Western Graduate&PostdoctoralStudies

Western University
Scholarship@Western

Electronic Thesis and Dissertation Repository

April 2015

Trust based Privacy Policy Enforcement in Cloud Computing

Karthick Ramachandran
The University of Western Ontario

Supervisor
Dr. Hanan Lutfiyya
The University of Western Ontario

Joint Supervisor
Mark Perry
The University of Western Ontario

Graduate Program in Computer Science

A thesis submitted in partial fulfillment of the requirements for the degree in Doctor of Philosophy

© Karthick Ramachandran 2015

Follow this and additional works at: https://ir.lib.uwo.ca/etd

Part of the Information Security Commons, OS and Networks Commons, and the Systems Architecture Commons

Recommended Citation
Ramachandran, Karthick, "Trust based Privacy Policy Enforcement in Cloud Computing" (2015). *Electronic Thesis and Dissertation Repository*. 2728.
https://ir.lib.uwo.ca/etd/2728

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact tadam@uwo.ca.

TRUST BASED PRIVACY POLICY ENFORCEMENT IN CLOUD COMPUTING

(Thesis format: Monograph)

by

Karthick Ramachandran

Graduate Program in Department of Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

© Karthick Ramachandran 2015

Abstract

Cloud computing offers opportunities for organizations to reduce IT costs by using the computation and storage of a remote provider. Despite the benefits offered by cloud computing paradigm, organizations are still wary of delegating their computation and storage to a cloud service provider due to trust concerns. The trust issues with the cloud can be addressed by a combination of regulatory frameworks and supporting technologies. Privacy Enhancing Technologies (PET) and remote attestation provide the technologies for addressing the trust concerns. PET provides proactive measures through cryptography and selective dissemination of data to the client. Remote attestation mechanisms provides reactive measures by enabling the client to remotely verify if a provider is compromised. The contributions of this work are three fold. This thesis explores the PET landscape by studying in detail the implications of using PET in cloud architectures. The practicality of remote attestation in Software as a Service (SaaS) and Infrastructure as a Service (IaaS) scenarios is also analyzed and improvements have been proposed to the state of the art. This thesis also propose a fresh look at trust relationships in cloud computing, where a single provider changes its configuration for each client based on the subjective and dynamic trust assessments of clients. We conclude by proposing a plan for expanding on the completed work.

Keywords: Trust, Privacy, Security, Cloud Computing, Subjective and Dynamic Trust, Remote Attestation, Privacy Enhancing Technologies

for my parents

Acknowledgments

I am deeply indebted to my supervisors Dr. Hanan Lutfiyya and Prof. Mark Perry for giving me an opportunity to work on this field. Dr. Lutfiyya is one of the most patient and humble people I have ever met. This thesis would not have been possible without her continued support, patient guidance and encouragement. Prof. Perry was always a Skype call away for any discussion. His support is deeply appreciated.

I would like to thank my colleagues at DiGS (Distributed and Grid Systems) research group for fruitful discussions and their help and support for the past 5 years. Thanks to Sharmila Mohan for collaborating with me in a project for trusted healthcare database.

Special thanks to the Computer Science Department and its amazing staff: Janice, Cheryl, Dianne and Angie.

To University of Western Ontario for providing such a beautiful and healthy environment to perform research.

I would also like to acknowledge the Ontario Government for granting scholarship that supported the final two years.

Ph.D is a long and sometimes a very lonely journey. I would not have made it, if it is not for my near and dear.

To my friends here in Canada and around the world (in alphabetical order): Achuthan, Aditi, Anu, Deepak, Dinagar, Gaston, Hareesh, Harpreet, Jananee, Lankesh, Manasi, Mike, Nirupama, Prakruti, Rachita, Roopa, Sharmila, Shivangi, Siva, Siya, Sneha, Sriram, Viji, Vijay and the countless others. Thank you so much for making my travel adventurous.

To my roommates Amit, Esperanza and Javier for accommodating me under the same roof and being the family in Canada.

To my sister's folks for their constant motivation and strength.

Finally this thesis is for my parents and my sister Ramya: for always placing my happiness before yours.

Contents

Abstract . . . ii
Acknowledgments . . . iv
Table of Contents . . . v
List of Figures . . . xi
List of Tables . . . xiii
List of Acronyms . . . xiv

1 Introduction **1**
1.1 Privacy Enhancing Technologies . . . 2
1.2 Need for trust: Remote Attestation . . . 3
1.3 Trust Relationships . . . 5
1.3.1 The Plumber Problem . . . 5
1.3.2 Trust Relationships in Cloud . . . 8
1.4 Contributions of Thesis . . . 10
1.5 Scope . . . 12
1.6 Thesis Organization . . . 12

2 Background **15**
2.1 Cloud Computing . . . 15
2.1.1 Service Models . . . 16
2.1.2 Deployment models . . . 17
2.1.3 Characteristics . . . 17
2.2 Privacy, Security and Trust . . . 18
2.2.1 Privacy . . . 18
2.2.2 Security . . . 21

2.2.3 Trust 22
2.2.4 Risk Vs Trust 24
2.3 Threat Model 25
2.4 Privacy Enhancing Technologies (PET) 26
2.5 Remote Attestation 28
2.5.1 Trusted Computing 28
2.5.2 Trusted Boot 29
2.5.3 Trusted Platform Module (TPM) 30
2.5.4 Realtime Measurements 33
2.5.5 Dynamic Root of Trust for Measurement 34

3 Chaavi: Webmail with Searchable Encryption 36

3.1 Motivating Example: Webmail Services 37
3.2 Background 38
3.2.1 Mail Architecture 38
Components 38
Privacy Threats 39
3.2.2 Pretty Good Privacy 40
3.2.3 Searchable Encrypted Data 41
3.3 Architecture 42
3.3.1 Browser 43
3.3.2 Browser Extension 43
3.3.3 Web Application 44
3.3.4 Database 44
3.3.5 Mail Server 44
3.4 Implementation 45
3.4.1 Browser Extension 46
3.4.2 Web Application: 47
3.5 Experiments 48
3.5.1 Time Complexity 48

Key Generation . . . 48
Encryption and Decryption . . . 49
Keyword Encryption . . . 50
3.5.2 Space Complexity . . . 51
Impact of increase in size on the keyword index . . . 51
Impact of increase on Final Message size . . . 52
3.6 Limitations of our implementation . . . 52
3.7 Conclusion . . . 53

4 A notification based remote attestation framework for IaaS 54
4.1 Related Work - Remote Attestation Infrastructure . . . 55
4.1.1 Integrity Measurement Architecture (IMA) . . . 55
4.1.2 Privacy Certificate Authority (CA) . . . 58
4.2 Virtual Machines based Architectures . . . 60
4.2.1 Terra . . . 61
4.2.2 HIMA . . . 62
4.2.3 In VM measurement . . . 64
4.2.4 Certicloud . . . 65
4.2.5 vTPM . . . 67
myTrustedCloud . . . 69
Implementation of Trusted IaaS using vTPM and Trusted Third Party (TTP) . . . 70
4.3 Our Work . . . 71
4.4 Architecture . . . 72
4.4.1 IaaS Cloud . . . 74
IaaS Trust Component . . . 75
4.4.2 Trusted Third Party . . . 76
Trust Verifier . . . 77
RIMM Datastore . . . 78
4.4.3 RIMM SaaS Cloud . . . 79

4.4.4 Client . . . 80

4.4.5 VM Policy . . . 81

Formalisms . . . 81

4.5 Scenarios . . . 83

4.5.1 Registration . . . 83

4.5.2 Verification . . . 85

4.5.3 RIMM policy update . . . 86

4.6 Establishing a minimal trusted computing base . . . 88

TCB using Debian Package Management (dpkg) . . . 89

4.7 Implementation . . . 90

4.8 Experiments . . . 90

4.8.1 Heartbleed Detection . . . 90

4.8.2 Overhead . . . 91

Booting . . . 92

Computations . . . 93

4.9 Limitations . . . 93

4.10 Conclusion . . . 94

5 Subjective and Dynamic Trust in SaaS **95**

5.1 Configurable Elements . . . 96

5.1.1 Storage Engine . . . 96

5.1.2 Policy Engine . . . 97

5.1.3 Monitoring Engine . . . 98

5.2 Configurations . . . 99

5.3 Architecture . . . 100

5.3.1 Application Server . . . 100

put (data, policy, config_info) . . . 101

get (data_id) . . . 101

get_tpm_quote () . . . 101

update (data_id, policy, config_info, key_{client}) . . . 102

delete(data_id, is_shred) . . . 102
5.3.2 Crypto Server . . . 102
crypt_init key (enc_type, operation, key, data_id) . . . 102
crypt (data) . . . 103
get_tpm_quote () . . . 103
5.3.3 Storage Server . . . 103
put(data, data_id) . . . 103
get(data_id) . . . 103
delete(data_id, is_shred) . . . 103
5.3.4 Monitoring Server . . . 104
5.3.5 Configuration of Servers . . . 104
5.4 Scenarios . . . 105
5.4.1 get_tpm_quote . . . 105
5.4.2 put . . . 106
5.4.3 get . . . 107
5.4.4 migrate . . . 108
Migration to trust level - 0 . . . 108
Migration to trust level - 0.5 . . . 108
Migration to trust level - 1 . . . 109
5.5 Implementation . . . 110
5.6 Experiments . . . 110
5.6.1 Performance of Put and Get . . . 110
5.6.2 Scalability Experiments . . . 111
5.6.3 Performance of Migrate . . . 113
5.7 Related Work . . . 114
5.7.1 Trust Model . . . 114
5.7.2 Policy Enforcement Mechanisms in Trusted and Untrusted Clouds . . . 116
5.8 Conclusion . . . 117

6 Subjective and Dynamic Trust in IaaS 118

6.1 Supporting trust relationships in IaaS . . . 118
6.1.1 Scenario 1: $User_1$ moves from trusting the provider to distrusting . . . 119
6.1.2 Scenario 2: $User_2$ starts to trust the provider . . . 120
6.1.3 Requirements . . . 121
6.2 Architecture . . . 121
6.2.1 Infrastructure Controller . . . 122
6.2.2 Trusted Pools . . . 123
6.2.3 Untrusted Pools . . . 123
6.2.4 Unallocated Servers . . . 123
6.2.5 Resource Middleware . . . 124
6.2.6 Modifications to Trusted Third Party (TTP) . . . 124
6.3 Algorithms . . . 125
6.3.1 Migration Algorithms in Resource Middleware . . . 125
6.3.2 Resource allocation algorithms in Controller . . . 126
Migration of VMs . . . 127
Defragmentation Algorithm . . . 129
6.4 Implementation Details . . . 130
6.5 Experiments . . . 131
6.6 Related Work . . . 131
6.7 Conclusion . . . 132

7 Conclusion 133
7.1 Contributions . . . 133
7.2 Application Scenarios . . . 135
7.3 Future Work . . . 136

Bibliography 139

Curriculum Vitae 150

List of Figures

1.1 The Plumber Problem 6
1.2 Alice and Pixelcloud 9
1.3 Contribution 14

2.1 Core P-RBAC 20
2.2 Trusted Boot 29
2.3 TPM Architecture 31

3.1 Email Architecture 38
3.2 Chaavi - Architecture 42
3.3 Sending and Searching for a Message 45
3.4 Key Management 47
3.5 Key Generation 49
3.6 Encryption and Decryption 50
3.7 Keyword Encryption Time 51

4.1 Integrity Measurement Architecture 56
4.2 ASCII Runtime Measurements 57
4.3 Privacy CA 58
4.4 Privacy CA: Trusted Computing Base 60
4.5 Terra Architecture [49] 61
4.6 HIMA Architecture [26] 63
4.7 In VM measuring framework 64
4.8 Certicloud Architecture Architecture [34] 66
4.9 vTPM Architecture [18] 68

4.10 myTrustedCloud Architecture 69
4.11 Trusted Cloud with TTP 71
4.12 Architecture 74
4.13 IaaS Server Components 75
4.14 Trusted Third Party 77
4.15 Registration Scenario 84
4.16 Verification Scenario 86
4.17 RIMM Update Scenario 88
4.18 Heartbleed Detection Workflow 92

5.1 System Architecture 100
5.2 Configuration of the servers 104
5.3 PUT 106
5.4 GET 107
5.5 Put, Get Performance 111
5.6 Plain Text with concurrent connections 112
5.7 AES Encryption/Decrpytion with concurrent connections 113
5.8 TPM with concurrent connections 114
5.9 Response time for trust transition 115

6.1 IaaS and Trust Relationship: Initial State 119
6.2 IaaS and Trust Relationship: Steps 120
6.3 Components of the architecture 122

List of Tables

4.1 Iterations per second for Setup A and Setup B 93

5.1 Configuration Parameters 99

List of Acronyms

List of Acronyms

AES Advanced Encryption Standard

AIK Attestation Identification Key

CA Certification Authority

CSC Cloud Service Client

CSP Cloud Service Provider

DRTM Dynamic Root of Trust for Measurement

IaaS Infrastructure as a Service

IMA Integrity Management Architecture

MTA Mail Transfer Agent

MUA Mail User Agent

PaaS Platform as a Service

PCR Platform Control Register

PET Privacy Enhancing Technology

PII Personally Identifiable Information

P-RBAC Privacy aware Role Based Access Control

PEKS Public-key Encryption with Keyword Search

RAD Rapid Application Development

RIMM Resource Integrity Measurement Manifest

RSC RIMM SaaS Cloud

SaaS Software as a Service

SMTP Simple Mail Transfer Protocol

TCB Trusted Computing Base

TCG Trusted Computing Group

TPM Trusted Platform Module

TTP Trusted Third Party

TVMM Trusted Virtual Machine Monitor

VM Virtual Machine

vTPM Virtual Trusted Platform Module

Chapter 1

Introduction

Cloud computing represents a model of computing, where consumers can rent a pool of configurable resources such as networks, servers, storage and services, from a Cloud Service Provider (CSP) [25]. There are three main types of cloud offerings: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). IaaS offers virtualized instances of bare machines leaving the installation and customization of softwares including the Operating System to the cloud computing clients. In PaaS, an application framework is provided to the clients for developers to develop their software with. A SaaS provider offers a particular application as a web service, which customers can customize to their needs.

Cloud Service Clients (CSC) can dynamically provision the resources as per the client's requirements, using a pay-per-resources business model. Cloud computing enables elasticity (grow or shrink as per the client's requirements) of resources, multi-tenancy, maximal resource utilization and the utility model (pay per use). These new features empower the provider to leverage large infrastructures through virtualization of resources and optimum job and resource management.

From the client's perspective, there are not only cost savings due to the economies of scale on the service provider side, but also the amount of investment on the client is decreased for buying the infrastructure upfront. Due to these benefits, users and enterprises are looking to delegate the storage of data and service execution on resources owned by a cloud providers. Forrestor (the independent market research firm) estimates that the cloud computing global market will increase from $40.7 billion USD in 2010 to $241 billion in 2020 [4].

Despite the benefits offered by the cloud computing paradigm, organizations are still wary of delegating their computation and storage to a cloud service provider. According to a survey conducted by IDC (a market research firm), the CIOs (Chief Information Officers) of enterprises identify *security* concerns as the top reason for them to not aggressively adopting the cloud [9]. A Fujitsu Research Institute study on potential cloud customers [21] found that 88% of potential cloud consumers are worried about who has access to their data, and demanded more awareness of what goes on in the physical cloud server.

Many clients, such as health-care companies, financial companies and government organizations require strict *privacy* and *security* requirements. A *policy* is a specification of these requirements. An example of a specification language that can be used to specify privacy policies is P-RBAC [86]. To enforce the security and privacy policy, clients need to *trust* that the remote cloud provider is not malicious and the provider will comply with the security requirements. There is a perceived lack of trust in cloud services, which is hindering some companies from completely embracing cloud computing.

These privacy, security and trust concerns of the cloud can be addressed by a combination of regulatory frameworks and supporting technologies [88]. The regulatory frameworks provides the clients with legal protection against any adversary provider and the supporting technologies aids the the clients to identify breaches and take any proactive and reactive measures against possible breach of the clients data.

The chapter starts with a discussion on the supporting technology, Privacy Enhancing Technologies (PET) with its limitations are presented in Section 1.1. This is followed by a motivation for the need of trust in cloud computing and the introduction of Remote Attestation in Section 1.2. Section 1.3 describes complex trust relationships by illustrating a real world example. The contributions of the thesis are enumerated in 1.4 and the scope of the thesis is discussed in 1.5. The chapter ends with thesis organization in Section 1.6.

1.1 Privacy Enhancing Technologies

Privacy Enhancing Technologies (PET) enable clients to transact with providers securely even if the clients do not trust the providers. PET denotes the set of tools and mechanisms that

allow users to protect their privacy online against adversaries. Blarkom et al. [108] defines PET as *a system of information and communication technology measures protecting informational privacy by eliminating or minimizing personal data thereby preventing unnecessary or unwanted processing of personal data, without the loss of the functionality of the information system.* Privacy preserving protocols [47], a class of PET, enable users to perform computation over cryptographically protected data. *Homomorphic encryption* and *searchable encryption* schemes are notable privacy preserving protocols.

Homomorphic encryption is an asymmetric encryption technique, where algebraic operations are performed directly on the cipher text which represents the encryption of plain text. This was first introduced by Goldwasser et al. [53], where the authors performed modular addition of two bits using multiplication of ciphertexts. Craig Gentry [51] designed an homomorphic encryption scheme that allows both addition and multiplication on plain text through their cipher texts.

Searchable encryption allows users to search for particular keywords on encrypted data. Public Key Encryption with Keyword Search (PEKS) [71] is one of the seminal works in the area of making encrypted data searchable. The authors of PEKS propose to encrypt the message using the Public-Private key infrastructure. Along with this cipher text a Public-Key Encryption with Keyword Search (PEKS) of each keyword (the words that make up the message) is appended to the final message. The PEKS of the keyword is used as the keyword index for searching in the server without revealing the exact plain text keyword (more in Chapter 3).

Although homomorphic encryption and searchable encryption are viable proven ways of preserving privacy of data in the cloud without compromising on the functionality, cryptography increases the computational and storage overhead on the server [94]. Computation over encrypted data even though theoretically possible is not yet practically feasible [83].

1.2 Need for trust: Remote Attestation

As the solutions proposed by PET are mostly in the theoretical realm, clients are forced to *trust* the cloud provider with the data and hope that the provider will not breach that trust. Trust is defined as "a particular level of subjective assessment of whether a trustee (cloud provider)

will exhibit characteristics consistent with the role of the trustee" [120]. The client establishes trust on emotional and cognitive (evidence based) grounds. To enforce the strict security requirements on the server, the trust relationship between the client and server should be formed more based on cognitive, evidence based grounds. The evidence should be unforgeable and should assure that the server will not act against the client's interest.

Remote attestation provides such evidence by allowing clients to accurately verify if the remote server's state is compromised. A server can be trusted if the client can accurately *verify* all the software binaries that the server has executed [50]. The veracity of a software is established through its identity, which is expressed by means of the hash[1] of the software binary. For verification of the server's state, the measured hash of all the software binaries (measurement list) is sent to the client. The client performs the comparison against known software hash values, whose security has been verified. This will enable clients to verify that the server is free of malware or any unauthorized software.

Remote attestation [50] refers to the *process of authenticating and verifying the state of the remote platform and its operating system outside of the platform*. The remote platform can either be hosted on a physical server or a Virtual Machine (VM) in the physical server or both. In the context of cloud computing, remote attestation of the cloud server is performed either by cloud clients or a trusted third party on behalf of the cloud clients.

Based on remote attestation, *trusted computing* technology was developed by the Trusted Computing Group (TCG). It provides specifications for securely reporting and verifying a remote platform (i.e. server hardware and software).

Existing work in remote attestation of the server includes: securely collecting and storing information about the software state (hash values) of the server [50], methods for using the information on the state locally in the server [17, 30], for conveying the state information to an external client for remote attestation [106, 98, 34] and for managing the list of software that is allowed to be executed in the server [60].

In theory, it should be possible to test whether the hardware and the software hosted in the server are secure and remotely attest if a server is hosting the tested hardware and software components. However, remote attestation has several practical limitations. Establishing a

[1]Hash is the unique short digest of a binary, that is more efficient to communicate than the exact software.

trusted computing base (TCB), the software and hardware stack that is audited and well tested for the security of a particular application is challenging. With the advent of technologies such as Rapid Application Development (RAD), maintaining the latest secure software updates on remote server is non-trivial. Moreover, economically, a remote attestation infrastructure can be expensive to the client.

The challenges and costs imposed by PET and remote attestation make them an undesirable option for clients that trust the provider already without the attestation provisions. This trust may arise from several *subjective* factors such as proven history of provider's strong SLA compliance with the client. In some context when the client is dealing with insensitive data, the clients need not completely trust the provider for operations on the data. For example, clients absolutely need to trust the banking provider for the everyday banking needs, however they do not really need to trust the social media platform to share their messages with their friends or followers.

1.3 Trust Relationships

Trust relationships are complex. Trust is *subjective*. Different clients may perceive the same provider differently. For example, a forgiving client may be willing to accept more failures from the provider while a less forgiving may not. Trust is also *dynamic*. A client's trust perception of a server may change over time based on the performance of server. To illustrate this relationship a societal (real world) scenario is presented through the plumber problem.

1.3.1 The Plumber Problem

The plumber problem is illustrated in Figure 1.1. Morpheus wakes up one day to find a broken pipe in his house. He needs to call a plumber to fix it. He has the following options: Neo, the most trusted plumber and the busiest in the town; Cypher, recommended by the government and Smith, a plumber with a criminal record. Neo chargers more for his services followed by Cypher and then Smith. Who should Morpheus choose for fixing the pipe?

There is no single right answer to this question. The safest plumber here, Neo, may not be the one Morpheus calls. The answer is dependent on various factors:

Figure 1.1: The Plumber Problem

- *Trust assessment* of Morpheus about the plumbers: The most trustworthy plumber in the town may not be the most trusted plumber by Morpheus. Morpheus may personally know and trust Cypher and Smith even though Neo is trusted by the entire town.

- *Mechanisms* that Morpheus can deploy to work with an untrusted plumber: Morpheus may have the ability to inspect the working of the plumber directly or remotely inspect through a camera. This may give him an option to work with an untrusted plumber safely.

- Location of pipe: Morpheus may choose an untrusted plumber to fix the pipe even without the mechanisms in place, if the pipe is located in a distant garage and the plumber cannot do any damage to the property of Morpheus.

- Economical constraints of Morpheus.

If Smith is someone Morpheus knows in person compared to Neo and Cypher, Morpheus may choose Smith as the plumber. Morpheus may also choose Smith if he can inspect Smith when he is plumbing. If Morpheus does not want to take a risk with Smith, he may choose Neo, paying Neo extra money. Morpheus may choose Cypher if he finds Cypher the right value for the money.

Morpheus's trust assessment of each plumber can be completely different from his friend Trinity's, making it a subjective assessment. This is because Trinity may be friends with Cypher where as Morpheus may be friends with Smith. Moreover, Morpheus may trust a plumber today and decide not to trust him tomorrow based on the plumber's performance. We observe that for a simple question on choosing a plumber there are various subjective and dynamic parameters that affect making a decision. Despite all this, the *process* of calling a plumber and fixing the pipe, in general seem to work.

There are two kinds of trust that influence the decision of Morpheus: *personal* and *impersonal trust* [101]. A personal trust requires intimate knowledge of the plumber by Morpheus. If the plumber is Morpheus' friend, the trust Morpheus has on the friend is not tied to the plumber's actions in the past. It's a reliance that, the plumber has good intention with respect to Morpheus and will not act against Morpheus' interests.

Shapiro et al. [102] studied *Impersonal trust* elaborately and explored how societies control trust relationships between entities that are not not influenced by personal relations. Our society has formed several systems that enable the process of working with entities (in this case plumbers) of different trust. These systems are referred by Shapiro et al. [102] as *guardians of impersonal trust*. There are legal systems in place to punish the criminal conduct of any entity, and there are recommendation systems (through word of mouth, sites such as yelp.com, plumbers association of canada) that make people aware of competent and incompetent entities.

The process of calling a plumber works because we tend to trust these systems more than people. These systems have become the backbone of our society. One may not trust the plumber, but may trust the systems that produced the plumber. There are laws that govern the system that keep the delinquent plumbers in check. There are mechanisms (such as monitoring the plumber while working) in place, that give users like Morpheus and Trinity enough flexibility to make the decision within the user's constraints and limitations.

1.3.2 Trust Relationships in Cloud

In a cloud computing context, the systems that govern computing and the mechanisms that protect the users are currently evolving. Therefore trust in cloud computing is considered as largely unidimensional without the subjective and dynamic properties.

To illustrate a cloud computing system with a multidimensional trust, let us consider an example of a photographer Alice. Alice is a photographer who wants to store her photograph collection with a provider called Pixelcloud. She has the following privacy requirement: Pictures should only be visible to her and her friends that have permission to view the photographs. This can be formalized using a P-RBAC [86] like policy specification language as follows:

```
(FR, ((RD, Catalog), Viewing, OC=Yes, Notify(ByEmail)))
```

The notation `FR` is used to denote the role: `friends of Alice`. The notation `(RD, Catalog)` denotes that the users in `FR` can only execute a read only operation, denoted by `RD`, on the photgraph collection, which is denoted by `Catalog`. The notation `Viewing` specifies the policy purpose. The condition, denoted by the notation `OC=YES`, represents that the owner needs to give consent before a read operation on the catalog can be carried out. The obligation's action is denoted by the notation `Notify(ByEmail)`, which states that Alice is notified by email for each read access of the catalog.

When the data storage is outsourced to Pixelcloud, Alice may or may not trust Pixelcloud to enforce the policy. If Alice trusts Pixelcloud, then she can depend on Pixelcloud to enforce the policy as per the specification. However, if Alice does not trust the Pixecloud, she will take the sole responsibility for enforcing the policy.

When Alice is uncertain if she can trust Pixelcloud or not, she will have to continuously monitor to make the Pixelcloud accountable when there is a policy violation. We present these three scenarios below (Figure 1.2).

The first scenario to be considered is that *Alice distrusts PixelCloud*. A possible approach to enforcing the privacy policy is to have Alice encrypt her photograph collection before sending it to PixelCloud. PixelCloud is not able to read Alice's photograph collection since the data is encrypted and PixelCloud is unaware of the keys for decryption. Alice privately shares the key needed for decryption with her friends. A friend uses the key to decrypt the encrypted

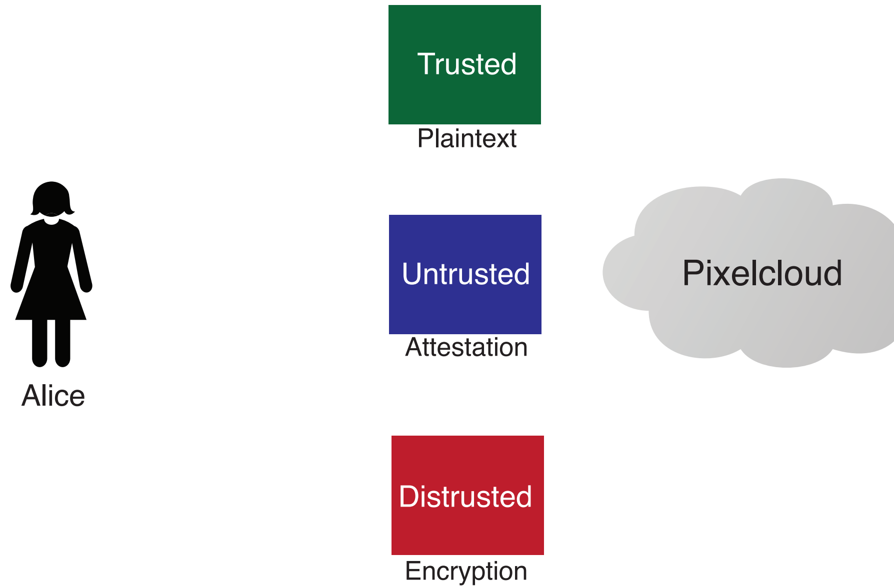

Figure 1.2: Alice and Pixelcloud

photograph collection in response to a request for the data from PixelCloud. Any unauthorized access of the encrypted files in Pixelcloud will be futile without having access to keys. Alice's friend uses the client software provided by Pixelcloud to stop the friend from using the data for any other purpose than for what it is intended by not allowing the data to be transmitted or stored.

When the data is encrypted and keys are with the owner, the data is secure and no policy violations by the provider can occur. Therefore, an argument could be made that a CSP should always be considered to be untrusted. However, the performance overhead introduced by such policy enforcement mechanisms may be prohibitively high [52, 99, 92].

If *Alice trusts PixelCloud*, she may provide her data unencrypted to PixelCloud and delegate the responsibility for policy enforcement to PixelCloud. The performance overhead is reduced since encryption and decryption operations are not used.

The responsibility of policy enforcement is shared between Alice and PixelCloud when *Alice is not sure if she can trust PixelCloud*. Alice requests that PixelCloud encrypts her catalog. PixelCloud's records transactions involving Alice's catalog. The logs with transaction information can be inspected by Alice to monitor for unauthorized accesses to her catalog.

When a friend of Alice, Bob, accesses the catalog, PixelCloud decrypts the catalog and presents it to Bob through client software provided by PixelCloud to Bob. The client software executing in Bob's computer will ensure that the picture shared with Bob is used only for the purpose it is shared. In this scenario of uncertain trust, If PixelCloud is an adversary or has a malicious insider, then the logging can be circumvented. This is addressed using a Trusted Platform Module (TPM) with a check of integrity values of the software that is executed in the server. The performance assuming uncertain trust, is higher than if PixelCloud is trusted but not as high, if PixelCloud is distrusted.

We observe that Pixelcloud has three different "views" (trusted, distrust and uncertain trust) and Alice can choose a view based on her trust perception of Pixelcloud.

In industry, unlike the example of Pixelcloud, we do not notice a single provider behaving in a trusted mode to a client and an distrusted mode to someone else. Google, Facebook, Dropbox are used even when users do not have complete trust on them. Users who highly suspect these providers switch to secure (sometimes limiting) alternatives such as Duckduckgo, Diaspora, Wuala etc.

We propose that there is a need for services to support the subjective and dynamic nature of trust in general. When trust assessment is performed based on subjective factors, the final configuration requirements (remote attestation or no attestation) of the provider can vary from client to client. Therefore the provider needs to let clients choose the final configuration based on the client's subjective trust assessment. The clients should be able to accommodate the dynamic nature of trust: when the trust on the cloud provider changes over time, the clients should be able to change their configuration.

Current work in the state of the art does not study these trust relationships and the corresponding policy enforcement mechanisms.

1.4 Contributions of Thesis

The contributions of this thesis are illustrated in Figure 1.3. Listed below are the contribution of the thesis:

- **PET in cloud architectures:** This thesis explores the state of the art in PET and proposes a privacy preserving architecture for a webmail SaaS system without compromising on the functionality. The prototype is benchmarked and based on the results show the feasibility to architect a privacy preserving solution for webmail systems in a real working environment. The thesis also discusses the limitations of PET and future work for PET in cloud architectures extensively.

- **Practicality of remote attestation in SaaS and IaaS:** A notification based remote attestation infrastructure is proposed that supports verification of servers and virtual machines using a TTP (trusted third party) and RIMM (Resource Integrity Measurement Manifest). The RIMM maintains a list of known secure software integrity values. The remote attestation of the server and VM is delegated to a TTP. The software state of the server and the VM is verified continuously by the TTP in predefined intervals through polling. The proposed remote attestation infrastructure is studied in a web application scenario. A known minimal trusted computing base (TCB) is established and its security implications and overhead of attestation are studied in detail.

- **Subjective and dynamic nature of trust for a cloud:** The dynamic and the subjective nature of trust is introduced and its implications to cloud computing is studied in detail in both the SaaS and IaaS cloud.
  For the SaaS cloud, a policy based approach to the implementation of subjective and dynamic trust so as to enable privacy policy enforcement in a SaaS cloud is proposed. An abstract model containing computational, storage and monitoring unit with configurable elements is introduced and algorithms that reflect how a change of trust influences the configuration of these elements are described.
  In the IaaS context, the server pool is divided into virtual trusted and untrusted pools. The trusted pools supports the notification based remote attestation infrastructure whereas the untrusted pool do not have the overhead of the remote attestation infrastructure. We describe algorithms for the migration of VMs between the pools based on the trust perception and study the overhead of the system.

1.5 Scope

The NIST definition of cloud computing [80] enumerates the following essential characteristics of cloud computing:

1. *On demand self-service* A consumer can dynamically provision services and resources, as needed without any human intervention.
2. *Broad network access* The resources can be accessed through standard mechanisms that enables use of heterogeneous clients such as mobile phones or browsers.
3. *Resource Pooling* The provider serves clients using multi-tenant model to optimize resource usage at the provider's end.
4. *Rapid elasticity* The consumers can scale-up or scale-down the resource usage automatically.
5. *Measured Service* The provider serves resources and services through a pay-per-use business model, where computing is treated as utility such as electricity and water.

The proposed architectures in this thesis addresses on demand self service, broad network access and resource pooling. The architectures do not directly focus on rapid elasticity and measured service. However, the architectures can be expanded to support rapid elasticity and measured service in future work.

1.6 Thesis Organization

Chapter 1 presents the introduction for the the rest of the dissertation.

Chapter 2 presents background on Privacy Enhancing Technologies (PET) and Trusted Computing with focus on remote attestation.

Chapter 3 explores the practicality of PET by presenting a prototype of a web mail system using searchable encryption. The prototype is benchmarked and its feasibility in a real world scenario is discussed in detail. This chapter concludes with PET's disadvantages and the need for trusted architectures.

