust Architecture for Isolation in Containers (TAIC) model will contains the authorisation plugin only. The authorisation plugin will create to allow users to perform specific tasks inside the container. Isolating users within the containers will address by using an authorisation plugin only, which has the second model to grant or deny the user access to the resources. Go language code will be written in the Docker daemon to create an authorisation plugin, where RBAC is used as access control to approve or deny users' requests. Then create an authorisation plugin by using go language programming code and install it within the Docker daemon in the host. After installing the Docker daemon in the host, the developer should update the Docker daemon to enable the authorisation plugin by using dockerd command.

7.4.1 Isolate users within the containers and protecting particular resources

Different methods will be used in future work such as adding role-based access control to users within the Docker container that are needed for in-depth analysis and improve the Docker socket. The user can access his resources only, and any users in the system could not access some resources. It could use more than one authorisation plugin in the system, one authorisation plugin has a role for users, and the other authorisation plugin has a role for the user resources.

The Docker daemon could have a particular configuration for Cgroup features that could help to improve container security. The user cannot access overloaded containers unless the issue solved by configuring cgroup in

the Docker daemon. In the future work, Cgroup and namespace authentication could be configured by the script in the future work. The script should be run automatically with any accessed containers.

The Picloud could be made more realistic by using different models and increasing number of devices. In this project, this project has built a prototype that consists of four nodes of Raspberry pi devices model B. One node as a master and three nodes as trust nodes. In future work, the number of devices will be increased to build a use case of the real word datacentre.

Another critical aspect of future work is to investigate developing a subsystem to protect the Docker daemon because the authentication namespace and authorisation plugin is part of the Docker daemon. Therefore, if the Docker daemon is attacked, that means the environment will not be secure, and user data will not be protected and private.

8. REFERENCES

[1] I. Ion, N. Sachdeva, P. Kumaraguru, Srdjan, #268, and apkun, "Home is safer than the cloud!: privacy concerns for consumer cloud storage," presented at the Proceedings of the Seventh Symposium on Usable Privacy and Security, Pittsburgh, Pennsylvania, 2011.

[2] S. Pearson, "Privacy, security and trust in cloud computing," in *Privacy and security for cloud computing*, ed: Springer, 2013, pp. 3-42.

[3] K. M. Khan and Q. Malluhi, "Establishing trust in cloud computing," *IT professional,* vol. 12, pp. 20-27, 2010.

[4] Y. Wang, J. Wen, W. Zhou, and F. Luo, "A Novel Dynamic Cloud Service Trust Evaluation Model in Cloud Computing," in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, 2018, pp. 10-15.

[5] K. Hwang and D. Li, "Trusted cloud computing with secure resources and data coloring," *IEEE Internet Computing,* pp. 14-22, 2010.

[6] Z. Shen and Q. Tong, "The security of cloud computing system enabled by trusted computing technology," in *2010 2nd International Conference on Signal Processing Systems*, 2010, pp. V2-11-V2-15.

[7] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, *et al.*, "Quantifying the performance isolation properties of virtualization systems," in *Proceedings of the 2007 workshop on Experimental computer science*, 2007, p. 6.

[8] F. P. Tso, D. R. White, S. Jouet, J. Singer, and D. P. Pezaros, "The glasgow raspberry pi cloud: A scale model for cloud computing infrastructures," in *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops*, 2013, pp. 108-112.

[9] M. D. Ryan, "Cloud computing privacy concerns on our doorstep," *Communications of the ACM,* vol. 54, pp. 36-38, 2011.

[10] G. Garrison, S. Kim, and R. L. Wakefield, "Success factors for deploying cloud computing," *Communications of the ACM,* vol. 55, pp. 62-68, 2012.

[11] C. Clavister, "Security in the cloud," *White paper,* 2008.

[12] H. Li and Q. Wu, "A distributed intrusion detection model based on cloud theory," in *2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems*, 2012, pp. 435-439.

[13] P. Mell and T. Grance, "The NIST definition of cloud computing," 2011.

[14] T. Mather, S. Kumaraswamy, and S. Latif, *Cloud security and privacy: an enterprise perspective on risks and compliance*: " O'Reilly Media, Inc.", 2009.

[15] M. Alouane and H. El Bakkali, "Security, privacy and trust in cloud computing: A comparative study," in *2015 International Conference on Cloud Technologies and Applications (CloudTech)*, 2015, pp. 1-8.

[16] S. Subashini and V. Kavitha, "A survey on security issues in service delivery models of cloud computing," *Journal of network and computer applications,* vol. 34, pp. 1-11, 2011.