In Chapter 4, a notification based remote attestation infrastructure for a Infrastructure as a Service (IaaS) cloud is discussed to address the limitations presented in Chapter 3. The chapter describes the components of the infrastructure along with the protocols and algorithms. A trusted computing base for hosting a web application in a virtual machine is established and the system is benchmarked to study its overhead.

Chapter 5 studies the subjective and dynamic nature of trust in detail. It presents a SaaS cloud, supporting dynamic configuration (using PET and remote attestation) of its services based on the trust assessment of the client. We describe an architecture and algorithms for policy mapping that considers viability of trust and present a cloud storage framework that enforces privacy policy according to the varying trust levels of the CSP. An abstract model containing computational, storage and monitoring unit with configurable elements is presented. Algorithms that reflects how a change of trust influences the configuration of elements are described.

Chapter 6 studies subjective and dynamic trust in the context of an IaaS. Trusted and non-trusted pools are established and the migration of VMs based on the trust assessment is presented in detail.

Chapter 7 presents the future work and the dissertation concludes.

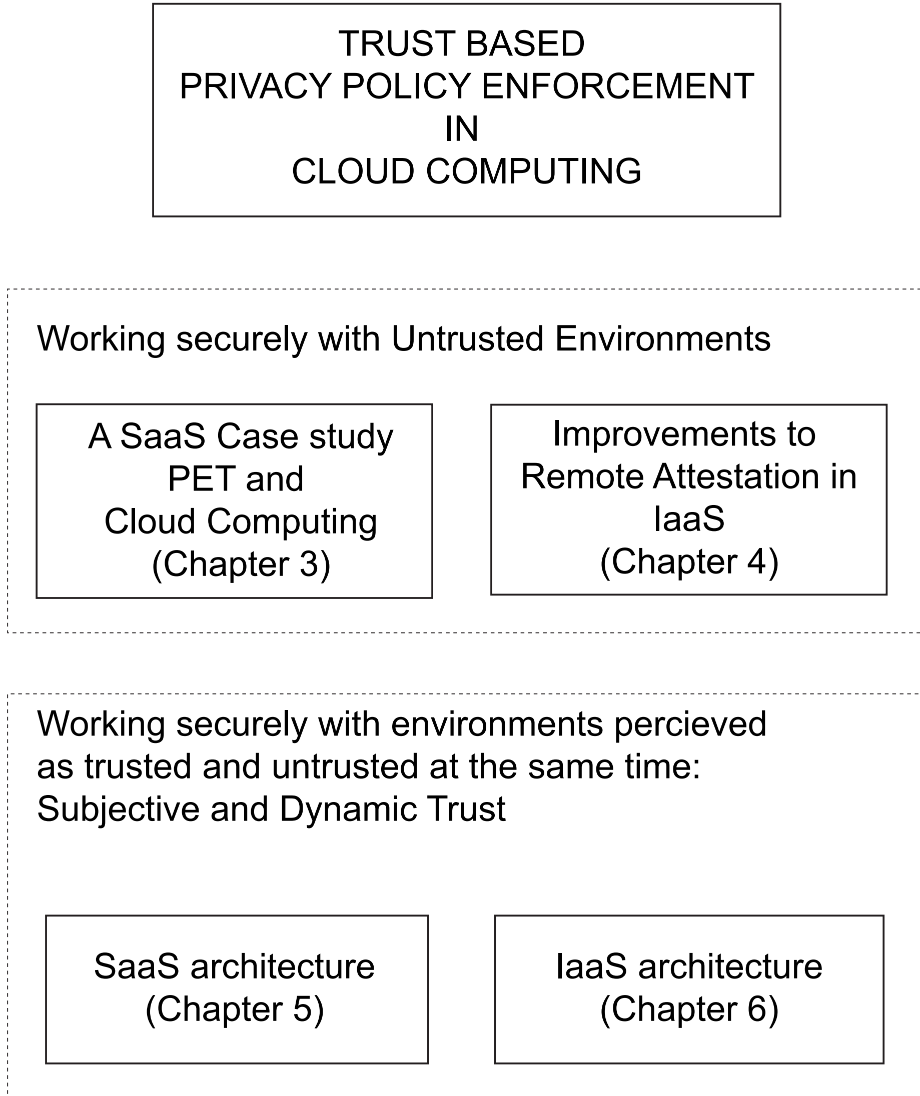

Figure 1.3: Contribution

Chapter 2

Background

In this chapter we present the background required for the rest of the thesis. We present the definition of Cloud Computing and its service models and deployment models in Section 2.1. In Section 2.2 we discuss privacy, security and trust implications of cloud computing. In Section 2.4, we introduce PET and present searchable encryption in detail. Section 2.5 presents a primer on remote attestation.

2.1 Cloud Computing

The idea of cloud computing was first proposed in 1963 by J.C.R. Licklider in "Intergalactic Computer Network" [69]. Licklider envisioned that everyone on the globe to be interconnected and accessing programs and data from any site, from anywhere. John McCarthy suggested that the time sharing technology of 1960s might lead to a future where computing power and applications could be sold as a public utility [87]. Over the years, the cloud computing idea evolved considerably under the several terms such as grid computing, utility computing and Service Oriented Architectures (SOA).

The word "cloud" was first used in the above context by Google ex-CEO Eric Schmidt to describe the business model of services across Internet. Amazon in 2006 launched Elastic Compute Cloud (EC2) as a commercial web service that allows small companies and individuals to rent virtual machines on which to run the company's own computer applications. Since then there have been several definitions for cloud computing [109]. We adopt the definition of

cloud computing by NIST [79] as it covers all the key concepts of cloud computing. The NIST definition of cloud computing states,

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

The service models, deployment models and characteristics of the cloud computing are presented in this section.

2.1.1 Service Models

The configurable computing resources, denoted in NIST definition, are provisioned to cloud customers in several layers, referred to as *service models*, each offering discrete capabilities. The service models provided by cloud computing are the following:

- **Infrastructure as a service (IaaS):** IaaS refers to the service model where resources such as the physical/virtual machines from the Cloud Service Provider (CSP) are rented to the Cloud Service Clients (CSC). The CSP may also enable resource management capabilities through a public Application Programming Interface (API), which can be utilized by the CSC for dynamic provisioning. Examples of IaaS include Amazon and Rackspace.

- **Platform as a Service (PaaS):** In PaaS, the CSP allows the CSC to develop new applications using the software stack and runtime environment (platform) provided by the CSP. Examples of PaaS include Google App Engine and Microsoft Azure.

- **Software as a Service (SaaS):** The SaaS refers to the delivery model where the provider implements the application (software) for the client and hosts the client data in a separate client specific virtual container. In SaaS, the client delegates the application development and the technical expertise to the host or scales the developed application to the provider. The CSP uses its infrastructure to provide the services to the client. Examples of SaaS providers include Salesforce and Google Docs.

These service models of cloud computing can be deployed in several ways in the cloud server.

2.1.2 Deployment models

The cloud computing model offers flexibility in its deployment, depending on the security requirements of the organization, organizational structure and the location of the infrastructure. The following are the four common deployment models [80]:

- **Private Cloud:** The cloud infrastructure is owned and managed within a private network by an organization. The infrastructure is not shared with any other organization.
- **Public Cloud:** The cloud infrastructure is owned by the cloud provider and the organizations (CSC) rent the required resources from the provider. The infrastructure is typically shared between different clients through virtualization of resources.
- **Community Cloud:** The cloud infrastructure is shared between organizations with shared interests. It may be owned and managed by one or more organizations in the community.
- **Hybrid Cloud:** Hybrid clouds are the combination of two or more different cloud deployment models (private, public or community) that are connected together by well defined secure protocols.

2.1.3 Characteristics

Along with the flexibility of service and deployment models, cloud computing offers the following key functionalities [79]:

- **On-demand self service:** The client can automatically provision resources from the server based on its demand without any human intervention.
- **Resource pooling:** The provider's computing resources are pooled to serve multiple clients using a multi-tenant model with different physical and virtual resources dynamically assigned and reassigned according to the client's demand.

- **Elasticity:** Capabilities provided to the client can scale and shrink as per the demand.
- **Measured service:** Cloud systems automatically control and optimize resource use as per the requirements of the client. The clients can be charged based on their usage of the resources (Utility model).

These functionalities and the cost benefits offered by cloud computing due to economies of scale make cloud computing extremely attractive for businesses and governments.

In terms of security, Small and Medium-sized Enterprises (SMEs) may gain from switching to a cloud computing provider as the cloud provider may have better security infrastructure and expertise to maintain them. Moreover, a privacy breach or security incident with respect to a SME may not be as damaging compared to the benefits offered to the clients by the provider as the functionality.

However, clients with mission critical applications, such as banking, healthcare and financing industries, may have stricter privacy and security requirements. Adopting cloud computing is risky for these enterprises, as they lose control over the data stored in a cloud computing provider's infrastructure. Loss of privacy can prove catastrophic for these clients.

To effectively work with the provider, the clients with mission critical applications need to make these assumptions: The cloud provider has the necessary infrastructure and expertise to secure the client's data from privacy breach and the cloud provider will not act against provider voluntarily or involuntarily against the client's interests. These assumptions make it necessary for the clients to trust the cloud provider.

2.2 Privacy, Security and Trust

Privacy, Security and Trust are complex concepts with many definitions. In this section we present the privacy, security and trust implications in cloud computing.

2.2.1 Privacy

In 1890, Warren et al. [114] defined privacy as the "right to be let alone" with the focus on protecting individuals. Privacy is recognized in the Convention for the Protection of Human

Rights and Fundamental Freedoms. Section 8 of the Canadian Charter of Rights and Freedoms and Section 4 of the Bill of Rights in the USA provides constitutionally enforced privacy rights for Canadians and US citizens respectively. In democratic societies, the right to privacy is a fundamental right [74].

In information theory, data privacy is described as the right of the user to know where the owner's personal data is transferred to and what it is used for. It is considered as "the indefeasible right of an individual to control the ways in which personal information is obtained, processed, distributed, shared, and used by any other entity" [20].

In today's technological world, there are several services that aim to enhance people's lives by using the personal data of individuals as the main resource. For example, a service can better target a particular user demographic, by knowing the users' date of birth and address. However, these benefits bring alongside new risks such as identity theft, surveillance, fraud, e-mail spams etc. The client's data that uniquely identifies the client, referred to as Personally Identifiable Information (PII), should not be be used for any purpose by the provider that the client has not authorized for. PII should be strictly bound by the privacy requirements of the client. A *privacy policy* is the specification of the privacy requirements. An example language for specifying privacy policies is P-RBAC [86] which has an XML representation.

P-RBAC has sets of entities: users, roles, data, actions, purposes, obligations and conditions (Figure 2.1). In P-RBAC, permissions are assigned to roles and users obtain permissions by being assigned to roles. Data is related to a user and action is the activity that can be performed on the data. The data is usually bound to a purpose (e.g., data collected for one purpose should not be used for other purposes without user consent). Obligations refer to actions that need to be performed after a particular action is executed on the data. Conditions are the pre-requisites that must be satisfied before performing an action.

Example Privacy Policy: A data owner has the following requirement: The user type researchers, can only read the zipcode field from the data for the purpose of Research, provided the data owner has given consent for that particular researcher. For every read by a researcher the owner is notified by their official email. This policy is expressed in P-RBAC as the following:

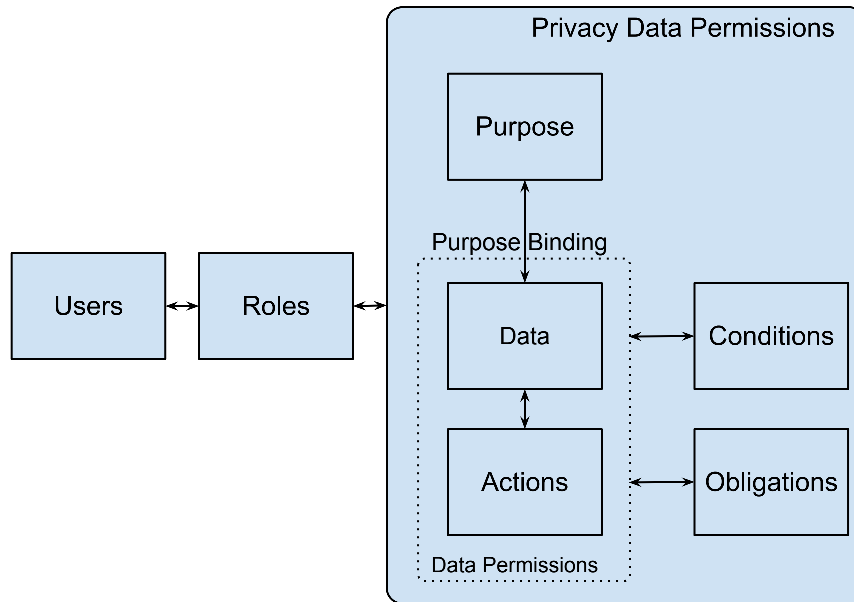

Figure 2.1: Core P-RBAC

```
(RS, ((RD, ZCode), Research, OC=Yes, Notify(ByOfficialEmail)))
```

There are two types of users: Researchers (RS) and Customers. We assume that there are also two roles which have the same names. The notation `(RD, ZCode)` denotes the data (`ZCode` for zip code) and action (`RD` for read only). The notation `Research` specifies the purpose. The notation `OC=Yes` denotes the condition OwnerConsent (the owner needs to give consent for revealing this data).

The notation `Notify(ByOfficialEmail)` is the obligation which states that the the customer is notified by email for each read access of the zipcode. The policy specification is independent of any configuration of services needed to support it.

Cloud computing and Privacy: Cloud computing is based on the concept of outsourcing storage and computing of data (sometimes sensitive data) to a third party. With the increased market appeal of cloud computing, there is growing concern over issues of data privacy within the cloud based architectures. In the cloud computing context, the privacy policy is created by the consumer of the cloud services, client and it is enforced by the server. Pearson et al. [88] studied the following factors that affect the privacy of a cloud client:

- Lack of user control of disclosed data with a cloud provider.
- Lack of training and expertise of the provider in handling PII.
- Unauthorized secondary usage of the data shared with the provider.
- Complexity of regulatory compliance in a dynamic cloud infrastructure, where data is hosted simultaneously in multiple countries with different regulations.
- Addressing transborder data flow restrictions where certain types of data (such as health information in USA) are not allowed to be transferred to a data center outside of the country.
- Legal issues such as litigations a CSP may face to handover the data to a third party or government.

The cloud faces the same privacy issues as other service delivery models, but it magnifies existing issues due to the innate nature of delegation in cloud. The privacy issues in cloud are deeply connected to data security.

2.2.2 Security

Security is one of the planet's oldest activities [101]. It is hardwired into our ancestors for survival against perceived threats from nature. Over the course of our evolution, security has taken different forms: Physical security, national security, financial security, home security, food security, school security, information security are few of them.

Security in information theory [88] is the preservation of *confidentiality, integrity and availability* of information. *Confidentiality* is the assurance that only the intended and authorized entities have access to the information. *Integrity* guarantees that the data is not modified in storage or transmission except by authorized entities. *Availability* ensures information is available to authorized entities.

Security is a necessary, but not a sufficient condition for privacy [88]. Privacy specifications allow clients to specify access and usage restrictions on the data, whereas security focuses on how to prevent unauthorized access to information, while ensuring authorized access.

Cloud computing and Security: As stated in Chapter 1, a survey conducted by IDC (a market research firm) found that the Chief Information Officers of enterprises identify security concerns as the top reason for them to not aggressively adopt cloud technologies [9]. Cloud Security Alliance (CSA) [59] enumerates the following threats to the data in a cloud computing environment.

- **Sharing of data with an unauthorized party:** The cloud provider could compromise the confidentiality of the client's data by sharing the data with unauthorized parties. This can go against the Service Level Agreement (SLA) between the client and provider. The client may never be aware of such a breach.

- **Corruption of data stored:** As the cloud provider has root access to physical machines, the access allows the provider to modify/delete client's data. The provider could tamper with the data making the data non-usable or modify the data in a way that the system cannot detect the modification. This poses a serious threat to the integrity of the application data.

- **Malicious internal users:** The employee of a CSP who has root access to these physical machines, could access the data and use it for their own advantage.

- **Data loss or leakage:** When a virtual machine is used in an infrastructure, it poses a variety of security issues [48] which could compromise the data security. Moreover, when the facility that hosts the user's data is subjected to a natural calamity, it could risk the loss of the user's data.

These security issues need not hinder the clients from adopting cloud computing. Clients can either deploy mechanisms such as Privacy Enhancing Technologies (PET) to protect their PII or *trust* the provider is competent and will not act against the client's interests.

2.2.3 Trust

The concept of trust spans across several domains. Sociologist Piotr Sztompka stated that "trust is a bet about future contingent action of others" [105]. In psychology, trust "is believing that the person who is trusted will do what is expected" [56].

Computational trust in cloud systems is defined by Yew et al. [120] as “a particular level of subjective assessment of whether a trustee (cloud provider) will exhibit characteristics consistent with the role of the trustee”. A model that aids in the calculation and analysis of computational trust, is referred to as a trust model. Sabater et al. [97] classified these models as conceptual models, information sources, direct experiences, witness information, sociological information and prejudice. Trust is not unidimensional. It has the following properties [120]:

- **Quantifiable and Comparable:** A trust assessment of a trustor about a trustee can be converted to a single scalar value which can be compared with the trustor’s assessment of another trustee.
- **Subjective:** A trustor’s assessment of the trustee can be completely different to some other trustor’s trust assessment.
- **Dynamic:** A trustor’s trust on a trustee can evolve/devolve with time.
- **Multidimensional:** A trustor may trust a trustee for a specific purpose but for not any other purpose.
- **Reflexive, Non-symmetrical and Non-Transitive**.

Trust is essential for every society to work. Bruce Schneier in Liars and Outliers [101] proposed when there is a lack of trust or *dilemma* on the part of the trustor several societal pressures can be introduced to induce trust (cooperation). These social pressures can be categorized as follows:

- **Moral pressure**: Pressure arising from an entity’s[1] convictions on what is right and wrong.
- **Reputational pressure**: Pressure that forces entities to follow group norms so that they do not get bad reputation.
- **Institutional pressure**: Rules and laws of a state, ensure *institutional pressure*. They induce entities to conform to the rules by imposing penalties and sanctions on defaulters.

[1] An entity may denote either a person or an organization

- **Pressure introduced through Security systems**: Security mechanisms induce cooperation, prevent defection and compel compliance. They include proactive and preventive systems such as door locks, alarms, tall fences and reactive systems such as forensic and audit systems.

In this work we study in detail about inducing trust in the cloud service provider through security systems. We focus on proactive preventive systems such as Privacy Enhancing Technologies (PET) and reactive systems such as remote attestation that apply social pressures on the cloud service provider to conform to the accepted SLA. We do not study the calculation of trust through a trust model. Trust models for clouds not only use performance related attributes (downtime, responsiveness of services) to assess the trustworthiness of a cloud but also, attributes that represent privacy compliance. We assume that there is a trust model that will provide the client with trust values.

2.2.4 Risk Vs Trust

A concept that is closely related to trust is risk. Risk is the statistical expected value of an unwanted event that may or may not occur [55]. Both trust and risk are essential tools used for making decisions in an uncertain environment [63]. In the context of cloud computing risks can arise from the following inherent issues in cloud computing [25]:

1. Business Continuity and Service Availability
2. Data Lock-in
3. Performance unpredictability
4. Data confidentiality and other security issues

Each of these inherent issues pose a potential risk to a cloud computing consumer. The focus of this work is on (4), addressing the data confidentiality and the security issues.

2.3 Threat Model

Threat models in a cloud computing infrastructure are discussed in the literature by McCune et al. [77], Bertholon et al. [34] and Bleikerts et al. [35]. Bertholon et al. [34] does not elaborate on the threats in cloud server and remote adversary. Bleikerts et al. [35] discusses threats arising from the cloud clients and the end users of the client's services. We presented the security threats to data stored in cloud computing by CSA in Section 2.2.2. To concretize the threat model for the rest of the thesis, we introuduce a model that is very similar to the one discussed in Mccune et al. [77] in a remote attestation context. The threats from the client and the end users of client's services are grouped under remote adversary. Threats for the client in a cloud computing architecture can arise from the following entities: cloud server or a remote adversary.

- **Cloud server:** A cloud server administrator has not only complete access to all the information stored in a cloud server but also has physical access to the server's hardware. They can execute and modify any binary in the server and have the ability to inject malicious code in an executing binary by modifying the memory. In the extreme case a malicious administrator has the ability to replace the bios, bootloader and the operating system with malwares. Physical attacks against the CPU, Trusted Platform Module (TPM) or the system busses are not considered as part of this threat model.

- **Remote adversary:** A remote adversary can either attack the cloud server or try to compromise the communication channel between the server and the client. The remote adversary has access to the IP address of the cloud server and can deploy state of the art attacks against the server.

As in any standard secure communication in the Internet, the Dolev-Yao intruder model [42] is considered for network communication. A network adversary has the ability to analyze, interject or modify network traffic between cloud server and the client. However, the adversary cannot break cryptographic primitives.

In a cloud computing infrastructure the focus is largely on the cloud server adversary. Dealing with remote adversary is well studied. A server is protected against remote attacks by in-

stalling firewalls and Intrusion Detection Softwares (IDS) [96]. It is also common among the system administrators to follow some of the best practices in industry to secure against remote attacks. The communication channels are protected using an encrypted SSL based pipeline [41] for transmitting data in the network.

2.4 Privacy Enhancing Technologies (PET)

Privacy Enhancing Technologies (PET) induces trust on the cloud service provider by providing with a set of tools and mechanisms that cryptographically protect the clients privacy online against the cloud service providers. PET is defined by Blarkom et al. [108] as "*a system of information and communication technology measures protecting informational privacy by eliminating or minimizing personal data thereby preventing unnecessary or unwanted processing of personal data, without the loss of the functionality of the information system.*". Privacy preserving protocols [47], a class of PET, enable users to perform computation over cryptographically protected data.

PET technologies include:

1. Privacy management tools that enable inspection of server-side policies that specify the permissible accesses to data.

2. Secure online access mechanisms to enable individuals to check and update the accuracy of their personal data.

3. Anonymizer tools, which will help users from revealing their true identity by not revealing the PII (Privately Identifiable Information) to the cloud service provider.

Privacy Enhancing Technologies (PET) can be used by the developers of the application to enhance the individual's privacy in an application development environment. In this section, we survey state of the art in PET.

Homomorphic Functions: Homomorphic encryption schemes refer to asymmetric encryption techniques, where algebraic operations on plain text can be performed directly on a respective cipher text. This was first introduced by Goldwasser et al. [53], where the authors

performed modular addition of two bits using multiplication of ciphertexts. The two kinds of homomorphic functions are the following:

1. Partially homomorphic functions and
2. Fully homomorphic functions

Partial homomorphic functions enable either addition or multiplication on plaintexts, but not both. However, in a fully homomorphic scheme, both operations are supported. Fully homomorphic functions, allows executions of programs in an untrusted party without revealing the input to the party. The untrusted party can been seen as a cloud provider.

Craig Gentry [51] described the first fully homomorphic encryption scheme based on lattice-based cryptography. However performing Google search on encrypted keywords using homomorphic encryption based on Gentry's scheme will increase the computing time by trillion cycles.

Homomorphic encryption remains in the theoretical realm as more advanced abstractions need to be created for using homomorphic functions in practical applications.

Privacy By Secure Computation: The objective of secure computation is to evaluate a function f that takes inputs from two parties A and B without revealing the exact inputs to each other. Yao's protocol [119] provides some of the basic techniques to perform a computation in a secure way without revealing the inputs. Yao's protocol forces the expression of a computation problem in terms of logical circuit using gates. The input of each gate is randomly encrypted and then the final resulting output is decrypted to get the exact answer of the computation. The encryption and the decryption is done at the client's end. The expression of a simple problem using Yao's protocol is found to be non-trivial. Applications that typically reside in the cloud (e.g., mail) are too complex to be converted into Yao's circuits.

Privacy By Encryption: Privacy can be enforced by encrypting all the data that is stored in the cloud. The main issue is that the cloud can be only used for storage of the data. As the data will be unrecognizable to the cloud service provider, it will not be possible for the cloud service provider to process the data nor to perform some number crunching tasks. Searchable

encryption uses an algorithm, which allows users to encrypt the data and then provides the server with trapdoor information [36], so that the server can search for a given string through the searchable encryption algorithm. This part is discussed in detail in Section 3.2.3 of Chapter 3.

Privacy-Preserving Multi-keyword Ranked Search over Encrypted Cloud Data [38] proposes a new encryption scheme for keyword search over encrypted data in cloud computing environment with privacy and performance requirements.

Although searchable encryption is a proven way of preserving privacy of data in the cloud without compromising on the functionality of searching, cryptography increases the computational overhead on the server [94]. Moreover, the use case in which searchable encrpytion is applicable to is ad-hoc (in this case for searching keywords) and may not be applicable to complex application scenarios.

2.5 Remote Attestation

Remote attestation is a reactive security system that provides mechanisms for a client to verify and attest a remote platform. This section presents the necessary background on remote attestation. The basic concepts of trusted computing and trusted boot is presented in Section 2.5.1 and Section 2.5.2, respectively. The cryptographic coprocessor, Trusted Platform Module (TPM), is discussed in detail in Section 2.5.3. This section concludes by presenting several methods of remote attestation such as realtime measurements and dynamic measurements in Section 2.5.4 and Section 2.5.5.

2.5.1 Trusted Computing

To address the cloud server threat model described in Section 2.3, trusted computing was introduced by Trusted Computing Group (TCG) [106]. Trusted Computing refers to the infrastructure that behaves consistently based on the expected behavior of the user. The behaviors are enforced by computer hardware and software through remote attestation. The Trusted Computing Group is a consortium founded by AMD, CISCO, IBM, HP, Microsoft and Wave Systems Corp in 2003 to develop technologies and standardized protocols for trusted computing. TCG

has devised specifications for trusted boot, specifications for a cryptographic co-processor, Trusted Platform Module (TPM) and protocols for remote attestation of the platform. They are discussed in detail in the subsequent sections.

2.5.2 Trusted Boot

The server's state (platform integrity) can be verified only by inspecting the software binaries that are loaded in the server. The list of software binaries that are loaded into the memory when booting (BIOS) to the time of verification will enable the client to accurately assess if the server's state is compromised. The list also referred to as *measurement list*, includes the hash of OS kernel binaries, user binaries, configuration files and all the files that are mapped into the memory by the OS ordered by their time of launch.

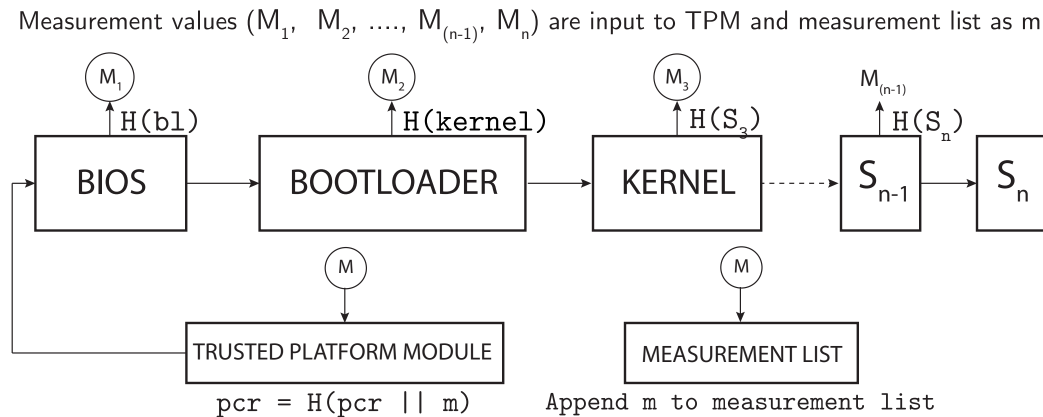

Figure 2.2: Trusted Boot

To create a measurement list, m, each software binary, S_i is responsible for properly measuring and recording the hash measurements, M_{i+1} of binary, S_{i+1}, that S_i loads in the memory (i, $i + 1$ refers to the sequence in which binaries are loaded in the memory) (Figure 2.2).

Hence this forms the *chain of trust*: that is the verifier trusts each software to have properly measured and recorded subsequently launched pieces of software. This leads to a condition where the first program that is loaded to the memory, BIOS (S_0), should be trusted.

However, a BIOS cannot be trusted as the server administrator, who has physical access to the server, can easily overwrite a BIOS (BIOS flashing) of the compromised server through a

BIOS rootkit attack [45]. Therefore, there is a need to verify the BIOS before measuring the entire software stack that is loaded subsequently.

The BIOS is verified by an untamperable hardware *root of trust*, Trusted Platform Module (TPM). On first system boot, the TPM calculates the hash value of the BIOS and records it in its storage before launching the BIOS. The BIOS measures and loads the boot loader and the boot loader measures and loads the operating system and so on, hence forming a chain of trust. Such a boot is trusted boot [50].

2.5.3 Trusted Platform Module (TPM)

To provide an untamperable hardware for *root of trust*, the Trusted Platform Module (TPM), a cryptographic co-processor chip, is developed using the specifications of Trusted Computing Group [106]. The TPM chip is shipped with all modern processors and chipsets [22, 23]. TPM measures the BIOS before loading it into the memory. It also provides the hardware for hashing the BIOS and the registers to store an aggregated hash of all the measured hash values. The operations cannot be tampered with, since all operations are peformed in the TPM hardware.

A TPM contains the following components in its architecture (Figure 2.3):

I/O: The input output controller of trusted platform module manages the information flow over the communication bus of the TPM, by routing messages to appropriate components of the TPM.

Cryptographic Co-Processor: The cryptographic co-processor, implements cryptographic operations within the TPM. This includes asymmetric key generation (RSA), encryption/decryption (RSA), Hashing (SHA-1) and Random Number Generation.

Key generation: This component generates key pairs for the use of RSA algorithm. A TPM can support up to 2048 bit keys.

Hash-based Message Authentication Code (HMAC) Engine: The HMAC engine is responsible for the calculation of HMAC as per RFC 2104. HMAC calculation is used to provide

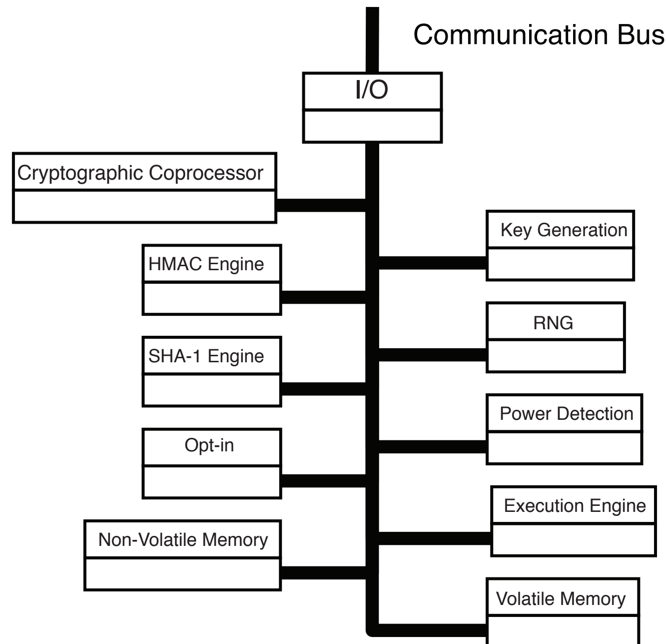

Figure 2.3: TPM Architecture

proof of data created by TPM and for verifying the HMAC of the data arriving to TPM.

Random number generator (RNG): The RNG component generates random bits that are used for nonces, key generation and randomness in signatures.

SHA1 Engine: The SHA1 engine is a message digest engine which uses the Secure Hash Algorithm [33]. This algorithm hashes the data given and produces a 20 byte digest. This forms the basis of the HMAC engine (Hash based message authentication code) which is used for computing digital signatures and creates objects necessary for integrity protection. The hash interfaces are exposed outside the TPM to support measurement during platform boot phases (measurement of BIOS).

Power Detection: This component manages the power states of TPM in conjunction with physical server power states. This component identifies reboots in the platform and notifies TPM of the reset.

Opt-In: This component provides mechanisms to allow the TPM to be turned on/off.

Execution engine: The execution engine runs program code to execute TPM commands that are received from the BIOS and TPM driver of the OS. The TPM commands provides TPM management functions such as starting a TPM, protecting the TPM by a password and also operational functions such as encryption and decryption of data and storing integrity values in its registers.

Non-volatile memory: Non-volatile memory is used to store persistent identity information of the TPM. It stores keys, that are used to authenticate the TPM.

Apart from these components, a TPM has special registers called Platform Configuration Registers that holds the aggregated hash of integrity values of the loaded applications.

Platform configuration registers (PCR): A platform configuration register is a 160-bit storage for the integrity values. A TPM has a minimum of 16 PCR registers. However, the latest TPM ship with 24 registers [106]. A PCR is designed to hold an aggregated hash value, Ag, that can act as the proof of all the measured hash values by a server.

The aggregated hash value at given stage n (value representing the hash of all the softwares loaded till this stage, n) is given by,

$$Ag_n = H(Ag_{n-1} \,||\, H(S_n)) \tag{2.1}$$

Every time a software binary, S_n is loaded in the memory, its hash value, $H(S_n)$ is concatenated with Ag_{n-1}, the aggregated hash of all the softwares up to S_{n-1}. The new aggregated hash Ag_n is given by the hash of the concatenated value, $Ag_{n-1} \,||\, H(S_n)$. This process is referred to as *extend* and is used widely in the architectures that use TPM for remote attestation.

Equation (1) can be further expanded as follows:

$$Ag = H(H(H(\ldots H(0 \,||\, H(Bios)) \,||\ldots) \,||\, H(S_{n-2})) \,||\, H(S_{n-1})) \,||\, H(S_n))$$

The aggregated hash property is not commutative and hence a change in the loading order of software causes a change in hash values. For example, measuring S_i and S_{i+1} is not the same as measuring S_{i+1} and S_i. The other hash property is also onewayness, where it is cryptographically impossible for an attacker to determine the input message (hash values of software) given a PCR value.

The values in PCR are modified only using a *PCR_Extend* command. This command is invoked by the BIOS or the operating system with the hash of the software binary, $H(S_n)$, that is most recently loaded in the memory and a PCR number that will identify the PCR among the 24 PCRs. *PCR_Extend* replaces the value of the PCR indicated by the PCR number with the hash of the current value contained in that PCR and hash of the most recently loaded software binary, $H(S_n)$, concatenated together (Equation 1). Hence the PCR value cannot be overwritten to a known secure hash value by a malicious program, because PCR values are updated only through the *PCR_Extend* operation.

Signed PCR values are retrieved by the operating system in the server using a *TPM_Quote* operation. *TPM_Quote* provides the invoker with the cryptographically signed aggregated hash value with a key that uniquely identifies the TPM. A third party verifier is provided with the measurement list (Section 2.5.2) and the value of *TPM_Quote* operation. The authenticity of a measurement list can be verified by calculating the aggregated hash of the values provided in the list and comparing it with the aggregated hash that is digitally signed by the TPM.

Thus TPM through its PCRs allows platform integrity reporting by creating a nearly unforgeable hash-key summary of the hardware and software configuration [TODO]. This hash-key summary serves as the proof for a third party that the software has not been changed.

2.5.4 Realtime Measurements

The aggregated cryptographic hash allows the client to verify the code identity[2] of the software binary. However, the measured cryptographic hash does not control the behavior of the binary, as it just verifies if the binary is not modified or corrupted before loading it into the memory. When an administrator modifies the part of the loaded program memory with malicious code, the client will not be able to detect it, since the static measurement is made only before loading

[2] Code identity of a binary allows the user to identify if the binary is modified before loading into the memory.

of the binary. This can be countered by reporting realtime hash measurements of the binary to the client. The realtime hash measurements are performed by calculating the hash over the memory in which the program is loaded rather than the binary image in the disk [43, 121, 26]. We discuss one such architecture in detail, HIMA [26], in Section 4.2.2.

2.5.5 Dynamic Root of Trust for Measurement

In a typical GNU/Linux operating system, reporting the integrity measurement will involve measuring the hash values of more than 5000 system and configuration files [34]. Establishing a Trusted Code Base (TCB) among the 5000 files can be very challenging. The different versions of code for different platforms and the consistent updates of the code only make integrity measurement management daunting. To address these issues, AMD and Intel extended the x86 instruction set to support *dynamic root of trust for measurement* (DRTM). A DRTM operation provides a temporary protected sandbox for a specified sensitive code to execute, where it cannot be accessed by other executables that are loaded in the memory. This makes a DRTM operation secure irrespective of the software binaries that are loaded in the real memory.

Invoking DRTM operation, resets the CPU and memory controller to a secure state. In this state, most of the hardware and software interrupts are disabled and the DMA protections for the region of memory in which the sensitive code needs to be executed is enabled. The CPU measures an executable sensitive code for the DRTM operation and *extends* (Section 2.5.3) it into the TPM. After the measurement, the sensitive code is executed in that hardware protected environment. The CPU is resumed to the state it was in before the DRTM operation, once the execution is terminated.

AMD DRTM is called Secure Virtual Machine (SVM) mode [23] and Intel's DRTM is called Trusted eXecution Technology (TXT) [22]. Both AMD and Intel are shipping processors with these capabilities as of 2010.

The Flicker project [76] used the Intel TXT support to implement on demand DRTM based secure execution. It allowed the currently executing module to pause to execute a measured sensitive piece of code to run. Once completed, the previous environment was resumed to run with full access. The context switches in Flicker was found to be costly. Trustvisor [77]

improved on Flicker project by implementing a hypervisor and a VM to provide DRTM functionality to applications thereby minimizing context switch overhead.

Chapter 3

Chaavi: Webmail with Searchable Encryption

The last chapter (Chapter 2) presented the privacy, security and trust challenges in cloud computing. Section 2.4 introduced Privacy Enhancing Technologies (PET) as one of the solutions to counter these challenges. This chapter proposes a webmail architecture using PET technologies, in which users can retain their mail in the servers of their service providers in a cloud without compromising functionality (searchability of mails) or privacy. We benchmark our system and provide the results showing that it is feasible to architect a secure solution for webmail systems using a PET.

This chapter also details the limitations of our proposed webmail system and the specificity of the webmail solution in addressing the general issue of privacy, security and trust in cloud computing. This chapter concludes with the motivation for remote attestation in cloud computing.

Section 2.3 presented the threat model for cloud computing. This chapter addresses the threats arising from the cloud adversary and a remote adversary. The proposed work in this chapter addresses the class of cloud services that stores data and provide searching as its primary functionality. This includes services such as webmail, collaborative document authoring (Google documents) and private blogs. The example used throughout this chapter is our webmail system, Chaavi.

The rest of the chapter is organized as follows. A motivating example of webmail services

is described in Section 3.1. Section 3.2 reviews background and related work for searching on encrypted data. Section 3.3 presents the architecture of Chaavi system. The implementation details are discussed in Section 3.4. Section 3.5 presents the experiments conducted to study the system and we conclude by stating the limitations of our system in Section 3.6. Section 3.7 concludes the chapter with discussion on the need of remote attestation.

3.1 Motivating Example: Webmail Services

Webmail services offer user convenience. A username and password are not tied to any particular equipment or location. Webmail services primarily offer the following functionality:

1. Mail Storage
2. Organization of mail
3. Keyword Searching

For (1) and (2), the service provider need not know the exact content of the mail. However, for performing a plaintext keyword search on email the cloud provider needs to know the content of the mail, so that the provider's infrastructure can be used to index the mail content, which can in turn be used for the search process. The use of webmail services, has the following shortcomings:

1. The need to trust the service provider (e.g., Google, Yahoo, or Microsoft) as the mail is stored as plain-text in the service providers' servers (or using single key encryption). The mail is then prone to insider attacks (anyone with the access control will be able to read the mails).
2. There is an assumption that the provider is honest and the security level is sufficient.
3. When the mail is transferred from one domain to another, it is transmitted through SMTP [93]. SMTP as a protocol does not support encryption. Technologies like Transport Layer Security [41] are used to transfer mail to other domains. However, the data is still protected only up to the layer at which it reaches the target mail server. Once it reaches the target mail server, the mail is again prone to insider attacks in the new domain.

To address such problems, various client encryption systems, such as Pretty Good Privacy (PGP) [122], have been developed. However, encryption using PGP make the mail non-searchable in the web server.

3.2 Background

In this section, we review the basic elements common to webmail infrastructures. We also present an introduction to PGP and searchable encryption.

3.2.1 Mail Architecture

The webmail infrastructure is responsible for end-to-end delivery of email. Figure 3.1 presents architectural components and protocols typically used to support webmail applications.

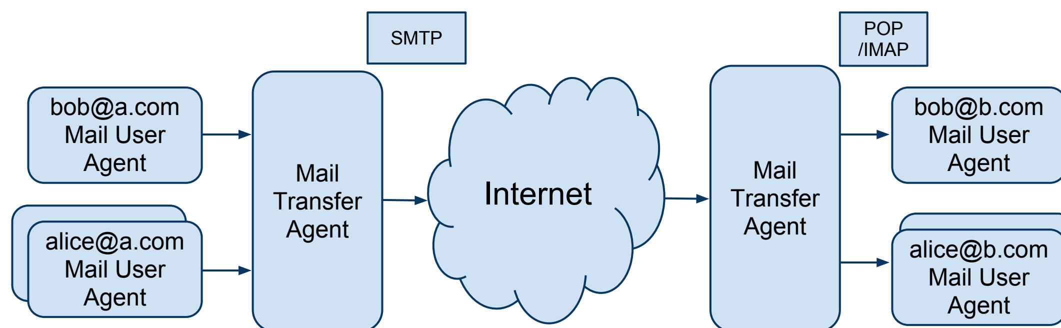

Figure 3.1: Email Architecture

Components

This subsection describes the architectural components.

- **Mail User Agent:** The Mail User Agent (MUA) is used to manage a user's email. It acts on behalf of the user to send and receive mail from the Mail Transfer Agent (MTA). Popular MUAs include Microsoft Outlook, Mozilla Thunderbird, Apple Mail. In a webmail system, the MUA runs in the server and the pages are rendered as HTML pages for the browser.

- **Mail Transfer Agent:** The Mail Transfer Agent (MTA) transfers messages from one server to another. It receives email either from another MTA or MUA. The transmission of email follows standardized protocols for message transfers.

- **Protocols:** The following protocols are commonly used in a mail architecture.

  - Simple Mail Transfer Protocol (SMTP): SMTP refers to the standard for the transfer of messages from one server to another. It is used by MUA to relay mail through the MTA and it is also used by the MTA to send and receive mail between other MTAs. SMTP as a standard does not encrypt messages (unless Transport Layer Security encryption is used).
  - Post Office Protocol (POP) / Internet Mail Access Protocol (IMAP) POP/IMAP are email retrieval protocols that specify standards for downloading messages from the MTA for MUA. Examples of use is found with support for POP version 3 and IMAP as provided by Gmail.

Privacy Threats

In webmail systems, there is a server for webmail introduced into the standard mail system (Figure 3.1). It acts as the Mail User Agent for a number of users and manages email for all the users. The MUA, unlike the standard model (Figure 3.1), is centralized at the server. The webmail server uses POP/IMAP to download messages from MTA.

There are several privacy concerns with respect to email systems. If the connection to the webmail server is not secured using Hypertext Transfer Protocol Secure (HTTPS) all the data between a user's browser and the server will be in plain text. SMTP, unless used with the Transport Layer Security (TLS) layer, is insecure. Even if the TLS layer is used, the mail will still be accessible by the owner of the MTA, through which the mail is routed. This is because TLS is designed to protect data in an insecure network (like Internet) and not from the communicating parties. Some of the security threats involved in email systems are identified by Kangas et al. [64], and Kaufman et al. [66]. These are listed below.

- **Eavesdropping:** When email is unencrypted, potential hackers who have access to network packets flowing through the network will be able to read the email sent. This can

be achieved by enabling the promiscuous mode on ethernet cards.

- **Identity Theft:** If the user's username and password is obtained, then hackers have full access to all the email content. Password information can be obtained by eavesdropping on the network.

- **Invasion of Privacy:** The recipient of the mail is able to get more information from the email header information than what the sender intends to reveal. For example, the header will reveal the sender's SMTP IP address and subject of the email sent.

- **Message Modification:** Anyone who has administrator access to the webmail server can modify the messages stored in the server. It is not always possible for a recipient to determine that email has been tampered with.

- **False Messages:** It is relatively easy to create false messages and send it as if it is from any person (as evidenced by spam).

- **Message Replay:** Akin to message modification, the message created by a user can be saved and sent again and again.

- **Unprotected Backups:** Messages are stored in plain-text on SMTP servers, and backups will also contain complete copies of the messages. Even when the user deletes a message from the server, the backup will still hold the content.

- **Repudiation:** As email messages can be forged (for example see your spam box), there is no way of validating that the email has been in-fact sent by a particular person. This has serious implications in business communications, electronic commerce.

3.2.2 Pretty Good Privacy

PGP was created by Zimmermann et al. [122], in 1991 to address the security issues with email. PGP encryption uses a serial combination of hashing, data compression, symmetric-key cryptography, and public-key cryptography. Each public-key is bound to an email address. It serves as the verification mechanism for the origin of the email. As the email is encrypted

using the private key of the user and the encrypted version is sent into the network, it addresses many security issues of the email infrastructure. For webmail systems, software such FireGPG [6] provide a browser extension that implements PGP. As PGP support enhances the security of the email system by encrypting the mails, the mail becomes unreadable by the server. Hence the server cannot perform keyword searches on the mail.

3.2.3 Searchable Encrypted Data

Searchable encrypted data enables the client to protect the privacy of their mails without compromising on the keyword search functionality on the mail. Public Key Encryption with Keyword Search (PEKS) [36] is one of the seminal works in the area of making encrypted data searchable. The authors of PEKS propose to encrypt the message (e-mail message in our case) using the Public-Private key infrastructure. Along with this cipher text a Public-Key Encryption with Keyword Search (PEKS) of each keyword (the words that make up the message) is appended to the final message. To send a message M with keywords W_1, W_2, ... W_m the following information is transmitted to the storage server, that performs the PEKS search:

$$E_{A_{pub}}(M) \,\|PEKS(A_{pub}, W_1) \,\|... \,\|PEKS(A_{pub}, W_m)$$

where A_{pub} is the public key of the user, $E_{A_{pub}}(M)$ is the encrypted message, $PEKS$ is the function that encrypts the keywords using A_{pub}. To test whether a word W is a part of the message, a user supplies $PEKS(A_{pub}, W)$ along with a trapdoor function T_w to the server. As the function $PEKS$ is probabilistic the value of the $PEKS(A_{pub}, W)$ changes randomly every time and a straightforward comparison of encrypted keywords will fail. The trapdoor function enables the user to test if two different cipher texts are formed using the same input keyword (W) to $PEKS$ function. Thus T_w can effectively test whether $PEKS(A_{pub}, W) = C'$, C' being the encrypted keyword that is stored in the server. If $PEKS(A_{pub}, W) \neq C'$ the server learns nothing more about W or W' that was used to create C' using $PEKS(A_{pub}, W')$.

Public Key Encryption with Keyword Search Revisited [29] identifies some of the issues with the original PEKS and proposed a provably secure algorithm. The authors argue that if in PEKS the server starts learning the trapdoor then there can be a categorization of mail formed just based on the learned trapdoor information. The trapdoor information is the extra

information sent to the server along with the encrypted keyword for the server to test for the existence of a keyword.

The authors also identify that in PEKS there is an assumption that the communication channel between the sender and the server is secure. To enable secure communication through insecure channels the authors propose a Secure Channel Free Public Key Encryption with Keyword Search (SCF-PEKS), that uses a server's public-private key pair for communication.

In our work we achieve privacy for a webmail system by using a modification of the PEKS [71] scheme. The email messages are encrypted using the traditional public-private key encryption. The keywords from the email messages are extracted and each keyword is encrypted using a deterministic symmetric encryption: AES without random Intialization Vector (IV) [40]. The deterministic nature of the encryption increases the chances of chosen plaintext attacks over our encryption scheme. However, our focus is to analyze how these encryption schemes can be engineered in a real working environment and study their performance implications.

3.3 Architecture

This section describes the various components of Chaavi. Figure 3.2 gives the overall architecture of the system.

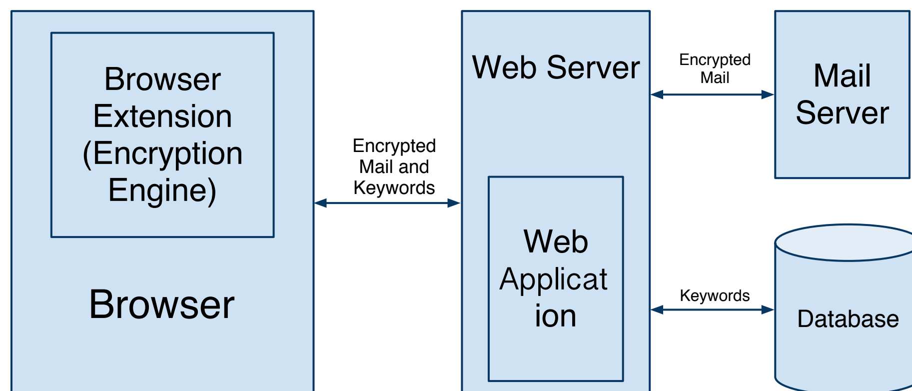

Figure 3.2: Chaavi - Architecture

3.3.1 Browser

The browser is responsible for rendering the pages created by the web application. Its default behavior can be modified or enhanced by using extensions or plugins in the browsers. Modern browsers such as Mozilla Firefox, Google Chrome provide functionality to write extensions/plugins and install the extensions locally.

3.3.2 Browser Extension

A browser extension is used in Chaavi to encrypt the secure message sent to the server. It is also used to decrypt the messages that are sent from the server. Additionally it has key generation and key management functionality. The extension is composed of the following modules.

Public-Private Key Generation: As stated earlier, Chaavi uses a public/private key model for securely communicating messages. In a public/private key model, a public-private key pair is generated when the system is initiated for the first time, for a particular user. The messages are encrypted by a symmetric key. The symmetric key is encrypted by the public key and can be decrypted only by use of a private key. The encrypted message is appended with the encrypted symmetric key and is transmitted as a message. The public key as the name implies is shared in a public forum.

Keyword Encryption Key Generation: Public-Private key pair is used for secure message communication. A symmetric key is also generated to encrypt the individual keywords present in the mail. A symmetric algorithm (unlike the Public-Private key) is used here as the keywords need not be decrypted by anyone else other than the sender of the message.

Key Management: Key management is performed using a graphical user interface (GUI). The GUI enables the user to add or delete the public keys of the recipients with whom the user wants to communicate through mails.

Encryption: The functionality of the encryption module is to encrypt the messages that are sent to the server from the browser. It also extracts and encrypts the individual keywords in the message. The encryption module is triggered from the web application when the user submits a mail to send it to the web server. This module encrypts the message using the recipients's public key and the keywords with the keyword encryption key.

Decryption: When an encrypted message is sent from the server to the browser, the decryption module decrypts the messages using the user's private key that is generated during system initialization.

3.3.3 Web Application

The webmail application provides graphical user interfaces for the users to read, send and search messages. It comprises of both email server and client-side (browser) functionality.

When a user sends a message from the web application (Figure 3.3), the Encryption module encrypts the message and extracts and encrypts the keywords. The web application sends the encrypted message and keywords to the web server. On receiving the encrypted message and the keywords, at the server-side the application saves the encrypted message alongside the encrypted keywords in a database for future retrieval. The application then transfers the mail to the Mail Server (SMTP server) for the mail to be be delivered to recipient.

When the user wants to search for a particular keyword in their inbox, the encrypted keyword is sent to the server-side. The web application then searches for the mails corresponding to that particular encrypted word and then sends the encrypted mails back to the user.

3.3.4 Database

The mail storage and organizational functionality is already handled by the web application provided by Squirrelmail. One custom table, *search* is added to the database, which stores the $< message_id, encrypted_keyword >$ pair. This database is looked up when the user performs a keyword search.

3.3.5 Mail Server

The mail server sends and receives email communicated to it through the Internet. The mail server functionality is not modified by our system. The web application communicates with the mail server to send and receive messages.

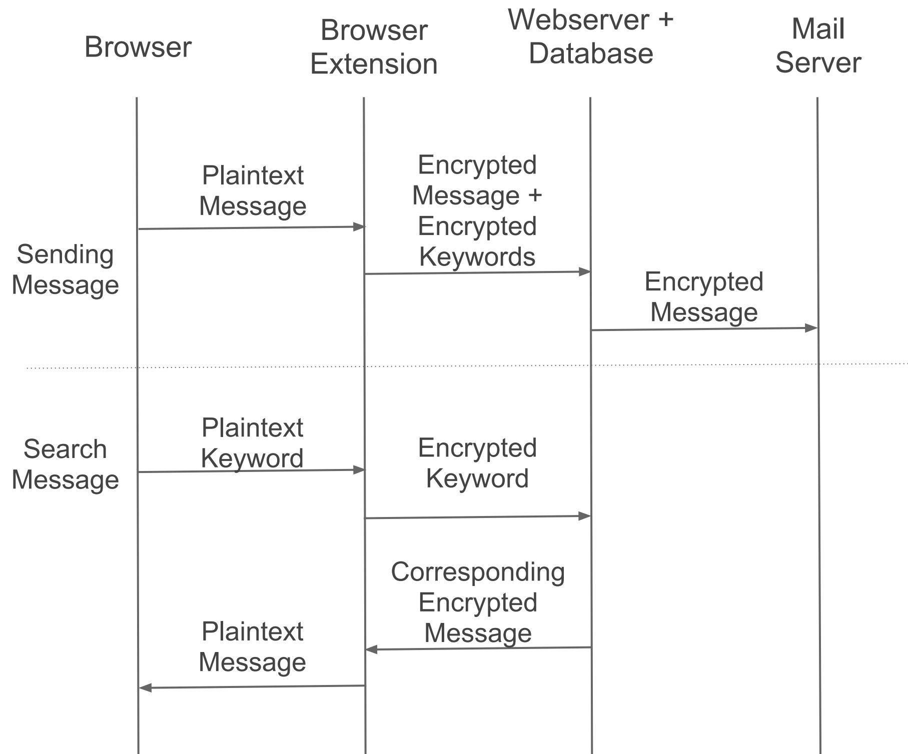

Figure 3.3: Sending and Searching for a Message

3.4 Implementation

The following software is used to implement the different components in the system:

- Browser - Google Chrome
- Browser Extension - Google Chrome using Javascript
- RSA encryption/decryption library from hanewin.net [8]
- AES encryption library [7]
- Web Application - Squirrelmail over PHP and MySQL
- Mail Server - Using the POP3 interface of the *csd.uwo.ca* mail server

The implementation details of individual modules of the system are detailed below.

3.4.1 Browser Extension

Public-Private Key Generation: The RSA algorithm [95] is used for the creation of keys. The key requires two large prime numbers as the input along with a random seed. All of these inputs are created by the extension randomly and provided as input for key generation. The keys are then stored locally along with the user name, for future retrieval in the local browser database. The key generation is implemented using the RSA libraries available from hanewin.net [8].

AES Key Generation: The symmetric AES key algorithm is used to encrypt the message and individual keywords present in the mail. The AES key generation algorithm takes as input a random seed, which is provided by requesting that the user move the mouse over the browser window. That generates some random coordinates, which is then used to generate the key.

AES is a natural choice for the symmetric key algorithm as it has been analyzed extensively and used worldwide [115]. However, unlike PEKS [29], AES algorithm does not support trapdoor and hence it is susceptible to chosen plaintext attacks (The attacker has the capability to choose arbitrary plaintext and the corresponding cipher texts). Moreover the encryption of the keywords under AES negates the possibility of performing range searches (e.g., $10 < b < 20$) or similarity searches (name staring with ‘ka’).

Key Management: The GUI for key management (Figure 3.4) is developed using the options functionality provided by the Chrome extension framework. It is used to insert the public keys of the recipients with whom the user wants to communicate. The private key of the user cannot be managed using this interface (the system automatically generates it when the user logs in for the first time). The keys are stored in the local storage database provided by HTML5. The local storage enables key-value storage locally managed by browser.

Encryption: The user is provided with a HTML form from the web application, which contains input fields to enter the recipient email address, subject and the contents of the mail. The form submission event (*onsubmit* event) is associated with a custom submit event handler,

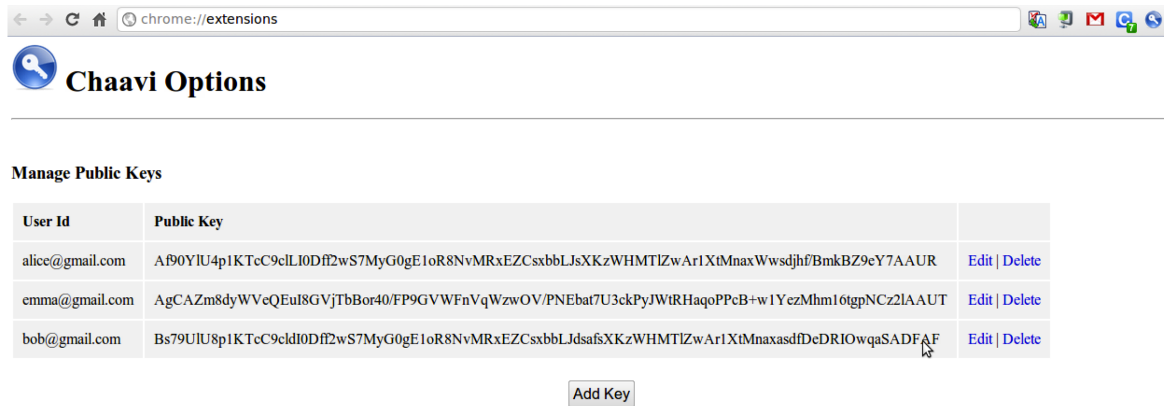

Figure 3.4: Key Management

which is hooked to the encryption module. The encryption module encrypts the contents of the mail using the user's public key and replaces the value in the field (contents of the mail) with the encrypted message. Along with this, the keywords in the message are extracted by the keyword extraction function and each keyword is encrypted using the AES key and stored in an object. This object is serialized in JSON (Javascript Object Notation) and sent to the server along with the encrypted message.

Decryption: When an encrypted message is sent from the server to the browser the server adds the attribute value $post-deencrypt$ to attribute $class$. The extension identifies these messages and decrypts the messages using the private key of the user. This decrypted message replaces the original encrypted message in the html page so that the user can see the message in the encrypted mail.

3.4.2 Web Application:

An open source web application (Squirrelmail) is identified and it is modified for our application. Squirrelmail is responsible for storage and organization of the mails. Our custom module is developed in PHP and added to Squirrelmail to save the encrypted messages alongside the encrypted keywords and for the retrieval of the messages based on the given encrypted keyword.

3.5 Experiments

The performance of algorithms used in Chaavi (Privacy Preserving Web Mail with Keyword Searches) is studied in terms of space and time consumed by the algorithm in the local client system. Even though the performance of the encryption algorithms has been studied before, we focus on the performance of our system. The results presented in this section are intended to provide some insight on the overhead provided by the algorithms in a browser based extension environment. Since encryption and decryption is performed in the client browser system, the encryption and decryption is independent of the number of users currently using the system. Hence, we focus on the performance of the encryption algorithms for a browser-based extension environment.

All the experiments are executed in a Pentium IV Core 2 Duo processor using Google Chrome 5.0.375.99 beta.

3.5.1 Time Complexity

The following algorithms are studied with respect to the execution time.

- Key Generation
- Encryption and Decryption (RSA Algorithm)
- Keyword Encryption (AES Algorithm)

Key Generation

Key generation is expensive since it involves finding two large random prime numbers and finding a product of the prime numbers based on the given random seed. The length of keys (as measured by bits) can be of sizes: 128, 256, 512, 1024. The higher the number of bits used, the more difficult it is to break the key (According to Schneier et al. [100], for breaking AES with key size greater than or equal to 256-bit through brute force will require fundamental breakthroughs in physics and understanding of universe). However, generating larger keys is time consuming. We present the average time taken for key generation for different bit sizes in Figure 3.5.

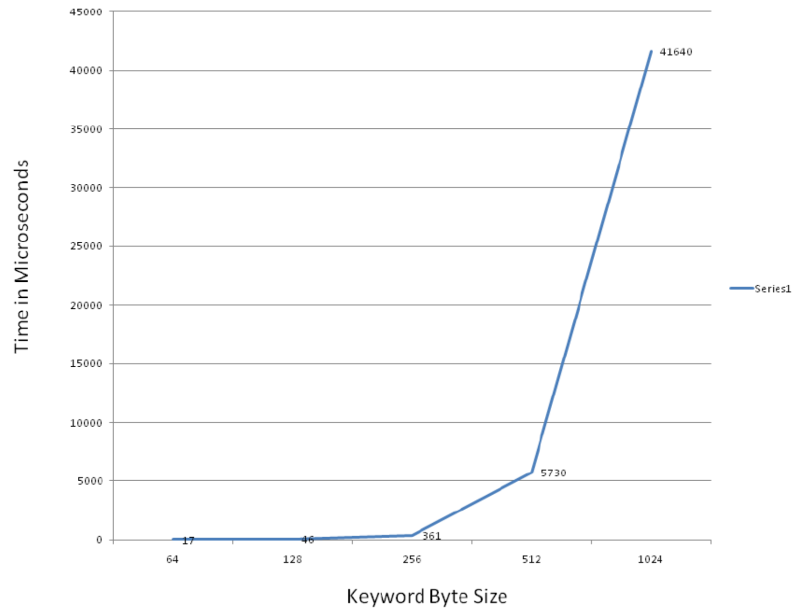

Figure 3.5: Key Generation

As can be seen the keyword bit size increases the creation time exponentially. The 1024 bit key generation takes around 41 seconds. However, as this is a one time activity (when the user sets up the system) the usability and inconvenience is minimal.

Encryption and Decryption

When the user wants to send an email the encryption module is executed each time, and the decryption module is activated when the user wants to read an email. This is a frequent activity and therefore more computation time spent on these modules will impact usability. The encryption and decryption algorithm is run over random data (which represents an email message) set using the Javascript library in Chrome browser. As described in Section 3.4.1, the encryption and decryption has two steps:

1. Encryption/decryption of the symmetric key using private key.
2. Encryption/decryption of message using the symmetric key.

The performance of RSA algorithm is studied for encryption/decryption of symmetric key here in a browser environment. Five trials of experiments show at an average a 512 bit key

takes 48 microseconds for encryption and 53 microseconds for decryption, respectively.

The following are the results for the encryption and decryption of a message using AES encryption (Figure 3.6).

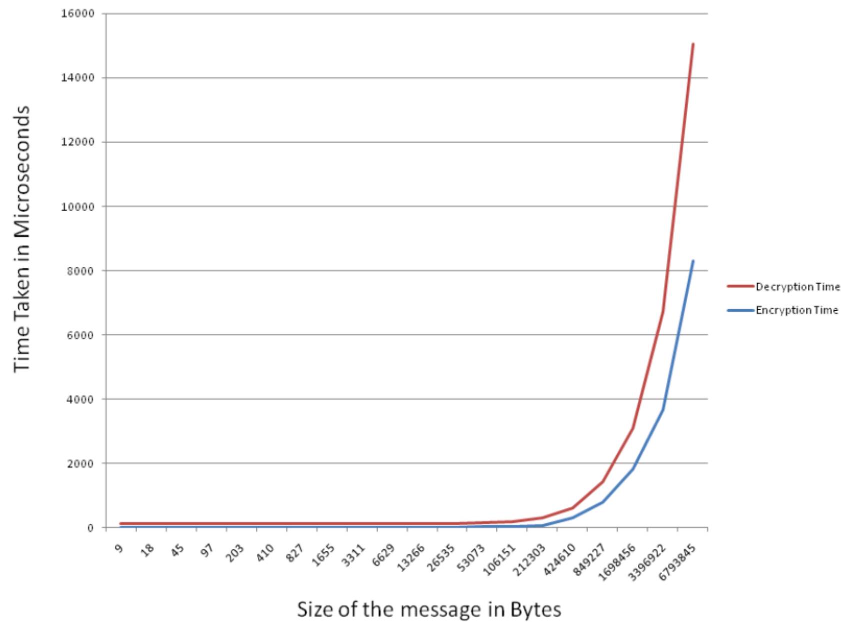

Figure 3.6: Encryption and Decryption

It can be seen that at a relatively larger message size, around 212 KB, the time taken for encryption and decryption is less than 2 seconds. However as the message size increases in the order of megabytes, the time is around 16 seconds. A 67 MB message takes around 16 seconds to encrypt and 9 seconds to decrypt, which is still acceptable for sending such a large message. Moreover, most webmail systems have a limit of 10 MB on message sizes.

Keyword Encryption

In this phase the performance of AES algorithm is studied (Figure 3.7). Each word from the message is extracted and is encrypted using the AES algorithm. There is no decryption phase here, as the encrypted words are checked against each other.

It can been seen that there is a linear relationship between the message size and time taken for encrypting keywords. It has to be also noted that when there are duplicate words the encryption is not done twice. However, in these experiments each word was generated at random

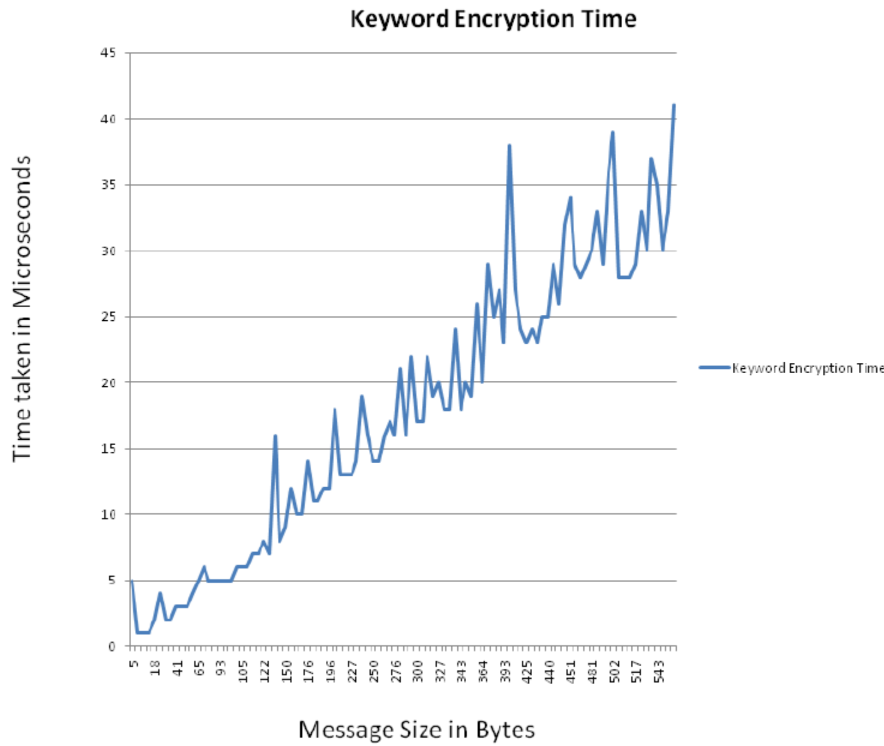

Figure 3.7: Keyword Encryption Time

with a random size (with maximum as 25 bytes). The probability of the same word repeating is very low for this case.