[17] D. Cearley and D. Reeves, "Cloud computing innovation key initiative overview," *Gartner Inc,* vol. 22, p. 2011, 2011.

[18] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. Lightweight Virtualization: A Performance Comparison," in *2015 IEEE International Conference on Cloud Engineering*, 2015, pp. 386-393.

[19] A. S. Horvath and R. Agrawal, "Trust in cloud computing," in *SoutheastCon 2015*, 2015, pp. 1-8.

[20] J. Bringer, B. Gallego, G. Karame, M. Kohler, P. Louridas, M. Önen, *et al.*, *TREDISEC: Trust-Aware REliable and Distributed Information SEcurity in the Cloud*, 2015.

[21] E. Chang, T. Dillon, and D. Calder, "Human system interaction with confident computing. The mega trend," in *2008 Conference on Human System Interactions*, 2008, pp. 1-11.

[22] M. Saudi, E. Mohd Tamil, M. Idris, K. Seman, and L. Mohd Nasir, *Defending worms attack through EDOWA system*, 2008.

[23] S. Pearson and A. Benameur, "Privacy, Security and Trust Issues Arising from Cloud Computing," in *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, 2010, pp. 693-702.

[24] M. Tang, X. Dai, J. Liu, and J. Chen, "Towards a trust evaluation middleware for cloud service selection," *Future Generation Computer Systems,* vol. 74, pp. 302-312, 2017.

[25] M. Chiregi and N. J. Navimipour, "A new method for trust and reputation evaluation in the cloud environments using the recommendations of opinion leaders' entities and removing the effect of troll entities," *Computers in Human Behavior,* vol. 60, pp. 280-292, 2016.

[26] E. D. Canedo, R. de Sousa Junior, R. Rafhael de Carvalho, and R. Albuquerque, *Trust Measurements Yeld Distributed Decision Support in Cloud Computing* vol. 1, 2012.

[27] H. Takabi, J. B. D. Joshi, and G. Ahn, "Security and Privacy Challenges in Cloud Computing Environments," *IEEE Security & Privacy*, vol. 8, pp. 24-31, 2010.
[28] K. Z. Bijon, R. Krishnan, and R. Sandhu, "A Formal Model for Isolation Management in Cloud Infrastructure-as-a-Service," Cham, 2014, pp. 41-53.
[29] E. Caron and J. R. Cornabas, "Improving Users' Isolation in IaaS: Virtual Machine Placement with Security Constraints," in *2014 IEEE 7th International Conference on Cloud Computing*, 2014, pp. 64-71.
[30] F. Hao, T. V. Lakshman, S. Mukherjee, and H. Song, "Secure cloud computing with a virtualized network infrastructure," presented at the Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, Boston, MA, 2010.
[31] V. Ashktorab and S. R. Taghizadeh, "Security threats and countermeasures in cloud computing," *International Journal of Application or Innovation in Engineering & Management (IJAIEM)*, vol. 1, pp. 234-245, 2012.
[32] A. Singh and D. M. Shrivastava, "Overview of attacks on cloud computing," *International Journal of Engineering and Innovative Technology (IJEIT)*, vol. 1, 2012.
[33] F. Rocha and M. Correia, "Lucy in the sky without diamonds: Stealing confidential data in the cloud," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2011, pp. 129-134.
[34] S. A. Almulla and Y. Chan Yeob, "Cloud computing security management," in *2010 Second International Conference on Engineering System Management and Applications*, 2010, pp. 1-7.
[35] P. Sen, P. Saha, and S. Khatua, "A distributed approach towards trusted cloud computing platform," in *2015 Applications and Innovations in Mobile Computing (AIMoC)*, 2015, pp. 146-151.
[36] Y. Liu, L. Tan, and Q. Yi, "A trusted network platform architecture scheme on clouding computing model," in *2012 International Conference on Computer Science and Information Processing (CSIP)*, 2012, pp. 890-892.
[37] F. Hock and P. Kortiš, *Commercial and open-source based Intrusion Detection System and Intrusion Prevention System (IDS/IPS) design for an IP networks*, 2015.
[38] J. Cropper, J. Ullrich, P. Frühwirt, and E. Weippl, "The Role and Security of Firewalls in IaaS Cloud Computing," in *2015 10th International Conference on Availability, Reliability and Security*, 2015, pp. 70-79.
[39] H. Mohamed, L. Adil, T. Saida, and M. Hicham, "A collaborative intrusion detection and Prevention System in Cloud Computing," in *2013 Africon*, 2013, pp. 1-5.
[40] N. Santos, K. P. Gummadi, and R. Rodrigues, *Towards Trusted Cloud Computing*, 2009.
[41] S. Cirani, M. Picone, P. Gonizzi, L. Veltri, and G. Ferrari, *IoT-OAS: An OAuth-Based Authorization Service Architecture for Secure Services in IoT Scenarios* vol. 15, 2015.
[42] A. Mansour, M. Sadik, and E. Sabir, "Multi-factor authentication based on multimodal biometrics (MFA-MB) for Cloud Computing," in *2015 IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA)*, 2015, pp. 1-4.
[43] M. H. Dodani, *The Silver Lining of Cloud Computing* vol. 8, 2009.
[44] Y.-C. Liu, Y.-T. Ma, H.-S. Zhang, D.-Y. Li, and G.-S. Chen, "A method for trust management in cloud computing: Data coloring by cloud watermarking," *International Journal of Automation and Computing*, vol. 8, p. 280, 2011/08/12 2011.
[45] H. Shen and G. Liu, "An Efficient and Trustworthy Resource Sharing Platform for Collaborative Cloud Computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, pp. 862-875, 2014.
[46] S. Kamil and N. Thomas, *Performance analysis of the Trusted Cloud Computing Platform*, 2015.
[47] D. Gonzales, J. M. Kaplan, E. Saltzman, Z. Winkelman, and D. Woods, "Cloud-Trust—a Security Assessment Model for Infrastructure as a Service (IaaS) Clouds," *IEEE Transactions on Cloud Computing*, vol. 5, pp. 523-536, 2017.
[48] S. M. Habib, S. Ries, and M. Muhlhauser, "Towards a Trust Management System for Cloud Computing," in *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, 2011, pp. 933-939.
[49] S. Bharti, *A Survey on Auditability for Data Storage on Cloud Computing*, 2015.
[50] P. Sirohi and A. Agarwal, "Cloud computing data storage security framework relating to data integrity, privacy and trust," in *2015 1st International Conference on Next Generation Computing Technologies (NGCT)*, 2015, pp. 115-118.
[51] M. P. Amith Raj, A. Kumar, S. J. Pai, and A. Gopal, "Enhancing security of Docker using Linux hardening techniques," in *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, 2016, pp. 94-99.