3.5.2 Space Complexity

In our study of the space complexity, we were interested in the following:

1. Increase in size of the keyword index

2. Increase in the size of the final mail

Impact of increase in size on the keyword index

The AES algorithm is executed over the generated keywords and the impact of the size of the encrypted keywords on execution time is examined. The results are in line with AES algorithm space complexity where there is no increase in the size of the message.

Impact of increase on Final Message size

Here we study the total increase in the email size. The email that is sent to the server of the recipient will be in the concatenation of the message encrypted by symmetric encryption and and key encrypted by asymmetric encryption. Any increase in size, will increase the overall network traffic.

As noticed with the keyword index, there was no change in the encrypted message size. However there is a very negligible constant increase in the size of 512 bit (64 byte) key to 524 bytes as it uses asymmetric encryption.

3.6 Limitations of our implementation

We proposed a privacy preserving architecture for our webmail system, that enables secure communication of messages using a public/private key model and privacy preserving keyword search functionality using AES key encryption algorithm.

Our approach requires every client to install an extension to their browser and the cloud computing provider to modify their webmail application to support encrypted keyword search. Even though technically this is a possible solution, economically a cloud provider might not prefer this approach. Most of the business models in web applications are built around the contextual advertising model, where the cloud provider relies on the user's data to deliver the relevant advertisements to the user. In our case as the data is encrypted in the server, the cloud provider will not have access to the user's data. Work such as Toubiana et al. [107], try to address this problem by offloading the keyword extraction in contextual advertising to the client browser. These approaches [107] need to be modified for our architecture so that our system remains economically viable.

Unlike in PEKS [29], our system does not use a probabilistic trapdoor function. The trapdoor is implemented by using deterministic synchronous encryption. This makes our system susceptible to chosen plaintext attacks. If a recipient of a mail is also a potential attacker, the recipient can eavesdrop the encrypted keyword information sent from the sender to the server, and make a guess on what keyword represents the encrypted cipher by analyzing a number of mails sent to the recipient (attacker) from the same sender. However, our contribution is the

proposal of the framework. The encryption algorithms used can be modified to utilize more secure alternatives in our architecture.

Our system makes an assumption that the browser and the browser extension framework is trustworthy. We believe it is a fair assumption, as the user can control and monitor the browser activity and any aberration of browser functionality can be detected by the user (at-least theoretically).

We have also not implemented the functionality to add the incoming messages to the encrypted search database. Future work should address this. Future work also involves detailed study on the strength of the encryption, support to range and similarity searches, improvements to the algorithms used whilst maintaining performance.

3.7 Conclusion

We observe from our results that the searchable encryption is a feasible way of preserving privacy of data in the cloud without compromising the functionality. However, the solution presented in this chapter addresses only searching, which is one of the many possible scenarios in a cloud computing environment.

Consider an e-commerce SaaS cloud such as Amazon.com. For authorizing a client's transactions and sending the orders to the cloud client, the cloud server may have to know the personal details of the client, including their credit card number, name and address. Any kind of encryption of this data will render the data unusable for the server. To secure this information, the cloud clients need to have guarantees that the cloud server will not use the information for any malicious purpose.

Similar to the e-commerce example, the cloud clients may have complex workflow and scenarios requirements that need to be executed in a privacy preserving way in the cloud server. The solutions offered by PET do not address these requirements of cloud clients.

Therefore, in many such scenarios, the clients are forced to *trust* the cloud provider with the data and hope that the provider will not breach that trust. Clients rely on the pressures introduced by the security systems (Chapter 2) to induce the client for cooperation and compliance. The rest of the thesis will heavily focus on one such security system, remote attestation.

Chapter 4

A notification based remote attestation framework for IaaS

Remote attestation provides mechanisms for the cloud clients to remotely verify and attest the cloud server. In the previous chapter (Chapter 3) we concluded that cryptographic methodologies provide privacy preserving solution for ad-hoc scenarios such as keyword searches. But, remote attestation can be used to verify the server's state independent of the scenarios as long as the application and the platform on which the application is hosted on known in advance to the verifier. The verifier can either be the client or a third party who is entrusted with the verification of the platform. Chapter 2 extensively focused on the background of remote attestation.

In this chapter we present the state of the art in remote attestation infrastructures in the cloud and identify two major issues: complexity in the management of software measurement values by the verifier and the client's need to add verification workflow in its transactions. To address these issues, we propose a notification based remote attestation framework for an Infrastructure as a Service (IaaS) that supports the verification of domain controllers and the virtual machines by a trusted third party (TTP). The TTP allows clients to register their Virtual Machines (VMs) and their corresponding IaaS controllers for attestation. The software state of the VMs and their domain controllers are verified continuously by TTP in predefined intervals through polling. Thus after delegating the verification to a trusted TTP, the client is concerned only with the managing of the application in the VMs. In the event of any malware attack in the

IaaS domain controller or VM, the TTP will notify the client and the client can take necessary actions to counter the attack.

This chapter is organized as the following. Section 4.1 introduces the state of the art in remote attestation infrastructure. Section 4.2 focuses on the virtual machine (VM) based infrastructures in the literature. Gap analysis and our contribution to the literature is discussed in Section 4.3. The proposed architecture and its components are described in Section 4.4. Several scenarios in the proposed architecture are presented in Section 4.5. Section 4.6 discusses the steps adopted for establishing a minimal trusted computing base. The implementation details are presented in 4.7. Section 4.8 describes the experiments and the results and this chapter concludes with future work in Section 4.10.

4.1 Related Work - Remote Attestation Infrastructure

This section describes the infrastructures built using technologies introduced in Chapter 2. Mechanisms of collecting and reporting integrity values are discussed in Integrity Measurement Architecture (IMA) [98]. PrivacyCA [91] presents a framework for verifying if the measurements emerge from a valid TPM.

4.1.1 Integrity Measurement Architecture (IMA)

Sailer et al. [98] proposed the Integrity Measurement Architecture (IMA) as an implementation of TCG standards in GNU\Linux. The IMA is designed to collect the integrity measurements of all the files before each file is loaded into the memory and executed by the operating system and then store integrity measurements in a secure location. Figure 4.1 presents the architecture of IMA.

The IMA architecture is hosted on the server that needs to be remotely attested. The IMA has the following components: Challenger, Attestation Service, Measurement Agents, Measurement List and TPM. The hash of the BIOS (trusted BIOS measurements), along with the hash values of all the files are collected and stored in the measurement list by the measurement agents. To secure the measurement list against tampering, the aggregated hash associated with

the loading of the last software binary is stored in one of the Platform Configuration Registers (PCR).

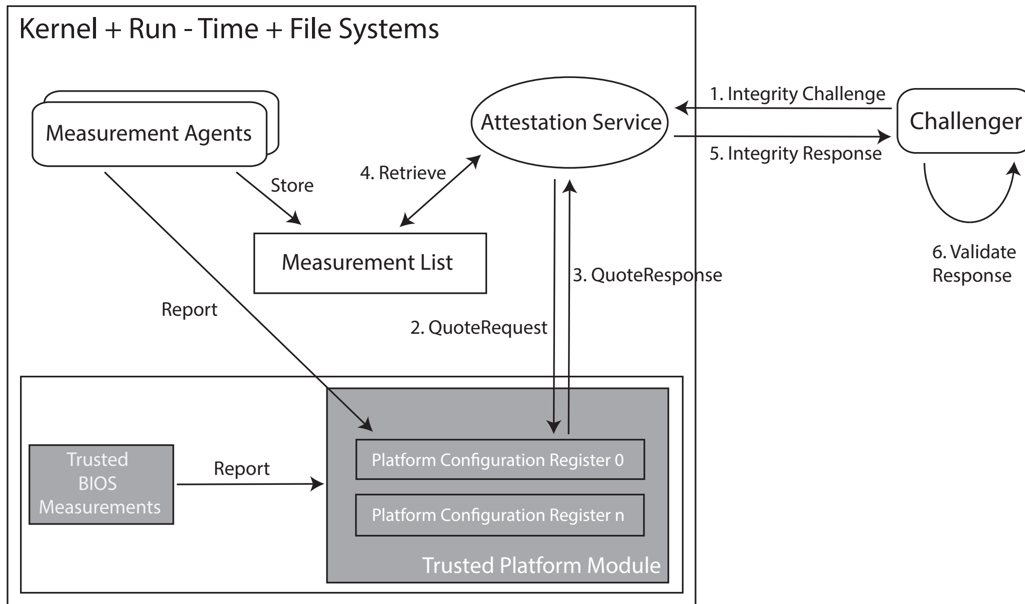

Figure 4.1: Integrity Measurement Architecture

When a client (challenger) wants to verify the state of the server, it requests the integrity values from the server by invoking the attestation service hosted in the server through the *integrity challenge* (Figure 4.1). The attestation service retrieves the measurement list along with the aggregated hash from the PCR and sends it back to the challenger. The aggregated hash is cryptographically signed by the TPM and is verifiable by the client. The client validates the integrity values against its local datastore of known secure values and then makes a decision on whether or not the server's state is "healthy", i.e, not corrupted by any malicious software.

Sailer et al. [98] does not explain how the known secure values of software binaries are collected. However, the TCG specification for remote attestation [14] presents the Remote Integrity Measurement Manifest (RIMM) for storing the secure hash values. The authors of the TCG specification state that a software is audited and tested for a period of time before adding it into the RIMM repository.

When the server executes an unknown software binary, the hash of the binary is recorded in the measurement list. During the integrity challenge, the server's state will be classified as

unsecure because of the unknown binary's hash value in the measurement list as there will be no record of it in the verifier's local repository.

IMA Message Format: If a TPM is in the server that needs to be verified the measurements of the BIOS are written into the *bios_measurements* file in the */sys/kernel/security/tpm0* directory of the server's filesystem. The measurements of individual files are written into a *binary_runtime_measurements* file in the */sys/kernel/security/ima* directory. The GNU/Linux kernel mounts the security file system in */sys/kernel*, where only the kernel can modify the data in the directory.

The IMA can be configured using *ima-policy* in which the type of files that needs to be measured are specified.

```
10 23de2aff5055a8c8e22f7bfa9c5991cda6dae0a8 ima 24397111529ff7abb0b04c55eaa638c983c29e13 boot_aggregate
10 fe6550ffc0f455d0699200823b18c3d643fb3c70 ima f6585d3d8fcec483efcd3c4741f81a94f3977030 /etc/init.d/mysql
10 bd2b6cdca6bc6f1369675a20c54991e4859382de ima 0a2e3beed68a10ea999c50eb087cd7514ba59646 hotplugpath.sh
10 64f392df673f0f164e476dd98c232ab9635af59f ima 2d7ce8c3a4d2bc68ad72ae954c09b3416f109110 init-functions
10 079e2b094aa849506adda43666562ea93a7b1a2b ima 89a259594978ae70388e35746ebff1b1b6b8e7d6 /bin/run-parts
10 987341a885f00fae89f0d57665d6fbb16c02a14f ima 254b19e8c5257c07df740e7a2b76bf6b06fea05b ld-2.13.so
10 2186f7d904a0197dcd4809dfca1553f15240dc54 ima 2a0beab00650843e54b97a7353104aa91130cf5d /usr/sbin/atd
10 f4b36068201709f27c3a58a3a7606979d0c5868c ima 2115209e18d9659ddc5560d29a749fb88a3776ab ld.so.cache
10 3553ded0d2ed8025ba8b5ad827e3c9fde5b9df8d ima 29d1df13fb8104b1e37bb09ba0895f9d5c682147 xend
10 580d139b73d9bc99731f006d977294125fe48175 ima 67656486e37794a1d878197c236e8e801ed2ff38 /bin/mkdir
```

Figure 4.2: ASCII Runtime Measurements

IMA supports several templates for representing measurements. The default template type, `ima`, is used for the ascii measurements log in Figure 4.2. The first column in the `ima` template (Figure 4.2) represents the PCR in which the hash values of the file are *extended*, in this case PCR 10. The second column represents the hash value of the file data. The third column indicates the template type of the measurement, `ima`. The fourth column contains the hash of the file meta data (UID/GID labels of the file along with Linux Security Module[1] (LSM) labels). The final column is the filename of the file that is being measured. The filename is restricted to

[1]LSM labels are xattr attributes of a file, used by SELinux for implementing mandatory access controls for the GNU/Linux filesystem.

255 characters. This IMA log file can be sent to a verifier to check if the applications that are hosted in the host machine are valid.

IMA also supports other template types: `ima-ng` and `ima-sig`. The `ima-ng` template allows the measurements to represent the filename without restricting it to 255 characters and the `ima-sig` template presents the file signature along with the file data hash.

IMA is supported from Linux 2.6 onwards.

4.1.2 Privacy Certificate Authority (CA)

The remote attestation infrastructure as implemented in IMA requires the server (attestant) to send a detailed description of its system state to the client (attester). The attestant also provides the attester with the Attestation Identification Key (AIK)[2] certificate which guarantees that the AIK is owned and secured by a TPM that enforces TCG specified policies. The TCG specifications [106] define the PrivacyCA service to issue these AIK certificates to the attestant. The certification process is presented in Figure 4.3.

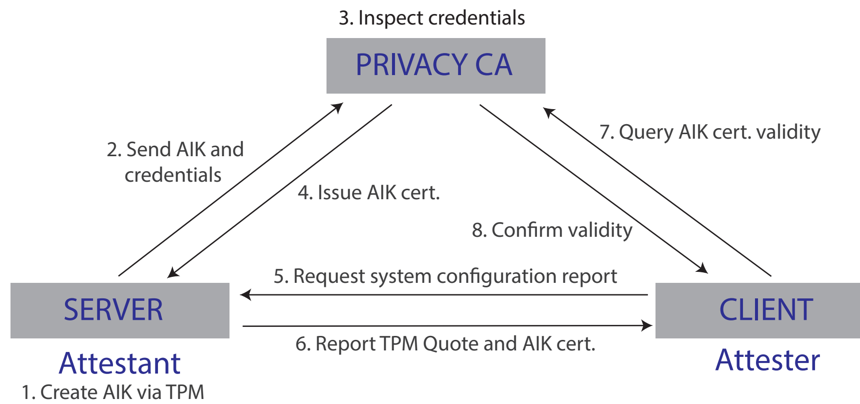

Figure 4.3: Privacy CA

The server initially registers its TPM's AIK with PrivacyCA and receives the AIK certificate. The AIK certificate is encrypted with the TPM's public key by the TPM. Only the TPM can decrypt the AIK certificate. When the client requests the server's current system configuration, the server sends the AIK certificate along with the aggregated hash value, TPM_Quote

[2]Attestation Identification Keys are special keys produced by the TPM for the purpose of securing the integrity values while transmitting to the clients.

(Section 2.5.3). The client sends the AIK certificate for verification to the PrivacyCA and after verification uses the TPM Quote.

Piker et al. [91] presented the following set of guidelines to implement a secure Privacy CA as a third party:

1. **Virtualization:** The Privacy CA is hosted as a VM in a hypervisor of the third party's infrastructure to isolate the trusted services in a compartment of their own.

2. **Restrict Trusted Computing Base to a minimum:** The platform (OS and its components) over which the Privacy CA is run is kept to the required minimum number of applications to ensure easy verification of the platform.

3. **Formal methods to prove the security:** The cryptographic protocols are formally verified.

4. **Use of safe programming language:** A safer programming language such as Java that does not allow direct memory access through pointers (such as C, C++) is used for them implementation of the service.

5. **Source code made available to public:** The application source code is made public for evaluation and improvement.

6. **Use Trusted computing:** Clients should be able to remotely attest Privacy CA using trusted computing.

Based on these guidelines Pinker et al. developed a self-contained image of Privacy CA that can be executed stand alone or over Xen. A custom ASCII-text based communication protocol is used against XML to counter the overhead of XML. The TCB is kept to a minimum as shown in Figure 4.4.

The Privacy CA implemented by Pinker et al. supports the following operations:

- **aik_create**: This operation creates AIK certificate as specified in TCG [106].

- **aik_validate**: This operation in PrivacyCA validates the AIK certificates issued by PrivacyCA. The certificates are provided for validation by the clients.

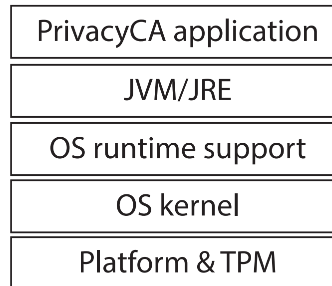

Figure 4.4: Privacy CA: Trusted Computing Base

- **aik_locate**: This operation searches for a particular AIK certification given an identifier for AIK

- **aik_revoke**: This revokes the individual certificate for future validation

- **tcb_quote**: This asks the PrivacyCA to integrity measurement of itself. This is used to remotely attest the PrivacyCA.

The infrastructure presented in this section describes how to effectively collect and report measurements to a third party and how to authenticate a TPM. The TCG specifications are the basis of the implementation of several other trusted platforms [37, 72, 68, 117]. The principles of remote attestation are also applied in embedded systems such as Automatic Teller Machines [90] and mobile phones [85, 44].

4.2 Virtual Machines based Architectures

This section presents the state of the art in VM based remote attestation architectures. There are a variety of approaches adopted in the literature for using remote attestation in VM based architectures: semantic division of VMs as secure and insecure [49], real-time measurement as described in Section 2.5.4, measurement within the VM [70] and multiplexing a single physical TPM to multiple vTPMs [18]. We present some of these architectures in this section.

4.2.1 Terra

Garfinkel et al.'s Terra [49] represents one of the earliest works in VM based architectures for remote attestation. It implements a VM monitor called TVMM (Trusted Virtual Machine Monitor) that partitions a TPM into multiple isolated VMs. Each VM can either be an open box or a closed box. This partition is made to provide the functionality of running the sensitive applications along with traditional applications. The open box is a general traditional platform without any trusted computing protocols and closed box is an opaque special-purpose platform that protects the privacy and integrity of the software using trusted computing.

Using a TVMM, existing OS and utility applications are run in a standard VM (open box). The applications that require strict privacy protection are executed in a closed box, where the running software stack (including the OS, all the applications) is cryptographically authenticated to remote parties through attestation.

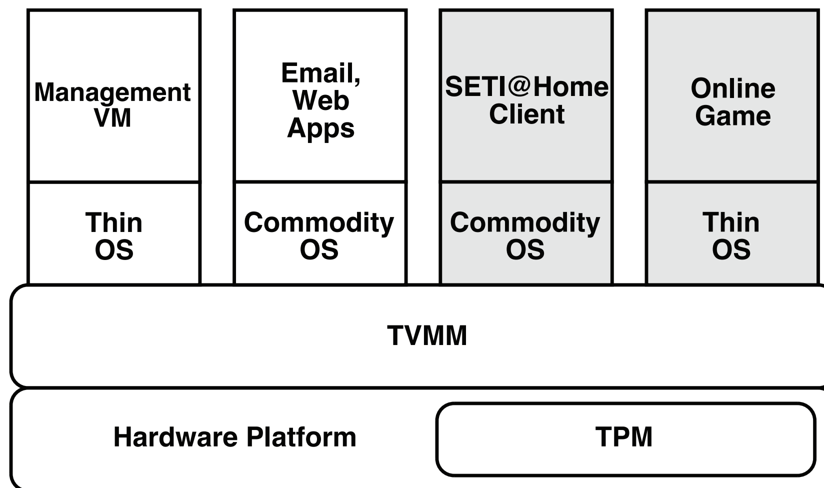

Figure 4.5: Terra Architecture [49]

The Terra architecture is presented in Figure 4.5. The TVMM is hosted on the hardware platform (TPM) that supports attestation. The hardware platform should be able to attest the booted operating system and the TVMM. The TVMM is secure from tampering not only from the platform user but also by the platform owner because of remote attestation mechanisms. It will neither falsely attest the VM's contents nor will it allow tampering of closed box VMs.

Garfinkel et al. [49] also studied two different types of attestation: Ahead of time attestation

and optimistic attestation. Ahead of time attestation is similar to IMA (Section 4.1.1), where each stage in the boot process is responsible for signing the hash of the next stage before invoking it (hash before load). Optimistic attestation works at a individual disk block level, wherein, each hard disk block of the VM is checked when read from the disk (The hash values of each block is stored by the TVMM for verification).

The TVMM supports two abstractions of the VM: closed box and open box. The closed box VMs are shown in gray in Figure 4.5. The content of a closed box cannot be modified or inspected by the owner of the server that hosts Terra. A remote client can verify that the closed box applications have not been modified by checking the measurement list of the closed box VM. The closed box may run a special trusted OS with a set of applications specially designed for it.

Figure 4.5 also depicts one open box VM that runs commodity OS and applications (Email, Web Apps) and a management VM. The management VM is responsible for granting storage and memory to other VMs. It also allows users to connect, start and stop a VM.

Garfinkel et al. [49] prototyped Terra in VMWare server with Debian GNU\Linux. Their prototype was not implemented over a trusted hardware device, as the authors believed the operations with the TPM device are well defined and hence the implementation will be superfluous. For experiments, the open source game Quake was run in the closed box as a standalone application without user shell. Booting the quake closed box without any attestation took 26.6 seconds, ahead of time attestation took 57.1 seconds and optimistic attestation took a total of 27.3 seconds without encryption and 29.1 seconds with encryption. Hence the authors concluded optimistic attestation to be ideal for Terra.

4.2.2 HIMA

One of the major issues with attestation is the Time of Check to Time of Use (TOCTTOU) vulnerability, where the platform is subject to attack after the verification is performed by the client. The adversary may wait until the verifier has received the attestation before rebooting the platform in order to boot a malicious software image. Under traditional attestation methods it is not possible to detect if the platform has been tampered with between the verification

and the execution of a software binary, as there will be always a time difference between the verification and the usage of the server. To address the TOCTOU issue, Ahmed et al. [26], proposed HIMA with two key functionalities: active monitoring of guest VM kernel events and guest VM memory protection. The former ensures that the integrity values are refreshed when the memory layout changes and the latter does not allow any program to bypass HIMA before execution.

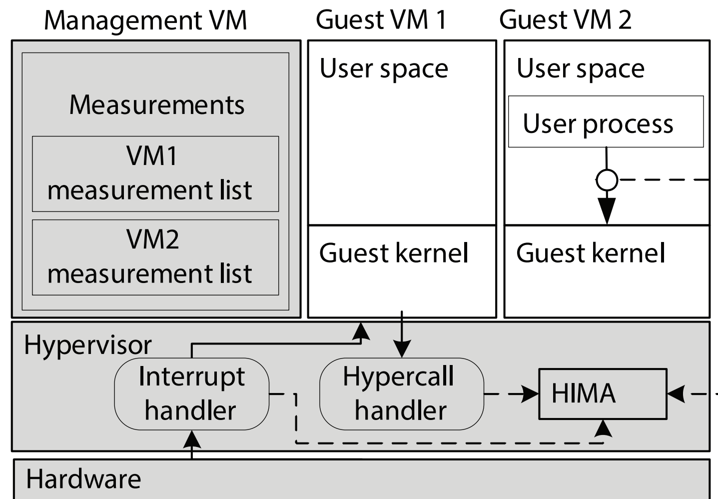

Figure 4.6: HIMA Architecture [26]

The HIMA architecture is presented in Figure 4.6. The trusted components in the architecture are depicted in gray. HIMA is isolated from the measurement targets (Guest VM1 and Guest VM2) and it is transparent to the VMs. The hypervisor's code is modified to place hooks where guest VMs kernel events are handled. HIMA intercepts all events including hypervisor service events, system calls and hardware interrupts. Using these intercepts, HIMA keeps the measurements lists fresh by actively monitoring the memory map of guest VMs for any change (This will include page faults, context switches, etc.).

The other important functionality of HIMA is guest memory protection, which ensures only the measured binaries are executed by Guest VM. This is achieved by setting a page protection flag on all the logical pages on the guest VM. Setting the protection flag traps into the hypervisor every time there is a guest memory access. At this stage, the hypervisor can check if the page is measured and is allowed to be executed.

Thus the guest memory protection and active monitoring of the memory enables HIMA to achieve TOCTTOU.

HIMA prototype was developed using Xen hypervisor and experiments were conducted using Unixbench and application benchmarks (`gzip` and `bzip2` compression). Except for process creation system calls such as `execl` and shell scripts HIMA introduced around 5% overhead. The `execl` and shell scripts introduced 75% overhead and 99% overhead respectively.

4.2.3 In VM measurement

HIMA and Terra had modules to measure the VM from the hypervisor. This approach is also called $out-of-VM\ monitoring$. These architectures suffer from major performance overhead because of context switching between the hypervisor and VM. Qian et al. [70] proposed an In-VM secure measuring framework for improving the performance of VM attestation.

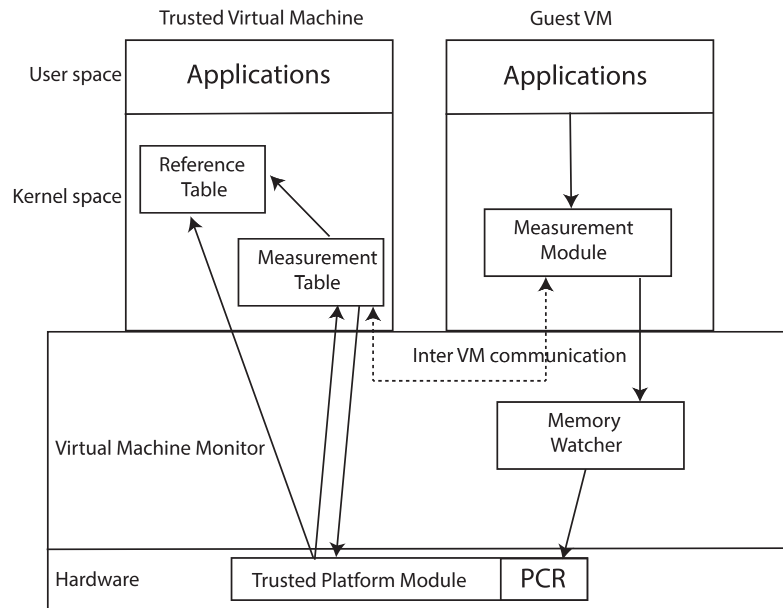

Figure 4.7: In VM measuring framework

Figure 4.7 presents the In-VM measurement framework proposed by Qian et al.. The server hosting the framework has a Trusted Platform Module (TPM). Each guest OS has a measurement module. The measurement module is responsible for storing the hash values of the ap-

plications that are loaded in the VM's memory. The measurement module transfers the new measurements to the measurement table of the trusted virtual machine (Dom 0). The framework also *extends* the new values to the PCR. The safe trusted hash values of the applications that are run in Guest VM are are stored in the reference table. The memory watcher module in the hypervisor periodically verifies the measurement module in the guest OS and recalculates the hash value to check if the values are secure by comparing them against the reference table values.

Qian et al. [70] implemented the In-VM framework using Xen VM. The hash value was measured using the `struct security_operations` hook in the kernel. Experiments were performed by running the framework both in Dom 0 and Dom 1 under three benchmarks from Stanford applications for shared memory [103]. The authors found the framework introduced very low performance overhead in the order of 8.79% in DOM1 and 4.13% in DOM0.

4.2.4 Certicloud

Benoit et al.'s Certicloud [34] proposed another "In VM measuring architecture" through two protocols `TCRR` and `VerifyMyVM`. These protocols guarantee integrity of the remote server and detects tampering attempts on its VM, on-demand.

TPM-based certification of remote resource (`TCRR`) allows the cloud user to verify only authorized code is running in the remote server. The protocol has three actors: user U, remote resource or node N and the associated TPM. The user U prior to the initialization of the protocol has the knowledge of the secure state of remote resource N. There are two phases to this protocol: Node integrity check and User session key exchange. In node integrity check, a user requests for the aggregated hash value, TPM_Quote (Section 2.5.3) from the TPM of the node N. The TPM returns the hash of the PCR (Platform Configuration Register) values to the user. The user checks the PCR values against the known value. This comparison permits the user U to conclude if the remote node is secure. The user after checking the remote node, sends across a session key to the IaaS cloud, encrypted by the public key of the TPM so that only the TPM can decrypt it. The session key is then used for any future transactions.

Certicloud was designed based on the `TCRR` protocol to protect IaaS platforms. It is pre-

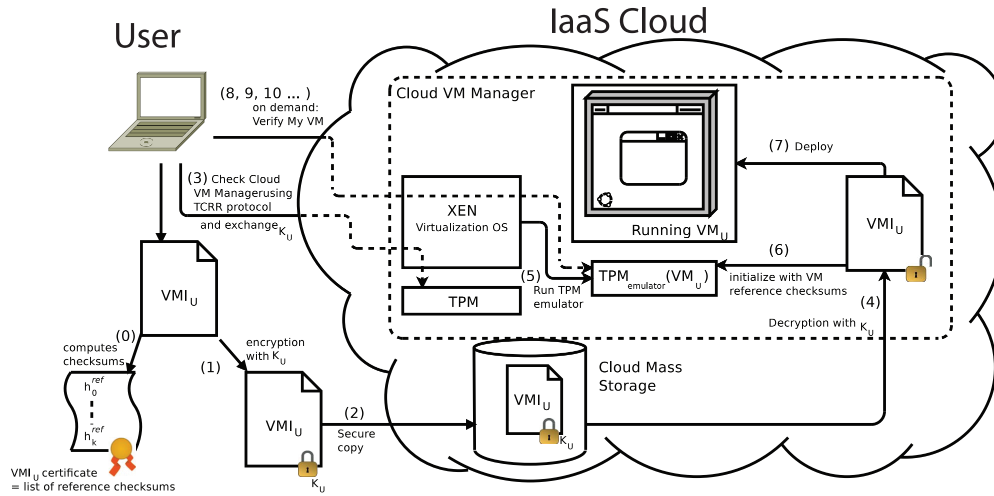

Figure 4.8: Certicloud Architecture Architecture [34]

sented in Figure 4.8. In this architecture, the user prepares a VM with known hash values and deploys it in IaaS cloud. To deploy securely, the following steps are proposed:

1. The user create a private key K_u and encrypts the user's VM image, VMI_u.
2. The encrypted image is sent to the IaaS cloud.
3. The TCRR protocol is invoked to verify the integrity of IaaS provider. Once the check is completed, the user shares the key K_u with the TPM of the cloud VM manager.
4. The cloud VM manager creates a new emulated TPM (virtual TPM) for the user image VMI_u.
5. The emulated TPM acts as the TPM for the user image and contains the PCRs of VMI_u.
6. VMI_u is deployed as a new virtual machine in IaaS.
7. The integrity of VMI_u is checked on demand by the user of the `VerifyMyVM` protocol.

`VerifyMyVM` provides the user with the hash values of the VM through the emulated TPM. The user checks the hash values against the known values to identify if the VM is tampered with.

Benoit et al. [34] implemented the Certicloud prototype using the Xen VM environment. `TCRR` took a total of 9.28 seconds and `VerifyMyVM` took 15.69 seconds to complete.

4.2.5 vTPM

Certicloud proposed two protocols to safely deploy the custom prepared VM by the user in the remote architecture. They also emulated TPMs in individual VMs. However, Stefan et al.'s virtual TPM (vTPM) [18] is considered to be seminal work on sharing TPM across multiple VMs. vTPM multiplexed a physical TPM as multiple virtual TPMs, providing each with an illusion of trust for enabling remote attestation. A modified implementation of vTPM is adopted by Xen and KVM.

Requirements: The requirements identified by Stefan et al. [18] for a vTPM are the following:

1. vTPM should provide the same command set to an OS, as a hardware TPM. The OS should be able to run on vTPM in the same way as it does over TPM without or minimal code change.

2. On migration of VMs, vTPM's needs to be migrated along with VM to maintain strong association.

3. During migration vTPM's should maintain an association with the underlying trusted computing base (TCB). The challenger while remote attestation should be able to follow the trust chain from the TCB (that includes the hardware TPM and VMM) to vTPM.

4. The remote party should be able to distinguish from a vTPM and a TPM, as both offer different security properties[3].

Architecture: The vTPM architecture is presented in Figure 4.9. The architecture consists of a vTPM manager hosted in a separate management VM consisting of vTPM instances. Each

[3] A vTPM's correct functioning relies on the TCB, hence the remote party should be able to identify that.

VM that needs a TPM functionality is assigned its own vTPM instance by the vTPM manager. vTPM implements TCG TPM specifications [106].

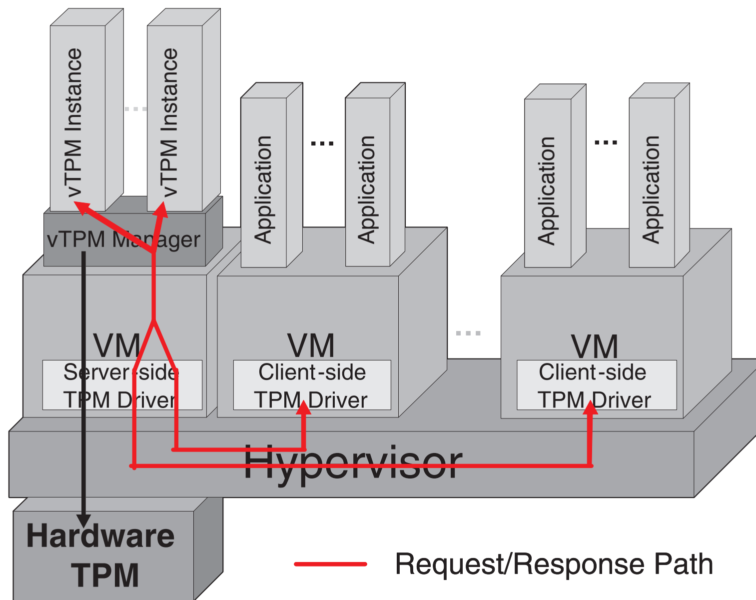

Figure 4.9: vTPM Architecture [18]

The VM communicates with its vTPM instance hosted by the vTPM manager, through the VM's client side TPM driver (denoted by red lines in Figure 4.9). VMs host the IMA enabled kernels (Section 4.1.1). The VM maintains measurement lists rooted by the aggregated hash stored in the vTPM instance hosted in vTPM manager.

The vTPM manager in itself communicates with the hardware TPM and is used to store the aggregated hash of the software hosted in management VM. Hence a trust chain relationship is formed between the hardware TPM and vTPM manager, and vTPM instances hosted by the vTPM manager and the VMs hosting the applications.

The vTPM manager supports creation, deletion and setting up of vTPM instances. It also aids the vTPM migration during the VM migration.

The authors implemented the architecture using Xen hypervisor. Indeed Xen 4.3 [16] onwards has started supporting vTPM based on the work form Stefan et al..

Some of the architectures that uses vTPM at its core are presented below.

myTrustedCloud

vTPM proposed a complete overall architecture for virtualizing TPMs to share a single physical TPM across multiple VMs. Wallom et al.'s myTrustedCloud [110] explored a use case for trusted computing system with vTPMs in a complete cloud infrastructure for a real world scenario. The authors integrated trusted computing into Eucalyptus (an open source implementation of cloud infrastructure) for the UK energy industry.

The use case involved different energy operators in the UK that share sensitive data within each other. The shared data has different access and usage privileges, across a known set of operators. The ownership and the integrity of the data must be trusted. The authors used trusted computing to enforce the access and usage privileges and to protect the integrity of the data.

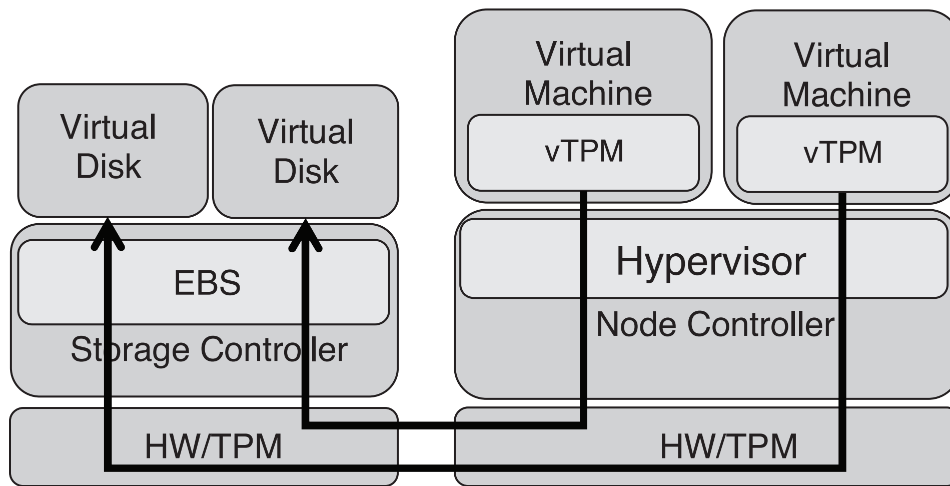

Figure 4.10: myTrustedCloud Architecture

Based on the use case, myTrusted cloud proposed an architecture with node controllers (NC) and storage controllers (SC) (Figure 4.10). The node controllers are responsible for the controlling the VMs that are computational units of the system. Each VM has a vTPM to measure its files. The hypervisor hosting the vTPM is measured by the hardware TPM.

The storage controller hosts Elastic Block Storage (EBS) that is responsible for file storage. The storage controller is measured by the physical TPM in its server.

For the framework to be trusted, NC, SC and the VMs needs to be attested. When a client

requests the measurements from the framework, an *iterative attestation* is adopted. Instead of sending attestation values from three different attestation sessions (for NC, SC and VM), in a single session, an attestation ticket combining PCR values from the three different TPMs is sent back to the client.

The authors implemented the system using QEMU and Kernel Virtualization Module (KVM) of Ubuntu based Eucalyptus cloud. The Open Platform Services (OpenPTS) [10] are used to implement the VM attestation procedure by the client. In experiments authors found that the framework has significant overhead during boot time (3.18 minutes for TPM based cloud compared 57 sec for non tpm based cloud). However there was negligible operational overhead, as most of the measurement happens when the system is rebooted.

Implementation of Trusted IaaS using vTPM and Trusted Third Party (TTP)

The frameworks suggested Terra [49], HIMA [26], Certicloud [34] and myTrustedCloud [110] focused on an attestation framework, in which the verification is performed by the client or the user of the services. However, this requires the client to have a record of all the measurements of the secure versions of the software that are executed in the cloud and the cloud configuration to be disclosed to the user.

Li et al. [67] proposed a trusted IaaS framework which involves a trusted third party that is responsible for attestation (Figure 4.11). Usage of TTP minimizes the cloud configuration disclosure to the user. Moreover, the user need not be aware of the measurements of secure versions of the software. The authors proposed a protocol that uses a TTP for VM verification.

The architecture of the system proposed by Li et al. is presented in Figure 4.11. The IaaS cloud consists of several nodes (servers) and is controlled by a cloud manager. The users interact with the cloud manager for using their services. The TTP is responsible for remotely attesting the state of the nodes in the cloud manager.

The software configuration of the framework in the node is inline with myTrustedCloud. The server hosts vTPM capable hypervisor (such as Xen or KVM) with IMA enabled VM in the nodes. The VM also hosts a collection agent that transmits the measurement values of the VM to the TTP.

Before starting any session with a node in the cloud, the users request that the TTP verify

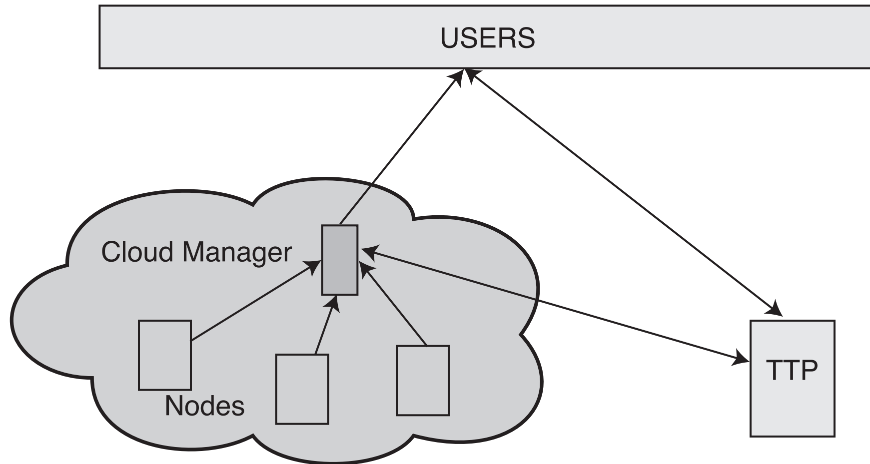

Figure 4.11: Trusted Cloud with TTP

the state of the cloud using their custom attestation protocol. The TTP invokes the attestation service from the cloud manager which in turn returns the measurement values to the TTP by calling the node collector service of the requested node's measurement value. The TTP verifies if the measurement values are secure and then it sends the result to the user. If the result is successful the user initiates the new connection with the cloud manager.

The authors prototyped the system using KVM based hypervisor. Similar to myTrustedCloud they used OpenPTS [10] services for transacting between the cloud and TTP. The experiments using the protocol showed a very little overhead while in operation (2% to 5% overhead). The booting overhead is not shared by the authors.

4.3 Our Work

In our experiments we found that a minimal server running Debian GNU\Linux without any VMs and extra applications requires at least 360 packages. Each package contains at an average of 5 to 10 files (Section 4.8). The verifier needs to be aware of these files and their measurement values for effective verification. A typical IaaS server can host hundreds of software packages. A major challenge in adopting remote infrastructure is the management of these software with its multitude of versions and updates that needs to be measured and maintained by the verifier.

None of the work we surveyed dealt with this problem.

Separation of concerns[4] is one of the design principles of software architecture design. In almost all of the work we surveyed the cloud client is responsible for two concerns simultaneously; transacting with the cloud provider for utilizing the provider's services and the verification of the cloud provider's state. We propose there is a need for a trusted third party. The cloud client can delegate the functionality of the remote attestation to the trusted third party. The delegation enables the clients to focus on consuming the provider's services and not on the complexities involving verification of the provider. Li et al. [67] proposed an architecture involving TTP. However, in that architecture, the clients are pro-actively required to verify the cloud provider's state through the TTP before initiating any connection with the provider. This introduces additional steps to the transaction workflow of the client. Moreover, there are no guarantees to the state of the cloud provider when the client is not actively communicating with the provider.

In this work, we address the software hash maintenance challenges of secure software by proposing an infrastructure ecosystem. This infrastructure is built on the design principle of separation of concerns. Our work does not require the client to modify their transaction workflow. The verification is performed at predefined intervals by the TTP and in the event of verification failure, the client is notified of the failure.

4.4 Architecture

The proposed infrastructure has five different entities as depicted in Figure 4.12: client, Trusted Third Party (TTP), IaaS cloud, RIMM Saas Cloud and Software vendors. The arrows in the Figure 4.12 denote the dataflow in the infrastructure. Adhering to the principle of separation of concerns, each entity in the infrastructure is responsible for one distinct functionality. These are listed below.

- **Client:** The client transacts with the IaaS server and VM for the delegation of functionality.

[4] Separation of concerns allows a module or a design to be split into distinct sections, such that each section addresses a separate section.

- **IaaS Server:** The IaaS server provides services for clients and allow remote attestation by providing measurement values to the TTP. All the servers in the IaaS infrastructure are referred to as IaaS servers.
- **Trusted Third Party (TTP):** The TTP verifies the measurement values provided the by IaaS cloud against the values provided by RIMM SaaS cloud and the client.
- **RIMM SaaS Cloud:** The RIMM SaaS cloud audits and maintains hash values of all valid software and its updates.
- **Software Vendor:** Software vendors provide fixes and notify the RIMM SaaS cloud with valid values.

The client registers the IaaS server hosting the virtual machine and the virtual machine (target) for verification with the Trusted Third Party (TTP). The TTP polls the target machines in a predefined interval and gets the measurement list from the target machines. Resource Integrity Measurement Manifest (RIMM) SaaS cloud is the repository of the valid integrity values (hash) of all the software binaries that are allowed to be executed in a server hosted in IaaS Cloud. The TTP uses these integrity values from the RIMM and verifies them against the integrity values obtained from the IaaS. When the verification fails, TTP notifies the client about it. The integrity values of RIMM are updated by authenticated software vendors. The proposed infrastructure assumes the following cardinality of the entities.

- The client and IaaS server have a many to one relationship. There can be many clients transacting with one IaaS server.
- The client and TTP have a many to one relationship and the TTP and the IaaS server has one to many relationship.
- The TTP and RIMM SaaS cloud have many to one relationship. There is only one RIMM SaaS cloud.
- There is one RIMM SaaS cloud to ensure a centralized auditing and verification system of software.

In this section, we detail the functionalities of the each component and the protocols used in between them for communication.

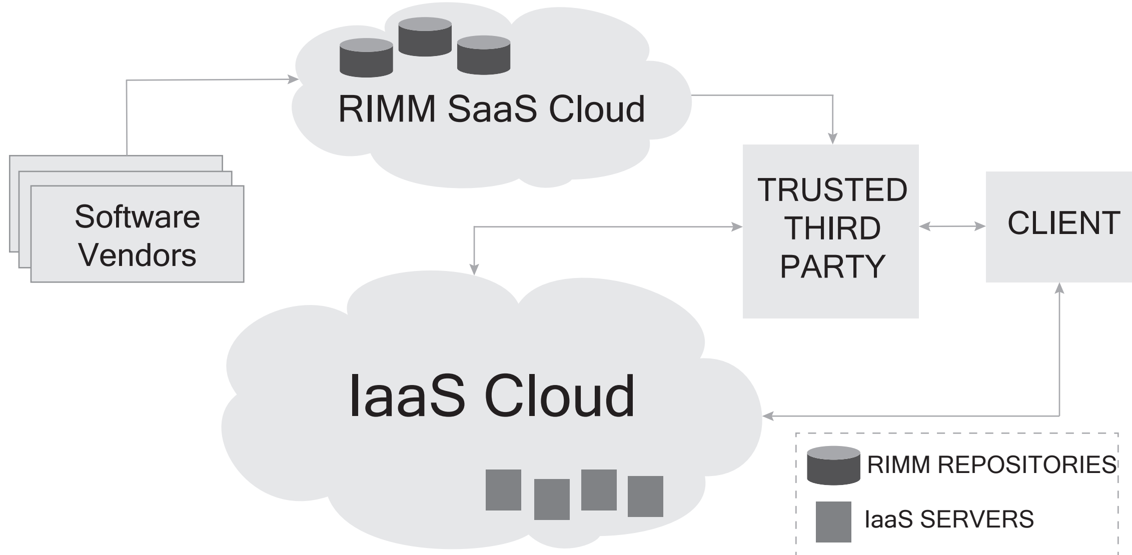

Figure 4.12: Architecture

4.4.1 IaaS Cloud

The IaaS is a cloud service model where resources such as the physical/virtual machines from the cloud provider are rented to the cloud client (Section 2.1.1). The IaaS cloud is composed of several IaaS servers. Each IaaS server hosts a domain controller that is responsible for creating and managing the server's virtual machines. For the rest of the discussion IaaS server and domain controller are used interchangeably. The IaaS sever has a predefined software and hardware platform.

The components of the IaaS server are presented in Figure 4.13. Along with the standard hardware such as processors, harddisk and memory, the IaaS server has a Trusted Platform Module (TPM) that serves as the root of trust. The IaaS server uses the Integrity Measurement Architecture (IMA) enabled GNU\Linux kernel [98]. The IMA kernel measures all the files before they are mapped (`mmap` system call) into the memory for execution. The hash values are stored in a secure measurement list in the server's file system. The aggregated hash value (Chapter 2) of the measurements are stored securely in a PCR of the untamperable TPM. The

IaaS Trust Component is responsible for relaying the measurement list along with the TPM PCR values to the trust verifier, hosted by the TTP. The server hosts a domain controller and other supporting applications that are used for managing the VMs.

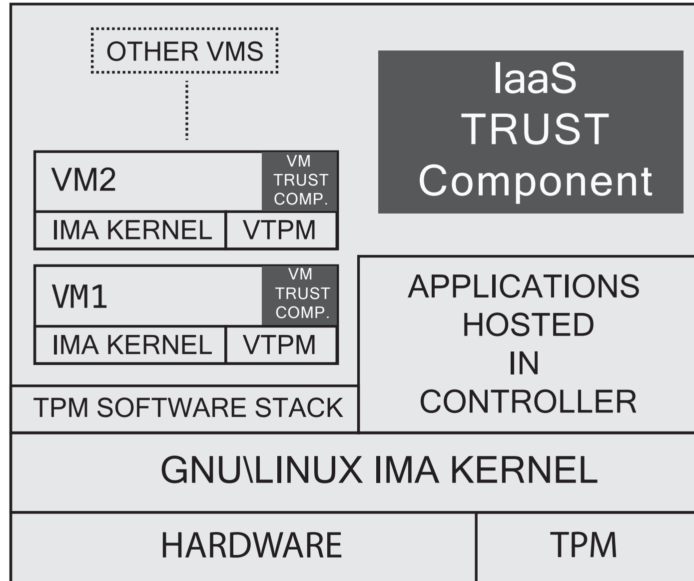

Figure 4.13: IaaS Server Components

The VMs are allowed to host only IMA enabled kernels. This can be verified through remote attestation. The virtual TPMs [18] act as the root of trust device for the VM-IMA. Each instance of the VM has a VM Trust Component, that relays the measurement list from the VM to the same trust verifier, hosted by the TTP.

IaaS Trust Component

The IaaS and the VM trust components provide interfaces to securely transfer and validate the measurement list. The trust components transact with the trust verifiers hosted by TTP, through services.

Services hosted by the IaaS Trust Component:

The IaaS trust component hosts the service `request_ml()` that is used to retrieve the measurement list of the domain controller.

`request_ml()`: The service reads the measurement log created by the IMA kernel of the domain controller and sends it to the TTP along with the aggregated hash from the PCR of

TPM signed by the TPM's Attestation Identity Key (AIK). The measurement list is a collection of the following information:

$$\{id, hash_value\}$$

where *id* is the unique identifier representing the software binary and the *hash_value* is the integrity measurement.

Services hosted by the VM Trust Component:

The services hosted by the VM trust component are used to validate the state of the VM. The following are the services hosted by the VM trust component:

- `init_ml(ml)`: This service is invoked by the logged in client user from the a freshly created VM in the IaaS server. It executes the following functions:
    1. Invokes the service `request_ml` of the controller and forwards it to the service `verify_controller_ml` of the trust verifier (Section 4.4.2) to check if the controller is in a safe state.
    2. Invokes the service `init_vm_ml` with the VM's measurement list of trust verifier (Section 4.4.2) to check the initial state of the VM and to register the VM.
- `request_ml()`: This service has the same functionality as the IaaS server's `request_ml`.

4.4.2 Trusted Third Party

The trusted third party hosts the trust verifier and the Resource Integrity Measurement Manifest (RIMM) datastore (Figure 4.14). The trust verifier registers the IaaS server's domain controller and the VM, and verifies the state of the controller and VM in predefined intervals. The verification of the state is performed by comparing the software measurements from the IaaS server and VM against the reference software integrity values stored in the RIMM datastore.

The RIMM datastore is updated with valid integrity values of safe software either from the RIMM SaaS cloud or by the client. The domain controller of the IaaS server is verified only using the values provided by the RIMM SaaS cloud. In other words, the IaaS is only allowed to execute software that is approved by the community managed RIMM SaaS cloud.

The verification of the VM can either be done by the values provided by the RIMM SaaS cloud or by the client. This allows the clients to deploy their own custom trusted applications on the VM.

The software stack of a TTP can be limited to known software as the TTP has a well-defined API. For example, if the TTP is exposing its services using XML web-services, the TTP can be restricted to run a minimal GNU\Linux OS with Java Runtime Environment and Tomcat Apache web server. Therefore the state of the TTP can be in turn verified by any external entity. However, to avoid complexity at the TTP level, we will assume that the the TTP does not need verification and is trusted by the client and IaaS.

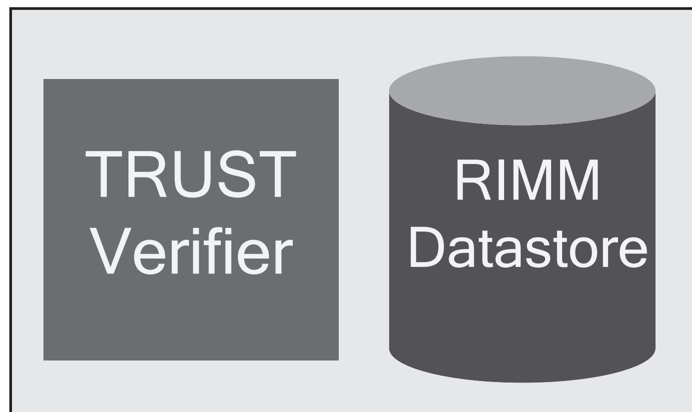

Figure 4.14: Trusted Third Party

Trust Verifier

The trust verifier is responsible for attesting the state of the IaaS server's domain controller and VM. It offers the following services:

- `register_vm(vm, controller, vm_policy)`: This service is invoked by the client to register a new vm and controller with a policy with the TTP. This service registers the domain controller referred by the variable `controller` and the virtual machine hosted by the IaaS server (domain controller), referred by the variable `vm` for verification with a policy specified in the variable `vm_policy`. In an IaaS scenario, a controller can be shared with multiple clients (multi-tenancy support). Hence a controller could be already

registered for verification by a different client. If the variable `controller` is not already registered by a different client, this service invokes the service `request_ml` from the IaaS trust component to verify the server's state and then it adds the variable `controller` and `vm` in its local controller verification list. The variable `vm_policy` specifies a set of software versions allowed to be executed in the VM and the predefined interval for the polling of the IaaS server. Section 4.4.5 discusses the policy in detail and Section 4.5.1 details the registration scenario.

- `verify_controller_ml(ml)`: This service is invoked by the VM to verify its controller. The service verifies the measurement list, `ml` against the valid values of the controller and returns *true*, if the verification was successful.

- `init_vm_ml(ml)`: This service is invoked by the service `init_ml` of the VM trust component during registration of the VM with the TTP (Section 4.4.1). The measurement list, `ml` is verified against the policy specified by the client while registration.

- `verify(target)`: Invokes the service `requst_ml` of the target machine. It can be either IaaS server's domain controller or a VM. The trust verifier verifies if the `ml` has valid values by comparing it with the reference values from RIMM datastore. The verification scenario is detailed in Section 4.5.2.

RIMM Datastore

The RIMM datastore is the repository of all the valid integrity measurement values of the software binaries. The RIMM datastore updates its database from two different sources: RIMM SaaS and client. RIMM SaaS is a community maintained database of all safe software values. The client provides the integrity values if they want to host their own custom developed applications that are not approved by the RIMM SaaS[5]. The measurement values from RIMM SaaS cloud are used as the reference values for comparison with the values from the controller and VM. The client provided measurement values are used as reference values for the comparison with the values only from the client owned VM. The RIMM datastore stores the software hash values in the following format:

[5]The client may not want to submit the application to the RIMM SaaS due to Intellectual Property (IP) concerns

$$\{id, version, hash_value, origin, client_id\}$$

The variable *id* is the unique identifier representing the software binary, the variable *version* denotes the version of the software, the variable *hash_value* indicates the hash of the binary and the variable *origin* takes two values: `global` or `client`. When the variable *hash_value* is `global`, the hash value is managed by the RIMM SaaS cloud and when it is client, the client specifies the hash values for this software binary. The variable *client_id* stores the client's unique identifier and the variable *client_id* is relevant only if the hash_value is `client`.

The integrity values are updated using the following service:

`publish_client(software_info)`: The RIMM SaaS cloud or the client notifies the TTP's RIMM datastore of software updates that were subscribed by the TTP. This service appends the new software update hash values in the datastore and alerts the clients when the IaaS servers and VM do not update their software as recommended by the vendors or clients. The variable `software_info` contains the following information:

$$\{software_id, prev_version, version, update_type\}$$

The *software_id* is the unique identifier representing the software binary. The variable *version* represents the current version number of the update. The variable *update_type* denotes the severity of the update. The variable takes one of the two values: `critical` and `non_critical`. The IaaS server or the VM is expected to install the critical updates as these updates carry security bug fixes. The non-critical updates need not be installed by the IaaS server or the VM. With critical updates, the client is notified immediately if the IaaS server or the VM has not updated the *prev_version* of the software binary with the new version. However, failure to install the non critical updates, leads to passive logging by the trust verifier, that can be in the future observed by the client.

4.4.3 RIMM SaaS Cloud

The RIMM SaaS Cloud (RSC) provides the clients with a list of all verified valid integrity values of the software that could be executed safely in a domain controller and a VM. RSC is also responsible for the auditing of the software to ensure they are not vulnerable or malicious.

The auditing functionality of the RSC is beyond the scope of this work. The RSC hosts the following services:

- `subscribe(software_list)`: Given the variable `software_list` that needs to be verified, this service subscribes the invoker to the future software update notification and returns the valid integrity value of all the versions of the software specified in the software list. This service is invoked by the trusted third party to update its RIMM datastore with the valid integrity values. When a software receives a critical update, RSC gets the latest valid verification values from the software vendor and RSC notifies all the subscribed clients about updated software. Each client of RSC, the TTP, is expected to implement the `publish_client` service for receiving the updates from the RIMM. The update scenario is detailed in Section 4.5.3.

- `notify_update(update_info)`: The software vendors invoke this service every time a new version of software is released by the vendor. The variable `update_info` represents the version and the update type. The update type denotes if the update is a critical update that the user of the software should apply or an non-critical update, where the update to a new version is optional.

In a typical workflow, the SaaS cloud audits the newly updated software binary and check for any security vulnerabilities. Once the software's audit is successful, the new update is published to all the clients (TTP).

4.4.4 Client

The client in this architecture registers the VM and the domain controller with their custom measurement list policy with the TTP. The client also subscribes for notifications from the TTP when the verification fails or there are recommended updates that needs to be performed in the VM, as specified in the VM policy.

4.4.5 VM Policy

The VM policy is used by the client to specify advanced software verification configuration and the update requirements of the client. The policy allows the client to denote a list of allowed software (whitelist) that can be installed in the VM along with the software's update configuration. Platform Trust Services (PTS) specification [12] proposes a static whitelist management of software. We present a policy with dynamic whitelist with the following functionalities:

- **Integrity value origin:** The integrity values can be either provided by the RIMM SaaS cloud or by the client (The client may trust the software that they deployed in the VM and do not want to take the RSC route). To accommodate both the integrity values from the client and the RIMM datastore, we introduce *origin* attribute. The *origin* attribute also allows the client to specify that certain software that are auto updated needs to verified against the hash value of the most recent version of the software.

- **Application Chaining:** For some deployments, there is a need to specify that few applications has to be loaded before other applications, e.g, the client may want the database server and the logging server to be loaded before the webserver, to avoid any error requests. We propose the use of run levels for achieving application chaining (Section 4.4.5).

- **Alert levels:** This functionality allows users to define and customize various notification levels and associate policy violations to a particular level, e.g, For some VMs the user may not want aggressive notifications when there are policy violations. They may desire in a daily/weekly digest mail with a list of violation. However, for violations in mission critical applications, the user may expect automated phone calls, SMS or high priority emails.

Formalisms

The formalism for the policy based on the functionalities described is presented here. This is an extension of the PTS specification [12].

The policy P is expressed as a set of the software configurations (S), notification levels (N) and time interval (T).

$$P = \{S, N, T\} \tag{4.1}$$

S is a list of individual software configurations, s.

$$s = \{id, build_info, origin, r_level, n_level\} \tag{4.2}$$

The variable *id* represents the unique identifier of the software binary. The variable $build_info$ denotes the details of the build (expanded in Equation 4.3). The variable *origin* represents if the client is presenting the hash value or the hash value has to be fetched from RIMM repository. The variable *origin* has valid values: `global` or `client`. The variable r_level is the run level of the software, indicated by integers. This will aid in establishing the application chain in the system. The variable n_level represents the notification level associated with the policy violation. This notification level maps to the variable *level* of Equation 4.5.

The variable $build_info$, either has the $version_number$ or *tag*.

$$build_info = \{version_number \mid tag\} \tag{4.3}$$

The variable $version_number$ specifies the fixed version number (build version, major version and minor version with patch levels) of the software that is allowed to be hosted. The variable *tag* allows extra flexibility in specifying the current software version. Instead of fixed number, the variable has one of the following values: `stable`, `unstable` or `testing`. The values `stable`, `unstable` and `testing` denotes the latest stable, unstable and the testing builds of the given software respectively. The trust verifier fetches the hash values of the specified tag branch of the software from the RIMM for verification.

The variable *origin* has one of the following values: `global` or `client`.

$$origin = \{\texttt{global} \mid \texttt{client}\} \tag{4.4}$$

where the variable `global` represents the hash value is updated by the RIMM SaaS cloud

and the variable `client` denotes that the hash value is provided by the client.

The notification levels, N from Equation 4.1 is represented as

$$N = \{level, options\} \quad (4.5)$$

where the variable *level* is the identifier that is used in the software configuration s (Equation 4.2) and the variable *options* represents the type of notification (e.g., notification through email, phone etc.).

4.5 Scenarios

In this section we present the orchestration of the different units in our architecture, for the registration, verification and update RIMM operations in the system.

4.5.1 Registration

The registration scenario of a remote server is presented in Figure 4.15. The client (user) that needs to use the services of the TTP for verification of IaaS, registers the IaaS with the `register_vm` call to TTP.

The algorithm of the `register_vm` service is presented in Algorithm 1. The algorithm takes as input the invoker *client*, *controller* and *vm* that needs to be registered along with the variable *vm_policy*. The TTP checks if the domain controller was registered in the TTP by some other client (Line 1). If the controller is not registered, the TTP invokes the function *get_ml* (function *get_ml* calls the corresponding controller's *request_ml*) to receive the *ml*, the measurement list, from the IaaS trust component (Line 2). The function *get_ml* returns the measurement list to the TTP and the TTP verifies the measurement list against the known software values. If verification is successful then the controller is added to verification list of the TTP (Line 6). The policy parameters, S, N and T, described in Section 4.4.5, are extracted from *vm_policy* (Line 9). The software binary configuration of each software in S is stored in the RIMM repository (Line 11). The control is returned back and the TTP waits from initialization of VM from the user at this point (Line 13).

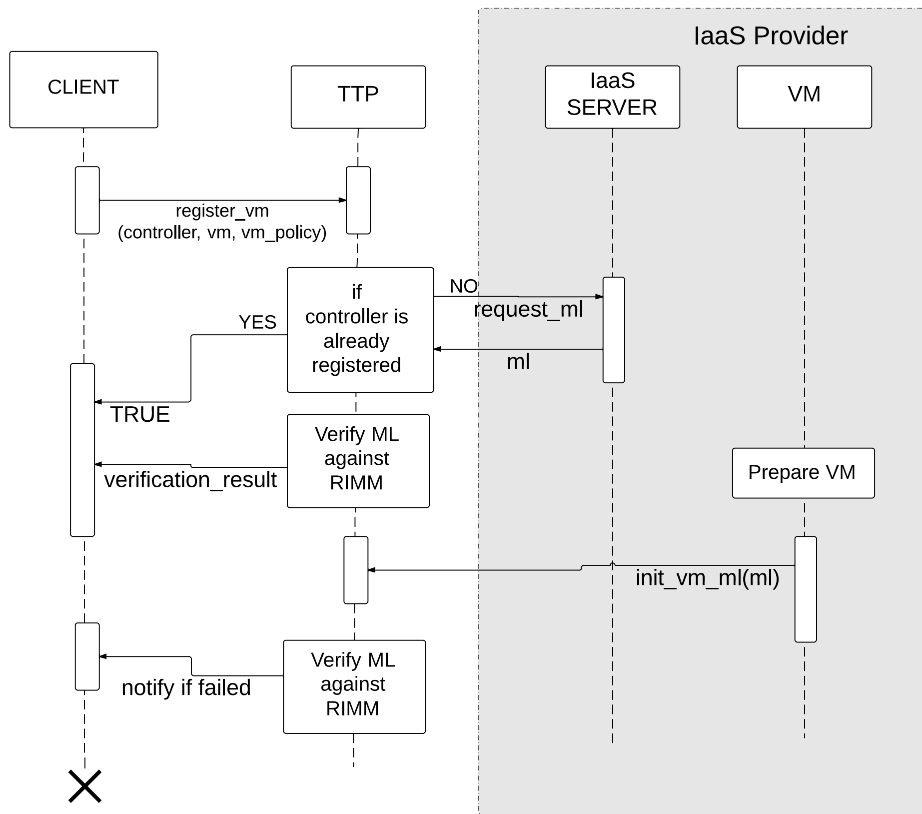

Figure 4.15: Registration Scenario

If the verification of the IaaS server succeeds the client initiates preparation of VM in the IaaS server and deploys the application environment[6] in the VM. After deployment, the client invokes the service `init_vm_ml` of the TTP from the user's VM in the IaaS provider by passing the measurement list of the VM to the TTP. The TTP verifies the measurement list with allowed values specified in the variable `vm_policy` that is passed to the TTP by the client during the invocation of the service `register_vm`. The client is notified if the verification fails in the future.

[6]Typically an application environment will involve the application that the user wants to host along with all the supporting applications

Algorithm 1 Registration of Controller

Input: *client*, *controller*, *vm*, *vm_policy*

1: **if not** *controller_list.contains(controller)* **then**
2:     *controller_ml = get_ml(controller)*
3:     **if** *verify_ml(controller, ml) = false* **then**
4:         *notify_client(CONTROLLER_VERIFICATION_FAILED)*
5:     **else**
6:         *controller_list.add(controller)*
7:     **end if**
8: **end if**
9: Extract policy parameters {S, N, T}
10: **for each** s in S **do**
11:     Store s in RIMM repository
12: **end for**
13: **return** WAIT_FOR_VM_INITIALIZATION

4.5.2 Verification

The verification is performed by the TTP in predefined intervals specified by the client. The scenario is presented in Figure 4.16.

The TTP, in predefined intervals specified by the client, invokes *get_ml* service (trust component) of IaaS server to get the measurement list of the server. The TTP verifies the measurement list against the known values of the IaaS server and notifies all the users that have registered with the IaaS controller, if the verification fails.

The verification of the VM is performed by the TTP by invoking the service *get_ml* of the virtual machine to retrieve the measurement list of the VM. The measurement list is verified against the valid measurement list specified by the client during registration. The user is notified when the verification fails.