[52] B. Kelley, J. J. Prevost, P. Rad, and A. Fatima, "Securing Cloud Containers Using Quantum Networking Channels," in *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, 2016, pp. 103-111.
[53] A. M. Joy, "Performance comparison between Linux containers and virtual machines," in *2015 International Conference on Advances in Computer Engineering and Applications*, 2015, pp. 342-346.
[54] M. Song, C. Zhang, and E. Haihong, "An Auto Scaling System for API Gateway Based on Kubernetes," in *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*, 2018, pp. 109-112.
[55] J. Shah and D. Dubaria, "Building Modern Clouds: Using Docker, Kubernetes & Google Cloud Platform," in *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, 2019, pp. 0184-0189.
[56] Twistlock. (2017). *Guide to Docker Swarm Security*. Available: https://www.twistlock.com/2017/10/11/guide-swarm-security/
[57] IBM. (2019). *Kubernetes*. Available: https://www.ibm.com/cloud/learn/kubernetes
[58] t. NewStack. (2018). *Kubernetes vs. Docker Swarm: What's the Difference?* Available: https://thenewstack.io/kubernetes-vs-docker-swarm-whats-the-difference/
[59] Kubernetes. (2018). *Kubernetes Reference*. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet-authentication-authorization/
[60] Kubernetes, "Kubernetes Refrences," 2019.
[61] A. Modak, S. D. Chaudhary, P. S. Paygude, and S. R. Ldate, "Techniques to Secure Data on Cloud: Docker Swarm or Kubernetes?," in *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*, 2018, pp. 7-12.
[62] d. docs, "Access authorization plugin," 2019.
[63] Hackernoon. (2018). *Kubernetes vs. Docker Swarm: A Complete Comparison Guide*. Available: https://hackernoon.com/kubernetes-vs-docker-swarm-a-complete-comparison-guide-15ba3ac6f750
[64] T. Archer. (2018). *Getting Started with Containers: Azure Container Services Explained*. Available: https://www.swc.com/blog/cloud/getting-started-with-containers-azure-container-services-explained
[65] M. Azure. (2019). *Securing Docker containers in Azure Container Service*. Available: https://docs.microsoft.com/en-us/azure/container-service/dcos-swarm/container-service-security
[66] G. Cloud. (2019). *Google Kubernetes Engine documentation* Available: https://cloud.google.com/kubernetes-engine/docs/#pricing
[67] aws. (2019). *Amazon EC2*. Available: https://aws.amazon.com/ec2/
[68] S. L. S. a. Wayne). *Demystifying Cloud Foundry's Diego*. Available: https://www.starkandwayne.com/blog/demystifying-cloud-foundrys-diego/
[69] W. Marques, P. Souza, F. Rossi, G. Rodrigues, R. N. Calheiros, M. da Silva, *et al.*, *Evaluating container-based virtualization overhead on the general-purpose IoT platform*, 2018.
[70] G. Obasuyi and A. Sari, *Security Challenges of Virtualization Hypervisors in Virtualized Hardware Environment* vol. 8, 2015.
[71] A. Younge, "Architectural Principles and Experimentation of Distributed High Performance Virtual Clusters," 2016.
[72] A. Desai, R. Oza, P. Sharma, and B. Patel, "Hypervisor: A survey on concepts and taxonomy," *International Journal of Innovative Technology and Exploring Engineering*, vol. 2, pp. 222-225, 2013.
[73] VMWare, *Understanding Full Virtualization, Paravirtualization, and Hardware Assist*, 2008
[74] H. Van, F. Tran, and J.-M. Menaud, "Autonomic Virtual Resource Management for Service Hosting Platforms," *Proceedings of the Workshop on Software Engineering Challenges in Cloud Computing*, 05/23 2009.
[75] Z. Wu, Z. Xu, and H. Wang, *Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud*, 2012.
[76] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, *et al.*, "Xen and the art of virtualization," presented at the Proceedings of the nineteenth ACM symposium on Operating systems principles, Bolton Landing, NY, USA, 2003.
[77] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift, "Resource-freeing attacks: improve your cloud performance (at your neighbor's expense)," presented at the Proceedings of the 2012 ACM conference on Computer and communications security, Raleigh, North Carolina, USA, 2012.
[78] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, "HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis," in *2011 IEEE Symposium on Security and Privacy*, 2011, pp. 313-328.
[79] T. Kim, M. Peinado, and G. Mainar-Ruiz, *StealthMem: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud*, 2012.

[80] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. D. Rose, "Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments," presented at the Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2013.
[81] A. Bettini, "Vulnerability exploitation in Docker container environments," *FlawCheck, Black Hat Europe*, 2015.
[82] R. Yasrab, "Mitigating docker security issues," *arXiv preprint arXiv:1804.05039*, 2018.
[83] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*, 2015, pp. 171-172.
[84] S. Soltesz, H. P, #246, tzl, M. E. Fiuczynski, A. Bavier, *et al.*, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," presented at the Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, Lisbon, Portugal, 2007.
[85] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, pp. 81-84, 2014.
[86] M. G. Xavier, I. C. D. Oliveira, F. D. Rossi, R. D. D. Passos, K. J. Matteussi, and C. A. F. D. Rose, "A Performance Isolation Analysis of Disk-Intensive Workloads on Container-Based Clouds," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2015, pp. 253-260.
[87] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs Containerization to Support PaaS," in *2014 IEEE International Conference on Cloud Engineering*, 2014, pp. 610-614.
[88] T. Combe, A. Martin, and R. Di Pietro, "To Docker or Not to Docker: A Security Perspective," *IEEE Cloud Computing*, vol. 3, pp. 54-62, 2016.
[89] Twistlock. (2019 GitHub). *twistlock/authz*. Available: https://github.com/twistlock/authz
[90] Casbin. (2019). *casbin/casbin-authz-plugin*. Available: https://github.com/casbin/casbin-authz-plugin
[91] M. Shah and W. Anwaar, *Energy Efficient Computing: A Comparison of Raspberry PI with Modern Devices* vol. 4, 2015.
[92] X. Wei, H. Li, and D. Li, "MPICH-G-DM: An Enhanced MPICH-G with Supporting Dynamic Job Migration," in *2009 Fourth ChinaGrid Annual Conference*, 2009, pp. 67-76.
[93] V. software. (2008). *An overview of the Secure Shell (SSH)*. Available: https://www.vandyke.com/solutions/ssh_overview/ssh_overview.pdf
[94] d. docs. (2019 Docker Inc). *Docker Plugin API*. Available: https://docs.docker.com/engine/extend/plugin_api/
[95] D. docs. (2019). *Access authorization plugin*. Available: https://docs.docker.com/engine/extend/plugins_authorization/