Algorithm 2 presents the logic for the *verify_ml* function in TTP. The function takes as input the varable *target* (controller or VM) for verification and the measurement list, *ml* from the target. Each software binary represented in the variable *ml* is verified against known values through a loop counter (Line 1). The variable *hash_value* is extracted from the *ml* and stored in a variable (Line 2). If the variable *target* is a controller then the reference value for verification is retrieved from the global hash for that software binary and build, that is updated by the RIMM SaaS cloud (Lines 4 to 5). If the variable *target* is a VM, the variable *origin* for the software

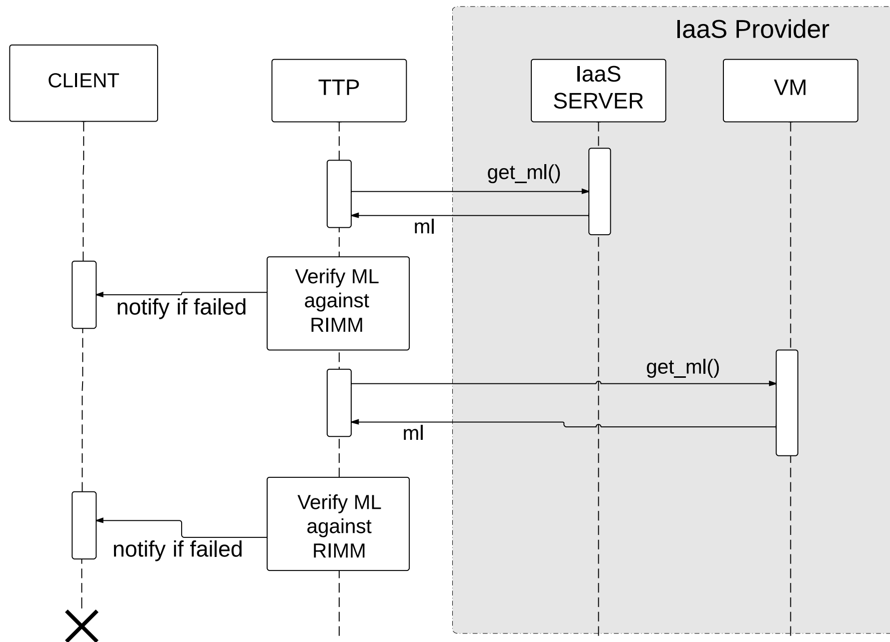

Figure 4.16: Verification Scenario

is checked in the RIMM repository of TTP. If the variable *origin* is `global` the reference value is retrieved from the global hash and if the variable *origin* is `client` the reference value is retrieved from the values provided by the client during registration (Lines 8 - 11). The variable $hash_value$ of the software is checked against the variable $reference_value$ and if they are not equal the notification options for that software binary is retrieved by the function $get_notification_options$ (Line 15). The client is notified based on the notification options (Line 16).

4.5.3 RIMM policy update

RIMM policy update scenario is presented in Figure 4.17. The TTP invokes the *subscribe* service of the RIMM Saas cloud by passing the software list as the parameter. the software list contains the identifier all the software binaries that the TTP is interested in getting the update information. The RIMM SaaS cloud adds the client to the subscription list of all the software denoted by the TTP. When a particular software in the software list gets an update the software

Algorithm 2 Verification of ML: *verify_ml*

```
Input: target, ml
1:  for each s in ml do
2:      hash_value = s.hash_value
3:      reference_value = NULL
4:      if is_controller(target) then
5:          reference_value = get_global_hash(s.id, s.build_info)
6:      else
7:          origin = get_origin(s.id, target)
8:          if origin is global then
9:              reference_value = get_global_hash(s.id, s.build_info)
10:         else
11:             reference_value = get_client_hash(s.id, s.build_info, target)
12:         end if
13:     end if
14:     if hash_value != reference_value then
15:         n = get_notification_options(s.id)
16:         notify_client(VERIFICATION_FAILED, s, n)
17:     end if
18: end for
```

vendors invokes the *notify_update* service of the RIMM SaaS cloud with the new hash value. This value is published to all the clients (TTP) that has subscribed to get the updates of that software. The TTP then adds that value to its datastore. Any future verifications by the TTP will use the new updated version value.

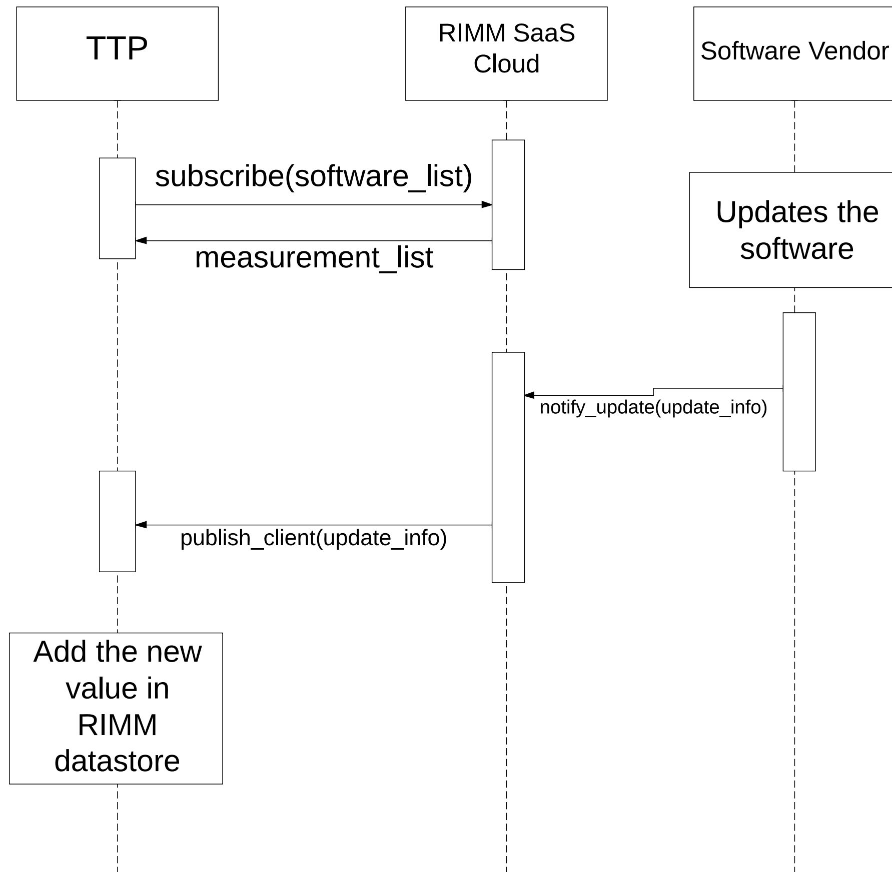

Figure 4.17: RIMM Update Scenario

4.6 Establishing a minimal trusted computing base

The Trusted Computing Base (TCB) consists of the hardware and software stack that is audited and tested thoroughly for the security of a particular application. For each software identified in TCB, the software need to be verified against valid values and all the updates of that software needs to be maintained by the system. Therefore, the size of the TCB is directly proportional to the complexity of attestation. Therefore it is necessary for the TCB size to remain small. Our goal is to establish a minimal TCB for the domain controller and the VM for effective attestation. We established minimal TCB using the Debian flavor[7] of GNU\Linux operating

[7]http://www.debian.org

system by leveraging Debian's advanced package management infrastructure, referred to as dpkg.

TCB using Debian Package Management (dpkg)

In the early days GNU\Linux programs were distributed as source code, expecting the users to compile and generate binary files. As the software ecosystem evolved considerably, the source code distribution was no longer feasible for effective software management as there are dependencies in the software and also all users cannot be assumed to spend time on compiling every program they install. To counter this, most GNU\Linux distributors use a pre-compiled and built collection of programs called packages that can be directly installed on their distribution. The Debian distribution of GNU\Linux uses dpkg to manage the software that can be installed in it.

A dpkg package file is a standard tar file, optionally compressed with gzip or bzip2 compression. The tar files consist of all the binary files and configuration files that are part of the software package along with a control file. The control file has meta-data about the package, such as the name of the package, version, architecture, list of dependent packages, maintainer etc. Along with these it has a priority that can contain the following values: *required*, *important*, *standard*, *optional* and *extra*. The required packages are essential for the execution of the operating system. Linux kernels, networking stacks are a part of required packages. The other class of packages are not of interest.

For our experiments we extracted the packages with priority, *required*, of the Debian. There were around 336 required packages in Debian Sid version. Each package has an average of 5 to 10 files. The hash values of each file are extracted and saved to form the RIMM SaaS cloud. We also installed the Xen VM machine and the supporting software for it and added to the measurement list in RIMM SaaS cloud.

If we assume the packages are secure then dpkg can be used to host the secure packages of the proposed infrastructure. Moreover, IaaS can utilize the dpkg sync and update functionalities to receive automatic updates from the dpkg repository.

4.7 Implementation

The TTP, server and client are implemented using the Twisted networking framework [46] in Python. Twisted is a platform for developing event driven networking applications. Twisted enables developers to implement non blocking servers using callback programming model. The services in all the servers use serialized Python objects (Pickle) to transfer messages in between them. Synchronized queues are used to regulate data flow during slow connections. The RIMM SaaS cloud and RIMM datastore of TTP use redis key/value database for storing the policy information.

4.8 Experiments

Experiments are conducted to analyze the effectiveness of the proposed infrastructure and to study the overhead. The effectiveness of the proposed infrastructure is illustrated by the Heartbleed bug scenario, where the clients are alerted when the server is running a vulnerable version of the OpenSSL software. The overhead of the system is measured against plain vanilla setup without remote attestation infrastructure.

4.8.1 Heartbleed Detection

On April 7 2014 a bug related to the OpenSSL cryptographic library, popularly referred to as Heartbleed, was made publicly disclosed. OpenSSL is a widely used implementation of Transport Layer Security (TLS) protocol. Heartbleed is buffer over-read security bug, that allows software to read more data than it is allowed. This vulnerability was propagated to all Debian machines through the OpenSSL version 1.0.1e-2+deb7u4 [5].

OpenSSL maintainers fixed this vulnerability and released a new version of OpenSSL. The Debian maintainers of the OpenSSL package released a new version 1.0.1e-2+deb7u5 with the updated OpenSSL. The servers that are executing OpenSSL are expected to update their version of OpenSSL with this new version.

For experiments, we established a TCB that contained the vulnerable OpenSSL u4. The following servers were setup based on the TCB:

- An RIMM server is deployed with the hash values of the vulnerable OpenSSL software binaries.
- An IaaS server is set up with software based on TCB, hosting the vulnerable OpenSSL binary.
- The TTP server was also deployed and the client was made the register the IaaS with the TTP.

The registration scenario (Section 4.5.1) is executed to register the IaaS for verification by the TTP. The registration scenario triggers the `subscribe` call by TTP to RIMM SaaS cloud for all the new software in TTP's RIMM datastore. At this point the RIMM datastore of TTP contains the vulnerable OpenSSL version and the verification cycle is successful as long as the software in IaaS server is not modified.

The hearbleed bug scenario is simulated and the workflow that follows the heartbleed detection is presented in Figure 4.18. When a new Debian package is released with updated binaries of OpenSSL, u5, `notify_update` of the RIMM SaaS is invoked by the software vendor with the variable *update_type*, `critical`. The RIMM SaaS updates its datastore with u5's hash values. The service `publish_client` of each TTP that has subscribed to the updates of OpenSSL is invoked by the RIMM SaaS with the variable *update_type*, `critical`. The TTP updates its RIMM datastore with new hash values. These hash values are used for verification in the next verification cycle of the IaaS controller and VM. If the controller or the VM is not running the new version of OpenSSL the client is immediately notified.

In this scenario the proposed system was successfully able to notify the client about the issue with the IaaS server.

4.8.2 Overhead

The overhead of the infrastructure is analyzed by comparing it with an IaaS server that does not run the remote attestation infrastructure. The booting overhead of the system and overhead during computational work is studied and analyzed.

For the experiments we set up two different infrastructures: Setup A and Setup B. Setup A, the remote attestation infrastructure, has three servers: IaaS server, TTP and the client. Setup

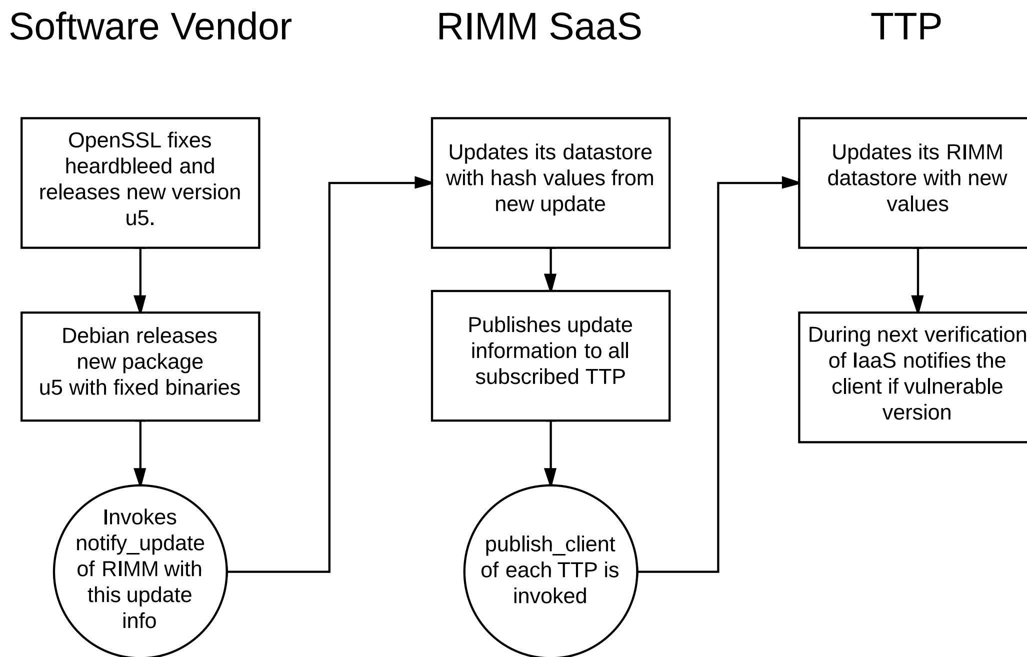

Figure 4.18: Heartbleed Detection Workflow

B without remote attestation has two entities: IaaS server and client. The IaaS servers of both A and B run the same version of the Debian operating system, Sid with kernel version 3.7.1. The kernel of setup A's IaaS is installed with IMA enabled kernel and the IMA kernel code is instrumented to log the time taken for hashing of each binary file that is loaded into the memory.

Booting

The booting process of IaaS server involved loading of 84 files to the memory. Both the IaaS in setup A and setup B are booted around 5 times to measure the average difference in booting. In our experiments setup A, that ran the IMA enabled IaaS took 1772ms longer to boot than the setup B without the remote attestation setup. This is because at the boot time many binary files are loaded into the memory for the first time, hence they need to be hashed before loading. In a complete boot process which takes around 30 seconds, 1772ms is negligible.

Computations

To measure the overhead on processing, Unixbench [104] was executed in the IaaS server on both Setup A and B. UnixBench executes a number of computationally intensive operations on the server and returns indicator values, that denote the performance of the system. Higher scores in Unixbench indicate better performance. The experimental output of both the operations are tabulated in Table 4.1. Setup A took 251.266s to complete the operations specified in Table 4.1 and Setup B took 252.521s.

Operation	Setup B	Setup A
Numeric Sort	813.44	836.32
String Sort	523.56	532.32
Bitfield	2.59E+08	2.59E+08
FP Emulation	275.8	275.76
Fourier	21142	18937
Assignment	26.544	26.576
Idea	5167.7	5165.8
Huffman	2340.3	2340.5
Neural Net	46.526	46.483
LU Decomposition	1267.7	1264.7

Table 4.1: Iterations per second for Setup A and Setup B

4.9 Limitations

We prototyped a remote attestation infrastructure and verified and validated the system through experiments. However, the infrastructure has several open issues.

Verifying the software for security vulnerabilities and malwares is non-trivial. An open source disk encryption software, TrueCrypt was commissioned for auditing in September 2013. It took two security engineers 6 months to complete the audit of a part of the software [61]. This experience show that auditing of all the software by RIMM SaaS cloud is an extremely challenging task.

Our infrastructure is based on static load-time measurements ("measure before load"). Inherently these measurements cannot completely predict the dynamic properties of the software. Once the software is loaded in the memory, there could be variety of reasons why the software

could fail or behave insecurely (e.g., buffer overflow). This behavior is not captured by static measurements as the behavior is also dependent on other unpredictable parameters. There is very little research on how to guarantee these dynamic properties of the software.

Also the proposed infrastructure is vulnerable to hardware attacks. There are known attacks on TPM [65, 19] that need to be resolved before TPM becomes truly tamper resistant.

4.10 Conclusion

In this chapter we showed it is feasible to create a remote attestation infrastructure that can force the IaaS to comply with the SLA and the clients can trust the remote attestation system. However, the infrastructure proposed by us introduces additional resources such as the TTP and the RIMM SaaS cloud. Someone will have to pay to keep these services running and typically the cost will be borne by the client.

The clients that already trust the cloud provider without the remote attestation provisions may find it is not in their interest to invest in the proposed infrastructure. This trust may arise from many factors such as the past record knowledge of strong SLA compliance of the provider with the client.

Hence there is a need for the infrastructure to take into account the subjective trust perception of the client and allow the clients to configure the remote attestation mechanisms based on the clients requirements. In further chapters we will explore how the infrastructures can be modified to adapt to this changing trust of client.

Chapter 5

Subjective and Dynamic Trust in SaaS

In Chapter 3 and 4, we presented architectures that allowed clients to transact with untrusted cloud providers. The architectures supported mechanisms such as searchable encryption and remote attestation, that induced cooperation and compelled compliance from the cloud service providers. Even though it was feasible to use these mechanisms in terms of security, these mechanisms have high overhead and require considerable investments in additional servers on the part of the cloud provider and client. Due to the overhead and additional investment some clients may not desire these mechanisms especially when they transact with trusted cloud provided. Therefore these mechanisms should be selectively targeted for clients that lack trust.

The client - server trust relationship decides the server configuration. However, trust relationships are complex. In Chapter 1 we discussed in detail the subjective and dynamic nature of trust in the real world by illustrating the plumber problem (Example 1) and the storage problem of the photographer Alice (Example 2). In Example 2, we noted a need for changing a server's configuration based on the client's requirements for effective policy deployment.

Policy mapping is the extraction of information from a policy to be used to configure elements of an underlying privacy management system in order to enforce the given policy. From Example 2, we observe that the level of trust in the cloud provider influences policy mapping mechanisms. In this chapter and Chapter 6, we continue exploring the influence of trust in privacy policy deployment in the cloud provider.

This chapter describes an architecture and algorithms for policy mapping that considers trust as a subjective and dynamic entity. In Section 5.1, we introduce an abstract model con-

taining the storage, processing and monitoring unit and describe different configurations for each unit based on trust level. We present the high level architecture of the cloud service framework and explain its individual components in Section 5.3. The implementation details are discussed in Section 5.5. The performance characteristics are studied in the Section 5.6 and finally we conclude in Section 5.8.

5.1 Configurable Elements

This section presents our proposed system model as a set of discrete components. The configurable aspects of each component are described. The clients decide on the configuration values based on their trust assessment of the CSP.

In this work we used three equivalence classes for the CSC's trust value of a CSP as follows: $(0, t_1), (t_1 + 1, t_2), (t_2, 1)$ ($t_1 = 0.25$ and $t_2 = 0.75$). The first class corresponds to "no trust", the second corresponds to "uncertain trust" and the third corresponds to "complete trust". Our definition of equivalence classes is similar to the trusted, distrusted and untrusted view of trust taken by Marsh et al. [75].

The configuration values based on the trust, are provided as input to mapping algorithms along with the data item and its privacy policy.

Each CSP consists of a storage engine, policy engine, monitoring engine and systems services.

5.1.1 Storage Engine

The storage engine is responsible for storing client data. Configurable elements include the encryption and logging levels.

Encryption Levels: The storage engine allows client data to be stored unencrypted or encrypted. One of the configuration parameters represents the type of encryption to be used which includes the following:

- **Unencrypted**: The data is not encrypted before being sent to the CSP. This is suitable when the client has a high level of trust in the CSP e.g., the trust level is 1.

- **Obfuscated**: A lightweight encryption (e.g., substitution cipher) is applied to the data before it is stored in the file system. This may be suitable when the client's trust in the CSP is medium to high e.g., the trust level is between 0.5 and 1.

- **Server Encryption**: Encryption (e.g., AES) is used for encryption of data before it is stored. This may be suitable when the client's trust in the CSP is medium e.g., the trust value is 0.5.

- **TPM Enabled Encryption**: A TPM (cryptographic co-processor) is used for encrypting the data.[1] This may be used when the client's trust in the CSP is low to medium e.g., the trust value is between 0 and 0.5.

- **Client Encryption**: The data is already encrypted by the client before reaching the storage system. This may be used when the client has no trust in the CSP e.g., the trust value is 0.

Logging Levels: The level of logging used to generate access logs on each read and write by the storage engine for a data item is also configurable. These access logs may be inspected by the client to assess if there have been any policy violations. If the trust level is low, detailed logs need to be generated for effective auditing, where as when the trust level is high, minimal logging should be sufficient. The detailed logging can contain the identifier of the entity making the data operation request, the IP address that the request originates from, timestamp, for what purpose, the server integrity value, etc. Minimal logging may only consist of the IP address that the request originates from.

5.1.2 Policy Engine

A data item is associated with a P-RBAC policy. These policies are translated into machine readable representations that can be interpreted by the policy engine. For each data operation request, the policies are checked by the policy engine to determine if the requested data operation can be carried out. The configurable elements include execution and logging levels.

[1]The client encrypts the data using AES, and encrypts the AES key using a binding key provided by the TPM and sends both the encrypted data and encrypted key to the CSP. The CSP decrypts the AES key using TPM and decrypts the data. The TPM can decrypt the key only when the server is running an uncorrupted version of the middleware.

Execution Environment: The policy engine supports the following types of execution:

- Trusted Co-processor: A cryptographic co-processor such as a Trusted Platform Module (TPM) is used to execute the Policy Engine. TPM ensures that the policy engine is not tampered with (using integrity values [34]). TPM also makes sure that the application environment does not run any malware. A TPM could be used when the trust level is low e.g., trust level is 0 to 0.5

- Traditional application environment: The normal application stack is used to run the policy engine. The could be used when the trust level is high e.g., trust level is 1.

Logging Levels: This is similar to that of the Storage engine.

5.1.3 Monitoring Engine

The monitoring engine (ME) inspects the logs generated by the policy engine and the storage engine and raises alerts as needed. The monitoring engine is designed as a reactive measure against a privacy breach. This engine can also be configured to monitor the other system services, such as the file system services and the network traffic. The logs read by the monitoring engine are useful for auditing and cyber forensics [24].

Execution Environment: The configuration arguments specified for the policy engine also apply to the monitoring engine, as it is a computational component of the middleware like the policy engine.

Log Storage: The logs read by the monitoring engine can be saved in one of the following locations:

- Within the CSP in an unencrypted format. This may be suitable when the trust in the CSP is high e.g., trust level is 1.

- With a TTP (Trusted Third Party). This may be suitable when the CSP's trust is uncertain e.g., trust level is 0.5.

- Client specific logs can be pushed back to the client. This may be suitable when the CSP is somewhat trusted e.g., trust level < 0.5.

5.2 Configurations

The configuration levels are specified by clients based on their trust. The configuration (C) is a tuple defined as follows:

$$C = (p, s, m) \tag{5.1}$$

The possible values that can be assigned to p, s and m are presented in Table 5.1 and were described earlier (Section 5.1).

Configuration Parameters	
p (Policy Engine)	r (type of run) l (log options)
s (Storage Engine)	e (encryption type) m (memory required) l (log options)
m (Monitoring Engine)	ev (event) l (log storage option)

Table 5.1: Configuration Parameters

The configuration parameters and the privacy policy (P-RBAC policy) are input from the client. The client uses the trust level it has in the CSP to determine the values of the configuration parameters.

The configuration details enable the CSC to express the desired setting in the server. Dynamic trust allows users to adapt to the changing perception of trust. As the trust opinion of CSC changes over time, the configuration of the services also changes with trust values. This may cause a change in the configuration. In this work we assume three configurations that are associated with a CSP that is distrusted, a CSP with uncertain trust and a trusted CSP. The configuration associated with a distrusted CSP is used when $E_x \in [0, 0.25]$. The configuration associated with a CSP with uncertain trust is used when $E_x \in (0.25, 0.75]$. The configuration with a CSP that is trusted is used when $E_x \in (0.75, 1]$ (The equivalence classes represented here are explained in Section 5.1).

5.3 Architecture

This section presents our proposed architecture (Figure 5.1) designed based on the configurable elements described in Section 5.1 and the formalisms explained in the previous section. The architecture has the following servers: Application Server, Storage Server, Crypto Server and Monitoring Server.

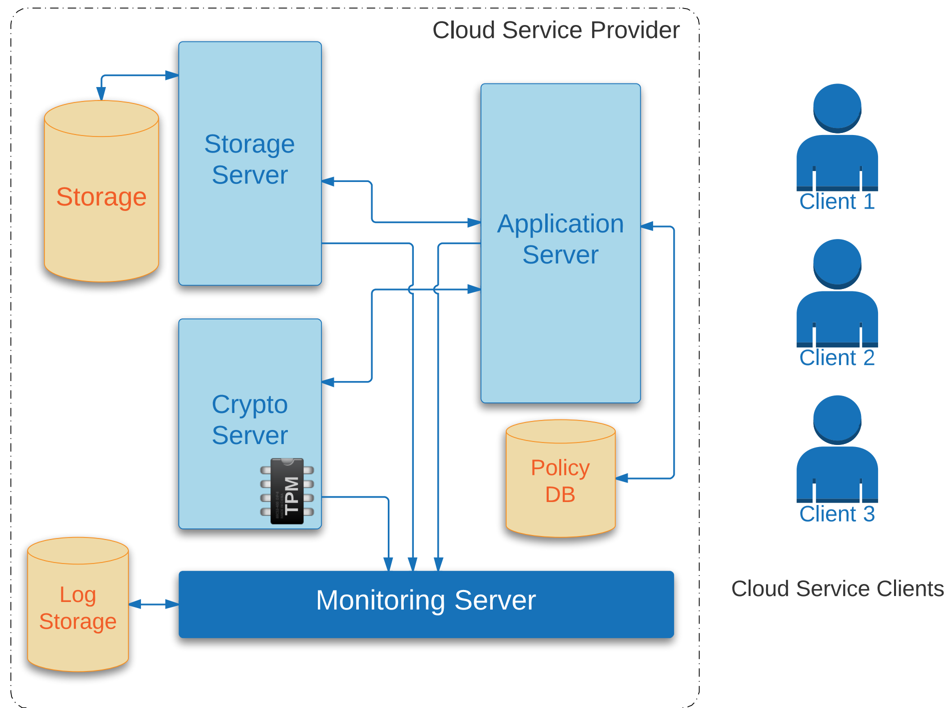

Figure 5.1: System Architecture

The CSC transacts with CSP through web services exposed by the application server. The application server uses the crypto server for encryption and decryption of the data and the storage server for the storage of the data. The monitoring server, collects the logs for all the servers and stores them for diagnostics in the future. The individual servers and the services offered by them are detailed below.

5.3.1 Application Server

The application server is responsible for providing service endpoints for the CSC to interact with the CSP. The following services are provided by the application server:

put (data, policy, config_info)

This service is called by the CSC, to store the `data` in the CSP. The `policy` parameter specifies the P-RBAC policy assignment details and the `config_info` parameter is the configuration tuple for the policy, storage and monitoring engine (from Section 5.2) for the data item. The application server stores the policy and the configuration details in the Policy database.

The storage component of `config_info` parameter contains the encryption requirements for the data. Based on the encryption requirement, the `data` is encrypted or obfuscated using the crypto server or sent in plain text to the storage server.

This service returns `data_id` which is used by CSC for data retrieval.

get (data_id)

When a `get` request is received from a CSC, the application server, checks if the CSC has access to the data, denoted by `data_id` in the Policy database. The data is read from the storage server. If the data was encrypted initially during the `put` operation, it is decrypted using the crypto server and the data is returned to the client.

get_tpm_quote ()

This service is invoked by the client to get the TPM quote [106] of the CSP to remotely verify if the CSP is tampered with malware (remote attestation). A quote is a value shared by the CSP with the CSC to prove that CSP is not running any malware or programs that the CSC does not trust. This service invokes Crypto server's *get_tpm_quote* to get the current TPM quote. TPM quote contains the integrity values[2] of the software that is executed in the server along with a binding key (k_b) that is bound to the integrity values. The usage of k_b is discussed in detail in Section 5.4.

[2]Integrity values are software binaries' computational hashes, representing a unique build of the software. If the software is tampered with, the integrity values of the software changes.

update (data_id, policy, config_info, key_{client})

This service is called by CSC when there is a change in the CSC's trust perception of CSP. This service needs `policy` and `config_info`, such as in the *put* web service. The configuration and policy changes are updated in Policy database. The data corresponding to `data_id` is transformed using the encryption type specified in the `config_info`. The parameter key_{client} is the client encryption key. It is used only when the encryption level needs to be changed from client_encryption. The usage of this service is explained in the migration scenario of Section 5.4.

delete(data_id, is_shred)

The CSC invokes this service to delete the data item from the server. The application server removes all the policy entries from the policy database corresponding to the data_id and invokes the *delete* service of storage server. The `is_shred` parameter is explained in the storage server's service description below.

5.3.2 Crypto Server

The crypto server is a streaming encryption decryption server with a TPM. It does not have any storage capability. It receives the data to encrypt/decrypt as a byte stream along with the key. It supports both server encryption and obfuscation of data. After performing the operation it does not store the data and key. The following services are provided by the crypto server:

crypt_init key (enc_type, operation, key, data_id)

This service is used to initiate any cryptographic operation. The `operation` parameter denotes if it is an encryption or decryption operation, `enc_type` specifies if it is server encryption (AES) or obfuscation, `key` denotes the symmetric key used for encryption encryption and `data_id` specifies the identifier of the data.

crypt (data)

The `data` as a byte stream is passed to this service and after `crypt_init` is invoked. This service encrypts or decrypts the data as specified in `crypt_init` invocation and returns the result as a byte stream. Invocation of this service without invoking `crypt_init` is invalid.

If the `operation` is *encryption*, the result is sent to the storage server. If the `operation` is *decryption*, the result is sent to the application server. The reason for this is explained in Section 5.4.

get_tpm_quote ()

As explained in Section 5.4.1, this service is invoked by Application Server's `get_tpm_quote` to get the TPM quote of the server.

5.3.3 Storage Server

The storage server is responsible for optimally storing the data in a storage pool. The data can be either encrypted data or plain text data. The storage server is oblivious to the policy and security requirements of the data. The services offered by the storage server are the following:

put(data, data_id)

This service saves the data specified by the `data_id` in its storage system and returns true after storing. If the storage is unsuccessful owing to disk space or disk error, the function returns false.

get(data_id)

This service fetches the data specified by the `data_id` and then return the data to the application server.

delete(data_id, is_shred)

This service deletes the data specified by `data_id` from its storage system. If `is_shred` is specified as true, the data from the storage system is securely erased to make it unrecoverable.

This is achieved by using *shred*[3] command in Unix.

5.3.4 Monitoring Server

The monitoring server provides services for the other servers in the architecture to store the logs. It also monitors the network traffic and file accesses for any unauthorized access. Monitoring server also implements a publish subscribe model for the CSC. The CSC can subscribe to alerts regarding data access and reject requests from the application engine through monitoring server.

5.3.5 Configuration of Servers

The dynamic trust value calculated is used to configure the individual servers in the architecture. The individual configuration of each server is tabulated in Figure 5.2.

	Application Server	Crypto Server	Storage Server	Monitoring Server
0	policy_info and config_info are encrypted	-not used-	no configuration	Logs are encrypted
0.5	policy_info and config_info are encrypted	Server encryption of data TPM encryption of data	no configuration	Logs are encrypted and file monitoring is enabled
1	policy_info and config_info are not encrypted	-not used-	no configuration	Logs are stored as plaintext

Figure 5.2: Configuration of the servers

[3]Shredding does not ensure that the any backup of the data in the system is deleted. We only delete the master file.

When the trust level is 0 (no trust), the `policy_info` and `config_info` are encrypted and stored in the application server. The Crypto server is not used, as the data is encrypted by the client. The monitoring server encrypts all the logs. With uncertain trust (0.5), the `policy_info` and `config_info` are encrypted and the data is encrypted either using server encryption or TPM encryption as specified by the client. The logs are encrypted and file monitoring is enabled in this mode. When the trust level is 1 (complete trust), the application server stores the `policy_info` and `config_info` as plain text and cryptoserver is not used for data encryption as the data is stored in plain text. The monitoring server also stores the logs in plain text. It has to be noted that the storage server does not have any special configuration specific to the trust level as the encryption details are abstracted from the storage server. The storage server sees the data as BLOBS (binary large objects) and saves it in storage server's filesystem.

5.4 Scenarios

In this section we describe the orchestration of storage and crypto servers by the application server for the following operations performed in the CSP.

5.4.1 get_tpm_quote

`get_tpm_quote` is requested by the client to remotely verify the server to ensure CSP's server is not executing any malware (remote attestation). The initial request is made to the application server, which makes the same request to Crypto server to fetch the integrity value and the binding key k. The binding key ensures that the key becomes invalid, when the server's integrity value changes in between a transaction. Both the integrity value and the binding key k are forwarded to the CSC. The CSC checks if the integrity value is valid and proceeds only if it is valid. For secure communication with the TPM, CSC encrypts the data using a new key, k_1 using symmetric encryption such as AES, $SEnc_{k_1}(data)$, and it applies asymmetric encryption, to the key k_1 using the binding key k, $AEnc_k(k_1)$. The crypto server can decrypt the data only if the integrity values bound to k do not change during decryption.

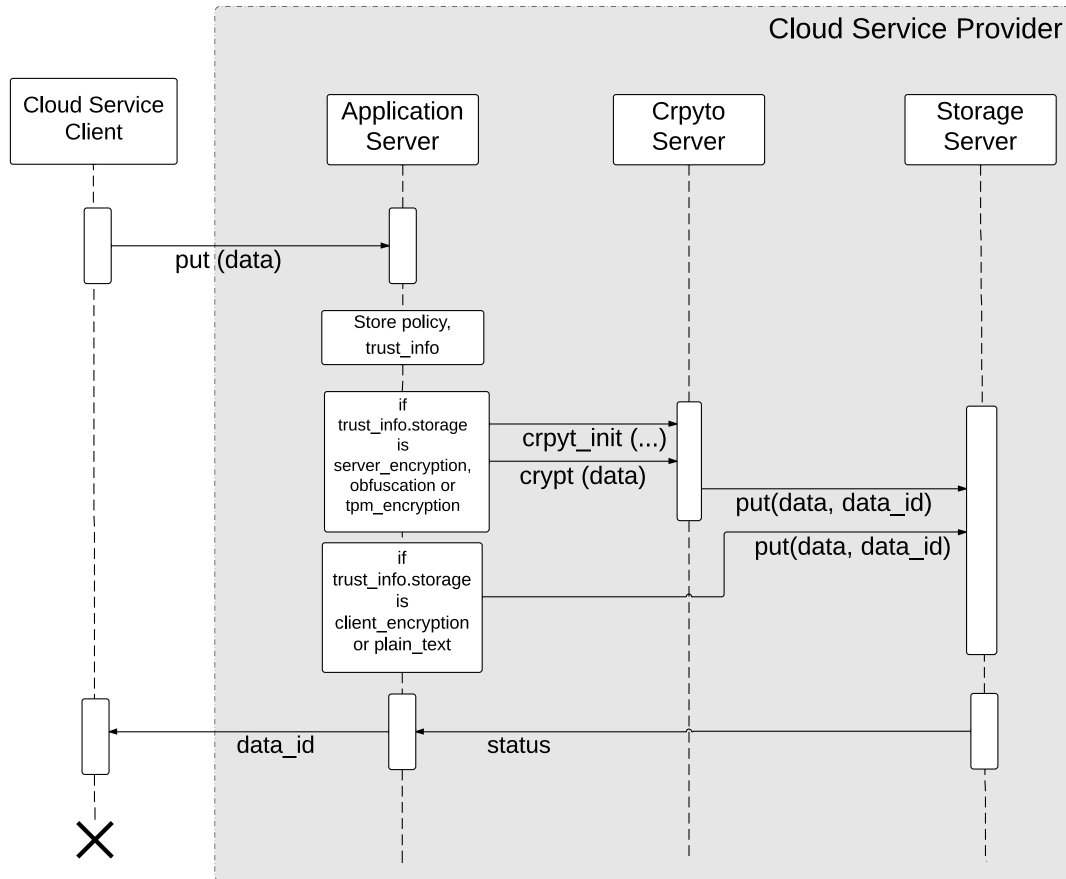

Figure 5.3: PUT

5.4.2 put

The *put* operation is invoked by the CSC to save a data item in the server. Figure 5.3 presents the use case using the *put* scenario. If the CSC wants to remotely attest the server before the transaction, CSC carries out the *get_tpm_quote* scenario described in Section 5.4.1. The application server on receiving the `put` request, saves the policy and config_info in its policy database. If the encryption requested by the CSC is either *server_encryption*, *obfuscation* or *tpm_encryption*, the application engine sends encryption messages to crypto server for encryption. The crypto server encrypts the data and sends the `put` request to the storage server for saving the data. If the encryption requested by the CSC is *plain_text* or *client_encryption*, data is sent directly from application server to storage server. The storage server return boolen `status` to the application server. On successful storage, the application server, returns the

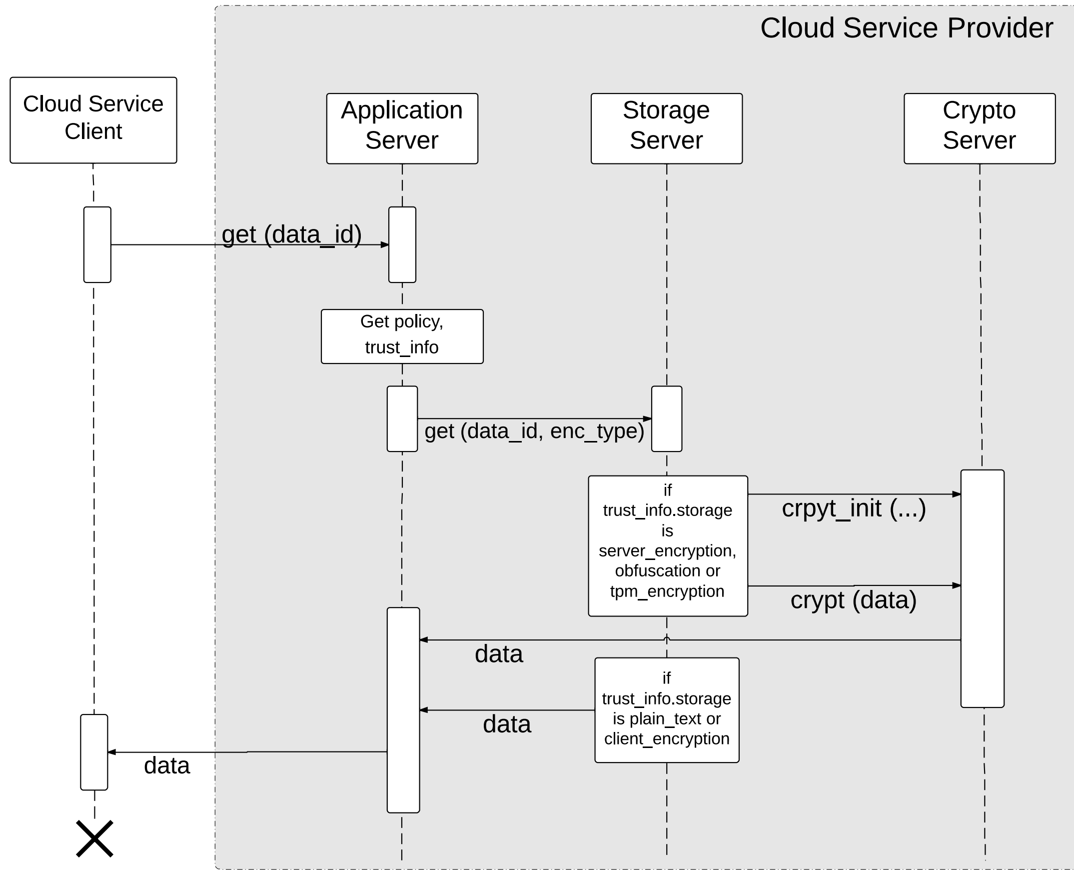

Figure 5.4: GET

`data_id` to the CSC.

5.4.3 get

The *get* service is invoked when the CSC wants to retrieve the data saved using *put* service. The `policy` and the `config_info` corresponding to the `data_id` is loaded from the policy database. A get request is sent to storage server, along with encryption type. The storage server sends the data to the crypto server for decryption if the `enc_type` is *server_encryption*, *obfuscation* or *tpm_encryption*. The crypto server decrypts the data and sends it to the application server. If the *enc_type* is *plain_text* or *client_encryption*, the data from the storage server is directly sent to the application server. The application server then sends the data to the CSC.

5.4.4 migrate

This service allows the CSC to change the configuration settings of the stored data item. This is invoked when there is a change in the CSC's trust perception of CSP. The algorithm for the migration to different trust levels for individual data items are described below. The CSC needs to execute these algorithms for every data item stored in the CSP.

Migration to trust level - 0

At trust level zero, encryption type is changed to *client_encryption* as the CSC no longer trust the CSP with the data. The CSC executes Algorithm 3, to ensure safe migration of data to the new trust level. The input to the algorithm include the data identifier, *data_id*, the new P-RBAC policy for the data, *policy*, and $config_info$, representing the new configuration for the data in the server.

Algorithm 3 Migrate to Trust Level: 0

Input: *data_id*, *policy*, $config_info$

1: $get_tpm_quote()$
2: $data = get(data_id)$
3: $delete(data_id, is_shred = true)$
4: $enc_data = client_encrypt(data)$
5: $put_data(enc_data, policy, config_info)$

Before initiating migration, CSC remotely attests the server state using *get_tpm_quote*. After verifying the server's state, the data is fetched by the CSC using the *get* operation. The data is deleted from the server using $is_shred = true$. The *is_shred* attribute will ensure safe removal to data so that it becomes unrecoverable. The data is encrypted by CSC locally using a secret key only known to client and the data is sent back to CSP with the new *policy* and $config_info$ objects. Algorithm 3 is executed for all the data items stored by the CSC in the CSP.

Migration to trust level - 0.5

Migration to 0.5 trust level changes the encryption to *tpm_encryption* or *server_encryption* based on the client's preference. The migration can be done from a lower trust level 0 or a

higher trust level 1. The CSC executes Algorithm 4 for the migration of a data item to the new trust level 0.5.

Algorithm 4 Migrate to Trust Level: 0.5

Input: $data_id$, $policy$, $config_info$

1: $get_tpm_quote()$
2: **if** $current_trust_level = 1$ **then**
3: $update(data_id, policy, config_info)$
4: **else if** $current_trust_level = 0$ **then**
5: $update(data_id, policy, config_info, k_{client})$
6: **end if**

When migrating from trust level 1, the data is already available in plain text in the server. Hence the *update* call (Line number: 3), initiates encryption using using the crypto server and the encrypted data is saved back in storage server.

When migrating from trust level 0, the client key, k_{client} is passed as the parameter in the *update* service (Line number: 5) of the application server. This key is used to decrypt the data and then the data is encrypted back by the server using a new key, known only to the server and is saved in the storage server.

Migration to trust level - 1

Migration to trust level 1 indicates an increase in trust level to a complete trust in CSP. The algorithm for this migration is presented below .

Algorithm 5 Migrate to Trust Level: 1

Input: $data_id$, $policy$, $config_info$

1: **if** $current_trust_level = 0.5$ **then**
2: $update(data_id, policy, config_info)$
3: **else if** $current_trust_level = 0$ **then**
4: $update(data_id, policy, config_info, k_{client})$
5: **end if**

When migrating from 0.5, the *update* service (Line: 2) is called with the new *policy* and $config_info$ objects. The crypto server will decrypt the data and send it back to storage server for saving.

If the $current_trust_level$ is 0, then k_{client} is passed as the parameter in the *update* service of the application server. The crypto server decrypts the data on *update* call with k_{client}.

5.5 Implementation

The application server, storage server, crypto server and monitoring server are implemented using the Twisted networking framework [46] in Python. Twisted is a platform for developing event driven networking applications. Twisted enables developers to implement non blocking servers using callback programming model. The services in all the servers use serialized Python objects (Pickle) to transfer messages in between them. Synchronized queues are used to regulate data flow during slow connections. The application server uses MySQL database for storing the policy information. The storage server uses the CSP server's filesystem to store the files. The monitoring server subscribes to events in all the three servers and logs them in a MySQL database. Pynotify is used to monitor file activity in the filesystem.

The server encryption is performed using GCM (Galois/Counter Mode) [78] algorithm. GCM is an authenticated encryption cipher that uses AES-128 (Advanced Encryption Standard) for encryption and Galois Message Authentication Code (GMAC) [78] for authentication. GCM and GMAC are part of the NIST official standards for authenticated encryption. Encryption provides confidentiality, integrity and authentication provides assurance on the data (against tampering). GCM algorithm is implemented using the M2Crypto library for Python. M2Crypto is an OpenSSL wrapper for Python. In our experiments we found M2Crypto to outperform the popular PyCrypto library of Python.

5.6 Experiments

This section describes the experiments used to evaluate the performance of the scenarios under different storage configurations.

5.6.1 Performance of Put and Get

In this experiment we study the response time of *put* and *get* scenarios described in Section 5.4 under varying data sizes. The response time is the total time for completing a transaction starting with a request from the CSC to the CSC receiving the output from the CSP. Within a transaction, the data flow in between application, storage and crypto servers, the processing of

data in an individual server, and the latency in the data flow contribute to the overall response time for a single connection.

Figure 5.5 represents the response time of *put* and *get* of different storage modes and different data sizes. We observe that the plain text version of *put* and *get* has the best response time, as the data is not transformed in the server. The same scenario is applicable for client encryption, where the data is stored in the server without any modification. In this experiment, with larger files, Put(tpm) and Get(tpm) is a more expensive operation than Put(Enc) and Get(Enc), as Put(tpm) and Get(tpm) involve interfacing with the TPM which is a limited resource device.

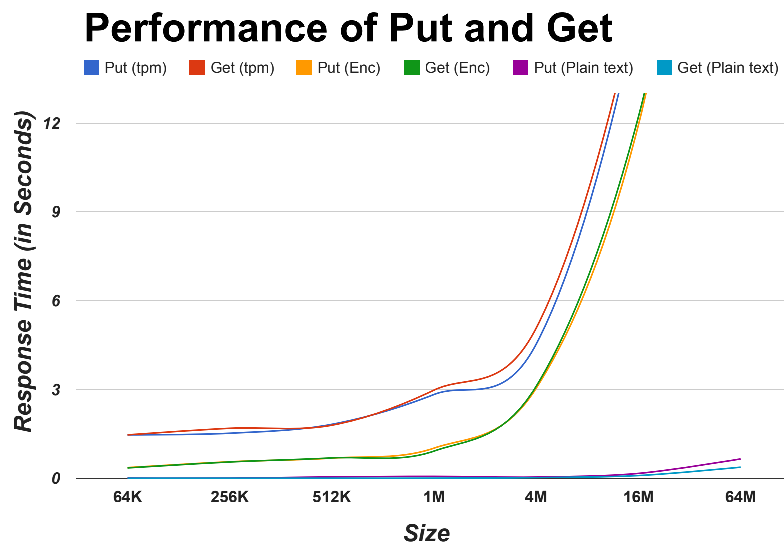

Figure 5.5: Put, Get Performance

5.6.2 Scalability Experiments

In these experiments, we study the behavior of the individual storage modes under concurrent connections, to understand the total number of requests, that the system can handle under acceptable response times. The number of concurrent requests are varied from 2 to 30 and we measure the average response time of requests at that given point of time. Each connection makes a *put* request of a 64KB file.

The plain text storage (Figure 5.6) deals with concurrency more effectively than other storage modes. The response time almost remains constant for up to 8 connections after which the

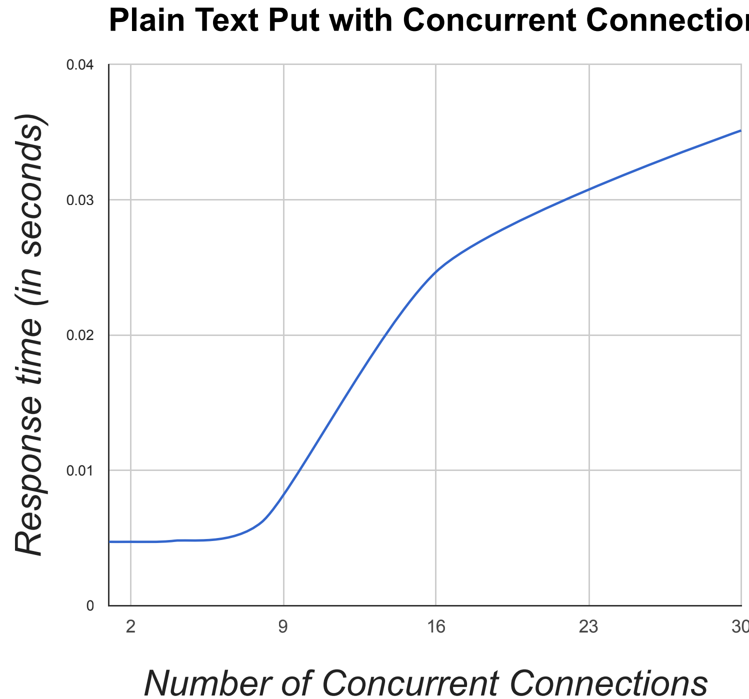

Figure 5.6: Plain Text with concurrent connections

response time starts to peak. Even with a load of 30 concurrent data requests, the response time stays well below 0.04 seconds.

With AES server side encryption (Figure 5.7), there is a linear relationship between response time and number of concurrent connections. We believe this is due to a bottleneck in the design of the Crypto server that makes the request handling serial. As encryption and decryption requires a lot of CPU cycles, in the current design we handle each encryption and decryption request serially. However, in faster hardware with multiple processors, parallel handling of requests with a scheduler will reduce the response time of put and get. This experiment still gives an insight into how the crypto server will perform in a uniprocessor environment with no parallel request handling.

TPM encryption degrades substantially under heavy load (Figure 5.8). TPM encryption is expensive. TPM is a uni-processor limited resource device. Even though, we use it only for protecting the symmetric keys that are used in the encryption of data, multiple usage within a short span of time appears to affect its performance.

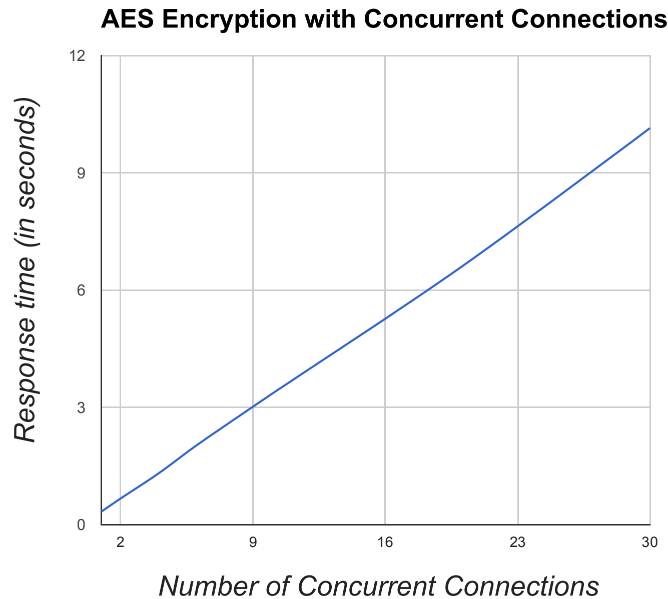

Figure 5.7: AES Encryption/Decrpytion with concurrent connections

5.6.3 Performance of Migrate

The migration scenario is executed by the CSC when there is a change in the CSC's trust perception of the server. To study this scenario we stored 10, 4MB files in the CSP using the *put* operation. All the different mode transitions described in Section 5.4.4 were executed on the stored files in CSP and their average respective response times are noted. In total 12 different scenarios are executed to study the cost in the mode transition. The results are graphed in Figure 5.9.

In the graph, 1-0, indicates the transition in trust level from complete trust 1 to no trust 0. Hence the transition will also involve change in storage mode from plain text to client side encryption. Similarly we observe the response time for the change from all the other trust levels. For uncertain trust, 0.5 two different storage modes: TPM encryption and client side encryption are also noted.

We observe that the change of client encryption to TPM encryption (0-0.5(tpm)) took the maximum time of 44.23 seconds. This is followed by TPM encryption to server encryption (0.5(tpm)-0.5(server)), server encryption to TPM encryption (0.5(server)-0.5(tpm)), client encryption to server encryption (0-0.5(server)). All the operations that involved TPM encryption

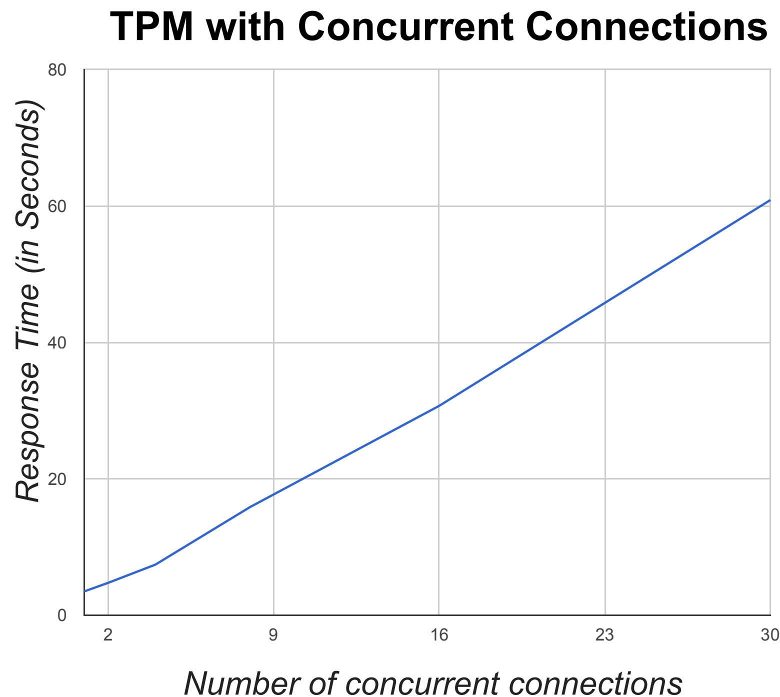

Figure 5.8: TPM with concurrent connections

or decryption were expensive.

5.7 Related Work

To the best of our knowledge, no existing work addresses the issue of subjective and dynamic trust in the context of privacy policy enforcement in Cloud Computing. Existing literature in privacy policy enforcement addresses trust models and the issue of trusted and untrusted cloud providers separately. We focus on state of the art in policy enforcement in untrusted providers in this section.

5.7.1 Trust Model

A trust model refers to the specification, evaluation and setting up of trust relationships among entities for calculating trust. Existing work studies trust modeling in specific systems such as P2P systems (e.g., [113]), ad hoc networks (e.g., [73, 31]), GRIDs (e.g,[81]), web services and

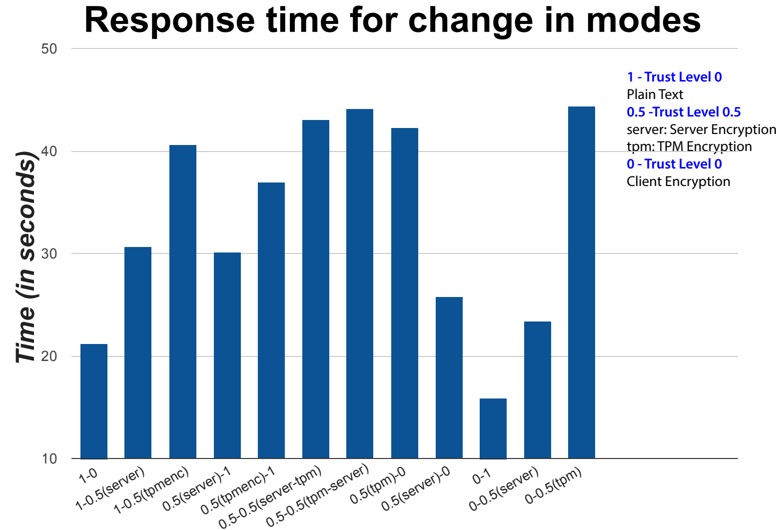

Figure 5.9: Response time for trust transition

component softwares [118]. The literature not only focuses on the factors that influence the trust (reputation, recommendation and trustor's past experience), but also studies the dynamic nature of trust. Balakrishnan et al. [31] proposed the use of a subjective logic in the context of a mobile ad hoc network. The factors that influence the trust of cloud computing model can be calculated using a trust middleware framework such as Yew et al. [120].

Liu et al. [73] proposed a dynamic trust model that is used for message routing in mobile ad hoc networks. Wang et al. [113] introduced a dynamic peer trust evaluation model for measuring the credibility of peer recommendations. Meng et al. proposed a dynamic grid trust model, DyGridTrust [81] that can evaluate trust dynamically by distinguishing between honest and dishonest recommendation. Wagng et al. [112] presented a quantifiable subjective trust evaluation approach for cloud computing model using randomness and fuzziness. Balakrishnan et al. [31] proposed a subjective logic [62] based trust model for mobile ad hoc networks.

Our work's focus is not on the calculation of trust. We assume that there is a trust model that will provide the client with trust values. These trust values will be used by the client to configure the services in the proposed architecture.

5.7.2 Policy Enforcement Mechanisms in Trusted and Untrusted Clouds

Privacy preserving storage in untrusted providers is achieved by one or more of the following methods in an untrusted cloud: i) Cryptography and ii) Trusted Platform Module (TPM). In cryptography, data owners (clients) encrypt data before the data is sent to the CSP. Therefore the data is completely hidden from the server. The key management is performed by the client. Pearson et al. [89] presented a privacy manager based framework in an untrusted cloud. This work investigates several obfuscation techniques to hide the data from the cloud provider. The Chaavi webmail system (Chapter 3) introduced an architecture for webmail systems, that supports privacy preservation in an untrusted cloud without compromising the functionality of the webmail.

Trusted platform module is a cryptographic co-processor chip developed using the specifications created by the Trusted Computing Group [106]. TPM has the following functionalities: Secure storage, platform integrity reporting, platform authentication and authenticated boot. TPM provides a secure execution environment within an untrusted CSP. This is achieved by a set of protocols that enables integrity checking of softwares running in the CSP along with secure encryption and decryption of data within TPM. Chen et al. [39] proposed solutions involving trusted computing based on remote attestation of software and hardware. Baldine et al. [32] proposed integrity based protection and access control. These attestation techniques are further refined using property based attestation which is done by hashing by Gupta et al. [54]. Bade et al. [28] developed a platform using TPM based on root of trust. Santos et al. [99] proposed a solution that uses a cloud monitor that stores the valid integrity values of all softwares that is certified to run on the CSP.

Cloud computing industry is dominated by Amazon, Dropbox, Google and Microsoft. Amazon S3 [1] offers a web service, that allows storage of objects in a collection abstraction referred to as a bucket. This allows clients to store data securely in the server with the option of storing keys both in the server and client. Dropbox is built over the Amazon S3 infrastructure and provides easy storage and synchronization services through a web client and a native OS client. Dropbox [3] offers encryption, but it stores the keys only at the server for accessing data. Dropbox needs to access the data in the server for data deduplication [57] to reduce the

file storage space. This makes Dropbox vulnerable to attacks such as, malicious insider attacks and hash value manipulation attacks [84]. To overcome this problem, the users can run client side encryption applications such as BoxCryptor [2] and Truecrypt [13] that encrypts the files and presents the encrypted file for dropbox to save it in their servers. Wuala [15] is a storage service like Dropbox, however it provides out of the box encryption before sending the files to the Wuala servers. Owncloud [11] is an opensource storage service that supports server side encryption and plain text storage.

Existing work in industry and academia makes an assumption that the trust value is not subjective and dynamic, and hence the same policy enforcement mechanisms is used for all the clients. Subjective and dynamic trust is studied for trust modeling by Balakrishnan et al. [31], Meng et al. [82] and Wang et al. [112]. The work proposed is distinct in that we take the multiple values of trust at a given point of time (the output trust value from the trust model) for a single CSP into consideration for configuring the services. Each of the work discussed above do not change its configuration dynamically based on clients trust requirements.

5.8 Conclusion

In this chapter we discussed the concept of subjective and dynamic trust in cloud computing and prototyped a working system. The prototype performed encryption in different modes and we studied the performance for the same.

We considered an abstract model with computational (policy engine), storage (storage engine) and monitoring unit. However a CSP will have multiple servers and when this model is extended across multiple servers, there are several resource management challenges and opportunities.

A storage service was implemented to validate our model. Other testing scenarios would include complex workflow scenarios such as a banking or health care management systems.

In the next chapter we consider the subjective and dynamic trust in an IaaS setup.

Chapter 6

Subjective and Dynamic Trust in IaaS

In this chapter we continue exploring the subjective and dynamic characteristics of trust with respect to IaaS providers. Chapter 5 described the mapping of policies to system configurations based on different levels of trust, with respect to Software as as Service (SaaS) cloud provider. Adhering to the same principle of policy mapping, in this chapter we will present architectures and algorithms that enable IaaS platforms to configure their services based on the client's trust perception.

Chapter 4 presented a notification based remote attestation architecture that supported dynamic whitelisting, softwares that are allowed be executed in the server, along with an architecture design based on separation of concerns. In this chapter we expand the work done in Chapter 4 to selectively apply remote attestation mechanisms on client demands.

This chapter is organized as follows: Section 6.1 presents the requirements for an architecture to support trust relationships in IaaS by illustrating different scenarios. The different components of the proposed architecture are described in Section 6.2. Section 6.3 presents the algorithms and Section 6.4 describes the implementation details. Section 6.6 presents the state of the art and this chapter concludes with Section 6.7.

6.1 Supporting trust relationships in IaaS

An IaaS provider has multiple IaaS servers. Each IaaS server hosts a domain controller which in turn manages multiple virtual machines within that server. To support multiple clients with

different trust perceptions, the IaaS provider should be able to deploy remote attestation on domain controllers and virtual machines on demand, based on the client's requests. This on-demand provisioning of remote attestation on clients can lead to many challenges.

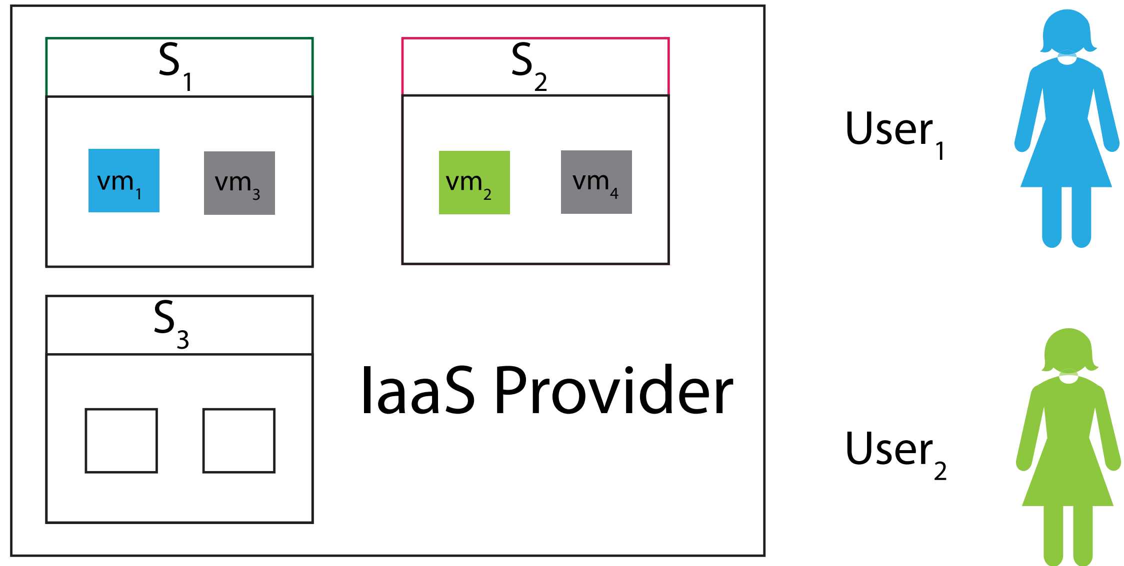

Figure 6.1: IaaS and Trust Relationship: Initial State

The problem is illustrated in Figure 6.1. Assume there are two users: $User_1$ and $User_2$. There is one IaaS provider with multiple servers (domain controllers): S_1, S_2 and S_3. $User_1$ owns vm_1 in S_1 and $User_2$ owns vm_2 in S_2. The user, $User_1$ trusts the provider, therefore remote attestation is not deployed in the server, S_1. The user, $User_2$ does not trust the provider hence server, S_2 has remote attestation mechanisms. The VM, vm_3 is owned by a client that trusts the provider and the VM, vm_4, is owned by a client that does not trust the provider. In this illustration, we assume that each server can host a maximum of two VMs for simplicity.

6.1.1 Scenario 1: $User_1$ moves from trusting the provider to distrusting

When $User_1$'s trust in IaaS deteriorates and $User_1$ starts to distrust the provider, the IaaS provider has one of the following options to improve the trust: a) the IaaS can deploy remote attestation in S_1 and vm_1; b) IaaS can migrate vm_1 to a remote attestation provisioned server and add remote attestation to vm_1.

With option (a), S_1 will continuously relay its measurement list to the trusted third party. There is an overhead in Server S_1 that may also affect vm_3's performance. However as vm_3 is owned by a user that already trusts the provider, they may perceive the additional monitoring as un-necessary overheard. Hence option (a) is not an ideal way of handling this.

Option (b) is to migrate vm_1 to S_2 which already has remote attestation. As S_2 has already the maximum number of VMs it can host (two VMs), S_3 is prepared for remote attestation and vm_1 is moved to S_3 (Scenario 1 of Figure 6.2).

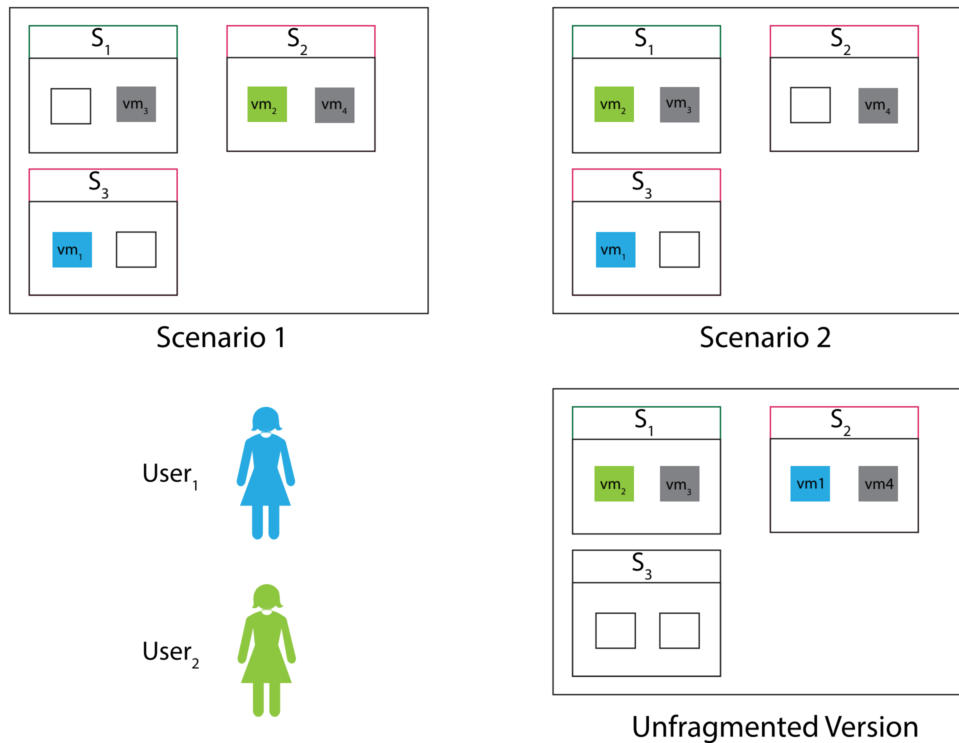

Figure 6.2: IaaS and Trust Relationship: Steps

6.1.2 Scenario 2: $User_2$ starts to trust the provider

When the user, $User_2$'s trust perception improves over time and $User_2$ starts trusting the IaaS provider, $User_2$ will no longer need remote attestation capabilities in vm_2 and S_2 (the server hosting vm_2). Hence remote attestation has to be disabled in vm_2 and S_2.

However disabling remote attestation in vm_2 can affect vm_4, as the owner of vm_4 relies on

the remote attestation for continuously monitoring S_2. Therfore vm_2 needs to be moved to a different server that does not support remote attestation.

The only server available without remote attestation capabilities is S_1. Hence vm_2 is moved to S_1. We can observe from Scenario 2 of Figure 6.2 the migrations from Scenario 1 and Scenario 2 has lead to fragmentations in the system. In an ideal system, the migration of VMs should have finally looked like the Unfragmented version in Figure 6.2.

6.1.3 Requirements

Based on these scenarios, we arrive at the following requirements for an architecture that deals with trust relationship in IaaS:

1. The architecture needs to support migration of VMs across different domain controllers (servers).

2. There should be a common middleware installed on all the servers that is able to enable/disable the support of remote attestation in domain controllers on demand.

3. Any server at a given point of time can either be hosting remote attestation enabled VMs or normal VMs. The server should not be allowed to host both the VMs simultaneously, as this may lead one of the VMs to suffer from unnecessary overhead.

4. The migration can result in fragmentation of VMs and the algorithms that are designed will have to effectively deal with the fragmentation.

6.2 Architecture

The proposed system contains the following components hosted within an IaaS infrastructure (Figure 6.3): Infrastructure Controller, Trusted Pools, Untrusted Pools and Unallocated Servers. The pool of unallocated servers enable the the IaaS provider to scale the trusted and untrusted pool based on demand. The infrastructure controller is responsible for allocating and deallocating resources to and from trusted pools, untrusted pools and unallocated servers. The

trusted pools, untrusted pools and unallocated servers are virtual organization of IaaS servers within the IaaS infrastructure.

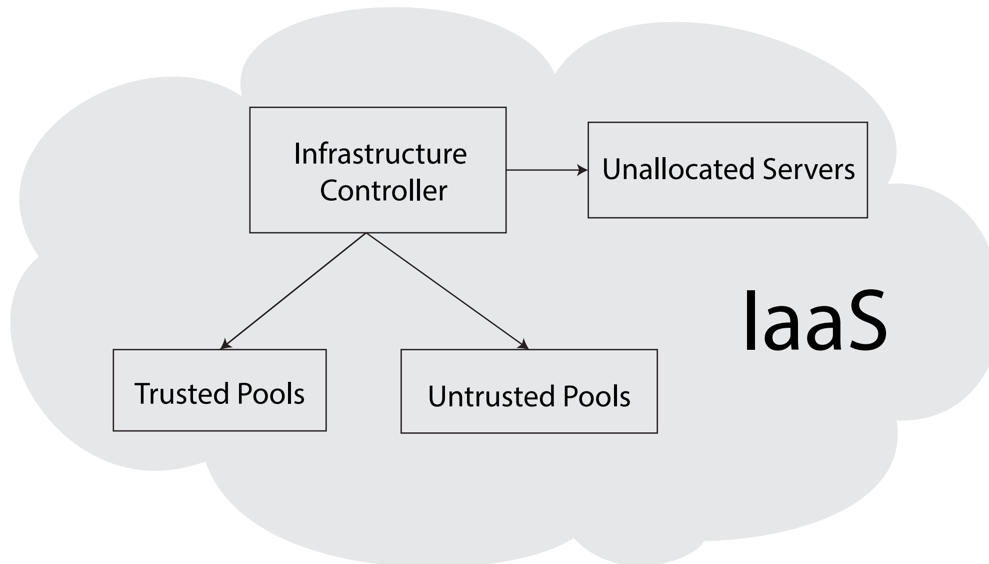

Figure 6.3: Components of the architecture

6.2.1 Infrastructure Controller

The infrastructure controller has algorithms (described in Section 6.3.2) that effectively manage resources between trusted pools, untrusted pools and unallocated servers. The resource allocation is performed on-demand from client requirements. The infrastructure controller contains an upper threshold limit for maximum number of servers that can be allocated for trusted pools (τ_t) and untrusted pools (τ_u). The values represented by the variables τ_t and τ_u determine the percentage of servers that is allocated to each pool ($\tau_t + \tau_u = 1$). For example, if the total number of resources are 100 and τ_t is 0.25 and τ_u is 0.75, the trusted pools, at a given point of time, cannot exceed 25 servers and untrusted pools cannot exceed 75 servers. The variable τ_t and τ_u values are decided by the IaaS provider based on several parameters such as demand for each type of pool, the profit margin for allocating a particular kind of resource etc. The details of this are beyond the scope of this work.

The infrastructure controller executes a controller middleware that is responsible for allocation and deallocation of servers in all the pools. To handle scenarios described in Section

6.1.1 and Section 6.1.2, the middleware also executes a periodical defragmentation algorithm that migrates the VMs and consolidates similar VMs together for optimum resource utilization.

The algorithms for allocation and deallocation along with defragmentation are presented in Section 6.3.2.

6.2.2 Trusted Pools

The trusted pools are formed by IaaS servers, that do not have remote attestation support. These pools are virtual organization of traditional IaaS servers. The IaaS servers within the pool do not communicate with each other and the controller decides which server is allocated to and deallocated from a trusted pool. The IaaS servers in the pool host the traditional hardware and software stack without any remote attestation support. The motherboards of the IaaS server need not be equipped with TPM (Trusted Platform Module). The software binaries loaded in these IaaS servers are not restricted by a limited trusted computing base. It has to be noted that a server equipped with TPM chip can be a part of trusted pool if the TPM chip is disabled and not used for remote attestation.

6.2.3 Untrusted Pools

The untrusted pools are a collection of IaaS servers that host the remote attestation enabled IaaS servers. These IaaS servers, similar to trusted pools, do not communicate with each other. The servers in the pools have a TPM chip and run IMA (Integrity Management Architecture) enabled kernels and the VMs also host IMA enabled OS. They also host the notification based remote attestation trust component discussed in Chapter 4. The services available through the trust component enables the client to remotely verify the domain controller (server) and the VM.

6.2.4 Unallocated Servers

The unallocated servers are a pool of resources that are waiting to be allocated by the controller. There are two types of servers in servers: unallocated servers that come with a TPM and servers that do not have TPM chip.

Un-allocated servers with a TPM chip can be used for both trusted (with the TPM chip disabled) and untrusted pools. The servers without a TPM chip can only be used for trusted pools.

The infrastructure controller goal is to increase the size of unallocated servers to reduce the resource and power consumption of the entire data center.

6.2.5 Resource Middleware

The resource middleware is the software component that is installed on all the servers and the virtual machines in the IaaS provider. This middleware, enables and disables remote attestation capability in the servers as per the demand. The resource middleware communicates with the controller middleware that is hosted by the infrastructure controller periodically to update the status of VMs running in the server. The controller middleware sends commands to resource middleware when a VM needs to be migrated or a server needs to be migrated.

6.2.6 Modifications to Trusted Third Party (TTP)

We described TTP and the services hosted by the TTP in Chapter 4. The TTP with the same functionality is used with the following additional modification. In the original TTP because the reference to a VM and IaaS server never changes (as the assumption was there is no migration between servers), the IP address of the VM and IaaS server can be used for identification. In this context an indirection table was introduced that stores the current IP address of the VM and IaaS server. Virtual machines and IaaS servers are identified by a GUID (Global Unique Identifier) in the TTP. The following service is added to TTP for supporting migration:

$update_server(vm, target_ip, target_server, RA)$:

This service is called by the IaaS infrastructure controller when there is a migration of VMs between different servers. The input variable *vm* represents the GUID of the VM, *target_ip* denotes the new IP of the VM, *target_server* represents the IP address of the server to which the VM is migrated to and *RA* denotes if remote attestation is enabled or disabled in the VM. The service changes the server associated with a client to a new server location, thereby updating the indirection table. The IP address of the VM changes to *target_ip*, when it is migrated.

This new server location (*target_server*) is used by TTP to check the server in the predefined intervals. The service is invoked when the client requests the migration to IaaS infrastructure controller or the infrastructure controller wants it for VM consolidation (defragmentation). For the given variable *vm*, the TTP updates its target IP in the indirection table to the new address.

6.3 Algorithms

This section presents algorithms that are used for migration of VMs. Section 6.3.1 describes the algorithms that enable migration in resource middlewares in IaaS server. The algorithms used by the infrastructure controller to manage resources are presented in Section 6.3.2.

6.3.1 Migration Algorithms in Resource Middleware

The migration algorithms presented in this section are executed within a resource through a middleware. The resource can be either a IaaS server or the VM. Algorithm 6 presents the algorithm for migrating a resource (change of pool of a IaaS server or migrating a VM) to the trusted or untrusted pool, specified by input variable *target_state*. The migration of the resource modifies the state of the resource from remote attestation enabled to remote attestation disabled or vice versa. This algorithm is executed by the the resource middleware that is hosted in the IaaS servers and VM. This algorithm is called when a resource needs to be moved to the target machine pool corresponding the input variable, *target_state*.

Algorithm 6 Prepare for migration: *prepare_for_migration*()

Input: *target_state*

1: *stop_all_jobs*()
2: **if** *target_state* != *current_resource.state* **then**
3:     *modify_boot_param*(*target_state*)
4:     *current_resource.state* = *target_state*
5: **end if**
6: **if** *current_resource.type* = *VM* **then**
7:     *halt_machine*()
8: **else**
9:     *reboot_machine*()
10: **end if**

The variable *target_state* can take two values `RA` and `NO_RA`, specifying remote attestation (untrusted pool) and no remote attestation (trusted pool) specifically. Before preparing for migration all the current jobs running in the resource are stopped (Line 1). If the target state of the resource (IaaS server or VM) is not the current state of the resource, then the boot parameters of the resource is modified to reflect the target. The resource's current state is modified to target (Lines 2-4) thereby moving the resource to the corresponding pool.

If the current resource is a VM, then the VM is halted so that it can be migrated to a new server. If the current resource is the IaaS server then the machine is rebooted, so that the machine on its next boot will be in the state specified by the variable, *target_state*.

Algorithm 7 Migrate VM: *migrate_vm*()

Input: *vm*, *target_state*, *target_server*

1: Invoke *prepare_for_migration*(*target_state*) in *vm*
2: *wait_for_vm_halt*(*vm*)
3: **return** *xen_migrate*(*vm*, *target_machine*)

Algorithm 7 denotes the algorithm for the migration of a VM. The input variable *vm*, represents the identifier of the VM and the input variable *target_state* denotes the target state of the VM and the variable *target_machine*, that denotes the machine to which the VM needs to be migrated to. Initially the algorithm calls the *prepare_for_migration* algorithm that prepares the VM for migration. It then waits for the VM to halt after that *xen_migrate* is called on the algorithm that migrates the VM to the new server.

6.3.2 Resource allocation algorithms in Controller

The resource allocation algorithm (Algorithm 8) is executed by the infrastructure controller for the allocation of the VM based on client's requests. This algorithm is invoked to allocate a new server from unallocated pool.

The algorithm takes as input the variable *server_type*. The variable *server_type* denotes the kind of server that needs to be allocated. It takes two values: `RA`, which represents remote attestation support and `NO_RA`, which represents no remote attestation support.

Initially the algorithm tries to find a server that has the ability to add a new VM (with free VM slots) for the given variable *sever_type* (Line 1). If a server is found with a free slot then

the server identifier is returned.

If a server with free slot is not found a new server needs to be prepared from the unallocated pool. If the requested *server_type* is RA then the variable *server_count* is set to S_u. The variable S_u is a global variable that denotes the total number of untrusted servers allocated. The threshold value denoted by τ is set to the threshold value of trusted servers, τ_t (Lines 3-5).

Similarly if the *server_type* is NO_RA, *server_count* is set to the total number of trusted servers allocated denoted by the variable S_t and the threshold value τ is set to the threshold value for untrusted providers τ_t (Lines 6-8).

The threshold value τ is assigned from either τ_t and τ_u (Lines 5 and 7) to enable verifying threshold limits independent of the type of the server (Line 9).

After setting the variable *server_count* and the threshold value, τ, there is a check to see if the allocation of new server will exceed the overall threshold limit (τ_t or τ_u) allowed for that type of server (Line 9). If it exceeds the new resource is not allocated.

If the allocation of new resource will keep the resources well within the threshold limit, then a new server with free slot (tpm server for *server_type*, RA and any server for *server_type* NO_RA respectively) is found from the unallocated servers. If there are servers available in the unallocated pool, then a new server is prepared by installing the OS and the required resource middleware. The OS is configured based on the variable *server_type* (Line 10). The corresponding global variable for the new server type is is incremented (Lines 10 to 11).

Migration of VMs

The migration algorithm for migrating a VM from one IaaS server to another IaaS server with the desired target state is presented in Algorithm 9. This algorithm is executed by the infrastructure controller. The input variable *vm* denotes the identifier representing the virtual machine and the variable *target_state* represents if the VM should be migrated to a RA or a NO_RA state.

Initially the algorithm tries to find a server with free slot of given *server_type* by invoking the function $find_server_for_vm()$ of Algorithm 8 (Line 1). If a server is not found, null is returned. If the server is found the VM is migrated by invoking the *migrate_vm* function call of Algorithm 7 (Line 5). The function *get_allocated_vms* returns the total number of VMs that are allocated in the specified server. If the migration is successful then the corresponding

Algorithm 8 Allocate VM: *find_server_for_vm*()

```
Input: server_type                                              ▷ RA or NO_RA
 1: server = find_server_with_free_slot(server_type)
 2: if server is null then
 3:     if server_type is RA then
 4:         server_count = S_u
 5:         τ = τ_u
 6:     else server_count = S_t
 7:         τ = τ_t
 8:     end if
 9:     if (server_count * τ) >= server_count + 1 then
10:         server = prepare_new_server(server_type)
11:         server_count = server_count + 1 ▷ Incrementing server count increments either S_u
    or S_t
12:     end if
13: end if
14: return server
```

variables in the infrastructure controller are incremented and decremented. The *source_vms* from the source machine in decremented and the *target_vms* are incremented (Lines 7-12). These variables will enables the infrastructure to find if there are any servers with free slots available for future allocations.

Algorithm 9 Migrate VM: *migrate_vm*()

```
Input: vm, target_state
 1: server = controller_of(vm)
 2: target_server = find_server_for_vm(server_type)
 3: if target_server is NULL then return NULL
 4: end if
 5: migration_status = migrate_vm(vm, target, target_server) of server
 6: if migration_status is TRUE then
 7:     source_vms = get_allocated_vms(server)
 8:     source_vms = source_vms - 1
 9:     set_allocated_vms(server, source_vms)
10:     target_vms = get_allocated_vms(target_server)
11:     target_vms = target_vms + 1
12:     set_allocated_vms(target_server, target_vms)
13: end if
14: return migration_status
```

Defragmentation Algorithm

The defragmentation algorithm migrates the VMs and consolidates similar VMs together for optimum resource utilization. A very primitive defragmentation algorithm is presented in Algorithm 10. The presented algorithm is a proof of concept of how the functions within the resource middleware can be orchestrated for VM consolidation.

This algorithm is invoked by the server in predefined intervals during a scheduled maintenance time. The algorithm takes as input the variable, *server_type*, which has the values `RA` or `NO_RA`. For each type, the algorithm is executed separately. This algorithm assumes that a server can host only a maximum of two VMs.

An IaaS server is referred to as completely filled if it hosts two VMs (the maximum, a server can host based on our initial assumption) and partially filled if it hosts only one VM.

Initially the list of all IaaS servers that are not completely filled with VMs (containing only one VM) are stored in the array $partially_filled_servers$ (Line 1). Each element in the array $partially_filled_servers$ represents a server object. The server object contains the identifier of the server (*id*), identifier of the server's VM (*vm*) and the current state of the server (*state*), `RA` or `NO_RA`. A pair of servers is considered by iterating through the array, $partially_filled_servers$ and the VM in the first server (i) in the pair is migrated to the second server ($i + 1$) in the pair. The migration is achieved by calling the function, *migrate_vm* (Algorithm 7) of the server denoted by i (Line 4). This step will make one server ($i + 1$) completely filled and the other server (i) is returned to the pool of unallocated servers. The service *update_server* is invoked in TTP to update the TTP of the migration (Line 5). At the end of the loop (Line 7), $partially_filled_servers/2$ number of servers will be completely filled and the remaining $partially_filled_servers/2$ servers will be unallocated.

The time complexity of this defragmentation algorithm is $O(n)$ because of the assumption that a server can host only a maximum of two servers. However, VM consolidation problems are similar to bin-packing problems and are NP complete problems [111]. Our work can be extended to support the VM consolidation algorithms [58] in the literature.

Algorithm 10 Defragment Virtual Machines: $defragment()$

Input: $server_type$

1: $partially_filled_servers = get_partially_filled_servers(server_type)$ ▷ Return all the servers hosting only one VM
2: **for** $i = 1$ to $partially_filled_servers.length$ step 2 **do**
3: **if** $i + 1 \leq partially_filled_servers.length$ **then**
4: ▷ Migrate the VM from i^{th} server to $(i+1)^{th}$ server, making i^{th} server return to unallocated pool
$migrate_vm(partially_filled_servers[i].vm,$
$partially_filled_servers[i].state,$
$partially_filled_servers[i+1].id$
) of $partially_filled_servers[i]$
5: $update_server$ in TTP
6: **end if**
7: **end for**

6.4 Implementation Details

As this is extension of the IaaS system described in Chapter 4, the TTP, server and client is implemented using the similar technology: Twisted networking framework [46] in Python. Migration of VMs was done using Python scripts by manually supplying commands to the Xen controllers. The infrastructure controller was implemented using python and Redis key value store for storing the datastructures. The datastructures stored the state of trusted, untrusted and unallocated pools.

Among all the migration capabilities available in Xen, a manual (script-based) Stop and copy migration is used. When a user starts to distrust the server, the user will be more concerned with immediately moving the VM to a remotely attested server rather than keeping the VM up and running during the migration, as it may cause security issues. Therefore the VM is stopped immediately after enabling the remote attestation, when the user requests remote attestation. The image is immediately moved to the server with remote attestation and the VM is booted again in the remote attested server.

6.5 Experiments

The functionality of remote attestation and the overhead was studied extensively in Chapter 4. We focused our experiments in this section to analyze the effectiveness of the proposed algorithms and to study the overhead of the migration. The algorithms were verified and validated by setting up a test environment and executing scenarios to test the system and the overhead of the migration was analyzed by measuring the time taken to migrate a VM from one IaaS server to the other IaaS server.

To study the effectiveness of the system, three different servers are used to verify the infrastructure, Server A, B and C. Server A was loaded with the infrastructure controller and the redis key value datastore for storing the datastructures. The server B and Server C were running the resource middleware in Debian operating system, Sid with kernel version 3.7.1. Server B hosted Sid Debian version with remote attestation enabled and Server C hosted the Debian OS with remote attestation disabled. Through the experiments we verified the migration of VMs between server B and server C as directed by server A.

The migration time of a 4GB VM image was found to be 10.3 minutes in 5 trials. This migration time can increase the time for defragmentation algorithms considerably as each migration for VM consolidation can potentially take 10 minutes. However this can be decreased by asynchronously initiating migration from the controller by not waiting for each migration to complete to start a new migration.

6.6 Related Work

Azzed et al. [27] introduced the concept of trust in resource management, such that allocation of a process to a server was made trust aware. Based on the security requirements of the process, the process is allocated either to a highly secure environment with overheads or a less secure environment with better throughput time. Tao Xie et al. [116] proposed a security aware scheduling algorithm, SAREC that integrated security requirements into scheduling for real time applications. The authors proposed a system model that consisted of list of tasks with security level and deadline requirements. Each security level in the system added additional

overhead to the running task. Each task is scheduled for execution in the highest security level possible (greater than the specified security level) as long as the deadline is not missed.

To the best of our knowledge no existing work addressed the issue of varying trust levels in an IaaS provider. There is considerable literature on realtime security-aware scheduling, however none in security-aware VM placement strategies in an IaaS provider.

6.7 Conclusion

We introduced the notion of subjective and dynamic trust for IaaS providers in this chapter. The architecture proposed in this chapter enabled on demand remote attestation provisioning on VMs based on the client's trust perception. We verified and validated the system through experiments. The clients can verify the migration through the TTP. However the system has several open issues.

The experiments found the migration time is in the order of 10 minutes for a 4 GB image which may be infeasible for certain kind of realtime applications. The VM consolidation will require a scheduled downtime of the server. To simplify the defragmentation algorithm we assumed that each server will host only a maximum of 2 VM images. In real world scenarios this can greatly vary.

Chapter 7

Conclusion

This dissertation has investigated the complex trust relationships of the cloud clients with cloud provider and proposed new architectures that take these trust relationships into account while enforcing a privacy policy. This chapter concludes the thesis by summarizing the contribution of the thesis in Section 7.1, presenting application scenarios in Section 7.2 and the future directions for the research in Section 7.3.

7.1 Contributions

This thesis focused on architectures that enable secure transactions between cloud clients and untrusted provider. We studied the feasibility of these architecture in a real world system using a Privacy Enhancing Technology (PET). We further investigated the limitations of PET and how Trusted Computing architectures (remote attestation) can be used to address these limitations. We identified issues with state of the art in remote attestation architectures and proposed improvements to it. Finally we introduced the concept of subjective and dynamic trust in the cloud computing context.

PET in cloud architectures: We studied the challenges in architecting a real world web-mail system using searchable encryption technology. The webmail system enabled secure communication of messages using a public/private key model and privacy preserving keyword search functionality using AES key encryption algorithm. The developed webmail system

used searchable encryption without compromising on the security and the functionality. The prototype is benchmarked and based on the results, we show the feasibility to architect a privacy preserving solution for webmail systems in a real working environment. However, the proposed solution addressed only searching, which is one of the many possible scenarios in a cloud computing environment.

Architectural Improvements to Remote Attestation: Remote attestation can be used in systems independent of the application scenarios. We studied the state of the art in remote attestation infrastructures in the cloud and identified two major issues: complexity in the management of software measurement values by the verifier (maintaining TCB) and the client's need to add verification workflow in its transactions. To address these issues, a remote attestation ecosystem of a Trusted Third Party (TTP), RIMM (Resource Integrity Measurement Manifest), Cloud Service Provider (CSP) and Cloud Service Client (CSC) was proposed. A notification based remote attestation infrastructure was prototyped that supported verification of servers and virtual machines using the TTP and RIMM. RIMM maintained a list of known secure software integrity values. The remote attestation of the server and VM was delegated to a TTP. The software state of the server and the VM was verified continuously by the TTP in predefined intervals through polling. The proposed remote attestation infrastructure was studied in a web application scenario. A known minimal trusted computing base (TCB) is established and its security implications and overhead of attestation were also analyzed.

Subjective and Dynamic Trust in IaaS and SaaS: Remote attestation introduces overhead on the system that may be undesirable to the clients that already trust the provider. Remote attestation may also be seen as an overhead if the clients start to trust the cloud provider. Therefore, there is a need for the infrastructures to adapt to the client's subjective trust perception of the client and allow the clients to configure the remote attestation mechanisms based on the clients requirements.

We studied the dynamic and the subjective nature of trust is a cloud computing context and its implications for SaaS and IaaS providers:

For the SaaS cloud, a policy based approach to the implementation of subjective and dy-

namic trust was architected to enable privacy policy enforcement in a SaaS cloud was proposed. An abstract model containing computational, storage and monitoring unit with configurable elements was introduced and algorithms that reflect how a change of trust influences the configuration of these elements were also designed.

In the IaaS context, the server pool is divided into virtual trusted and untrusted pools. The trusted pools supported the notification based remote attestation infrastructure whereas the untrusted pool does not have the overhead of the remote attestation infrastructure. We introduce secure protocols for the migration of VMs between the pools based on the client's trust perception and then benchmarked the system.

7.2 Application Scenarios

The contributions of this thesis can be applied in several application scenarios. Few of them are enumerated below:

- **Improvements to the traditional webmail:** Using a webmail system requires disturbing levels of trust from the cloud service client on the cloud service provider. Pretty Good Privacy (PGP) [122] is one of the widely used mechanisms for protecting the email messages from the attackers including malicious web mail servers to ensure only the recipients, who possess the secret private key can read the email. However PGP requires extra bootstrapping and installation of the PGP plugins on the webclient. Moreover encrypting email messages using PGP renders it un-searchable in the email server. Our contributions in Chaavi (Chapter 3) can be used to extend PGP by supporting secure email through browsers and also making it searchable using the mail server infrastructure. The lessons learned from Chaavi can also be used in implementing other types of PET in SaaS cloud architectures.

- **Verification of applications hosted on a minimal software stack:** Applications that are hosted on a limited software stack can be verified using contributions in Chapter 4. An example is a hospital management system software that runs over a known application framework and operating system stack. This hospital management system along with

the application framework and the operating system can be installed in a virtual machine (VM). A trusted third party (TTP) can be utilized to verify the software stack running on the VM and the VM controller. The services of the TTP can be utilized by multiple hospitals to host their own instances of the hospital management system. Regulators can control the approved software (software whitelist) that can be hosted on a VM and the VM controller. In general the architecture proposed in Chapter 4 can be used in any application architecture that uses a limited software stack, but requires strict monitoring.

- **Adaptive trust-based architectures:** Patterns of trust based service configuration can be observed in cloud industry. The clients work with untrusted services by deploying proactive mechanisms such as client side encryption. For example, in a storage system scenario, when the cloud service client trust the cloud provider, the clients use services such as Dropbox to store the content. However when the clients do not trust the provider, they execute client side encryption applications such as BoxCryptor [2] and Truecrypt [13] that encrypts the files and save them in dropbox. Using our contribution in Chapter 5, the cloud service clients can seamlessly configure the cloud service provider based on the client's trust perception without relying on multiple software and services. The configuration of the services can be dynamic based on the changing trust perception of the client over a point of time. This architecture can be extended to any application that requires subjective and dynamic configuration of services based on trust.

7.3 Future Work

There are number of areas the thesis can be taken in the future. These areas include:

- **Setting up of Government Regulations:** Even though trusted computing and remote attestation have been around in academia for 20 years, they are not widely adopted by the industry. The principle of remote attestation is used in the mobile industry by Apple and Google to maintain their private application market ecosystems and to prevent users from installing unauthorized software. More research should focus on the steps government can take to enforce the providers to use remote attestation. This can start by government

setting up regulations on the remote attestation protocols a cloud provider should support for third party verification.

- **Risk:** Much of the thesis focused on the trust concerns of the cloud clients. However, assessment of the risk, along with trust is another important tool that is used for making decisions in an uncertain environment [63]. The risk involved in choosing a cloud provider will also take into account the purpose for which a cloud provider is being used by the client. For example, the risk of the client in choosing a provider to store their personal photos can be relatively less than choosing the cloud provider for banking needs. The natural succession to this work is to focus on studying the risk factors affecting the cloud client and technological solutions to mitigate them and how risk and trust works together.

- **Remote Attestation Infrastructure:** The verification of the software against security vulnerabilities and malwares is non-trivial. The software measurement values (hash of the software) cannot accurately predict the dynamic behavior of the software when it is loaded in an unpredictable environment. Therefore the software needs to be tested in multiple environments before being passed as "secure". This process can be infeasible to adopt for each and every software. The effective auditing of a software against software vulnerabilities under dynamic environments needs to be studied in detail.

- **Platform as a Service:** This thesis focused on working with uncertain and changing trust in Software As A Service (SaaS) and Infrastructure as a Service (IaaS) cloud service models. Platform as a Service (PaaS) service models allow users to use the cloud server's Application Programming Interfaces (API) to implement their application. Future work can study the policy mapping of the subjective and dynamic trust to PaaS architectures.

- **Dataflow between different trust domains:** When there is dataflow in the system between clouds (federation of clouds) of different trust levels, the data needs to be translated for that configuration and the policy will have to be transported with the data. Moreover, when multiple clients are involved in this federation, each client will have its own trust view of the cloud service providers in the federation. Behavior and configuration of

services in the federation of clouds for multiple clients needs to be explored.

- **Effectiveness of Subjective and Dynamic Trust:** The architectures proposed by this work for enabling subjective and dynamic trust in cloud provider needs to be studied in a real world industry scenario to observe how clients react to these adaptive architectures. Detailed surveys along with user study will give deeper insight into the clients trust perception and will verify if technological means of addressing the trust issues gets reflected in the actual usage of the system.

The complex trust relationships in the social world will permeate into the computational arena as the technology evolves. Our attempt in this thesis was to address those complexities of trust relationships.

Bibliography

[1] Amazon S3.

[2] BoxCryptor (https://www.boxcryptor.com/).

[3] Dropbox (https://www.dropbox.com/).

[4] Forrestor Research: Growth of Cloud Computing.

[5] Heartbleed Bug Info.

[6] http://getfiregpg.org/s/home (Last accessed on June 23rd 2012).

[7] http://www.hanewin.net/encrypt/aes/aes.htm (Last accessed on June 23rd 2012).

[8] http://www.hanewin.net/encrypt/rsa/rsa.htm (Last accessed on June 23rd 2012).

[9] IDC Enterprise Panel: Survey 2009.

[10] OpenPTS User's Guide.

[11] OwnCloud.org.

[12] TCG IWG Integrity Report Schema Specification.

[13] Truecrypt (www.truecrypt.org).

[14] Trusted Computing Group - TNC Architecture for Interoperability Specification.

[15] Wuala Secuirty (Wuala.com).

[16] Xen vTPM Support.

[17] Client Specific TPM Interface Specification (TIS) Version 1.2. *Trusted Computing Group*, 2005.

[18] *vTPM: virtualizing the trusted platform module*, 2006.

[19] Security Failures in secure devices. In *Black Hat DC Presentation*, 2008.

[20] *Digital Privacy*. Auerbach Publications, November 2009.

[21] Fujitsu Research Institute: Personal data in the cloud: a global survey of consumer attitudes., 2010.

[22] *Intel® Trusted Execution Technology: Software Development Guide*. June 2013.

[23] AMD. AMD64 TechnologyAMD64 Architecture Programmer's ManualVolume 2: System Programming. pages 1–664, May 2013.

[24] Ali Reza Arasteh, Mourad Debbabi, Assaad Sakha, and Mohamed Saleh. Analyzing multiple logs for forensic evidence. *Digital Investigation*, 4:82–91, September 2007.

[25] Michael Armbrust, Ion Stoica, Matei Zaharia, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, and Ariel Rabkin. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, April 2010.

[26] Ahmed M Azab, Peng Ning, Emre C Sezer, and Xiaolan Zhang. HIMA: A Hypervisor-Based Integrity Measurement Agent. In *2009 Annual Computer Security Applications Conference (ACSAC)*, pages 461–470. IEEE, January 2009.

[27] F Azzedin and M Maheswaran. Towards Trust-Aware Resource Management in Grid Computing Systems. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)*, pages 452–452. IEEE, 2002.

[28] S A Bade, R Perez, Van Doorn, L.P., and H H Weber. Method for extending the CRTM in a trusted platform. Technical report, ˜2012.

[29] J. Baek, R. Safavi-Naini, and W. Susilo. Public key encryption with keyword search revisited. *Computational Science and Its Applications–ICCSA 2008*, pages 1249–1259, 2008.

[30] S Bajikar. Trusted platform module (tpm) based security on notebook pcs-white paper. *White Paper, Mobile Platforms Group–Intel Corporation*, 20, 2002.

[31] Venkat Balakrishnan, Vijay Varadharajan, and Uday Tupakula. Subjective logic based trust model for mobile ad hoc networks. In *the 4th international conference*, page 1, New York, New York, USA, 2008. ACM Press.

[32] I Baldine, Y Xin, A Mandal, P Ruth, A Yumerefendi, and J Chase. Exo-GENI: A Multi-Domain Infrastructure-as-a-Service Testbed. In *TridentCom: International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*, 2012.

[33] H Bar-El, H Choukri, D Naccache, M Tunstall, and C Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[34] B Bertholon, S Varrette, and P Bouvry. Certicloud: A novel tpm-based approach to ensure cloud iaas security. *Cloud Computing (CLOUD)*, 2011.

[35] S Bleikertz, S Bugiel, and H Ideler. Client-controlled Cryptography-as-a-Service in the Cloud. In *11th International Conference on Applied Cryptography and Network Security (ACNS'13)*, 2013.

[36] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Advances in Cryptology-Eurocrypt 2004*, pages 506–522. Springer, 2004.

[37] Andrew Brown and Jeffrey S Chase. Trusted platform-as-a-service. In *the 3rd ACM workshop*, page 15, New York, New York, USA, 2011. ACM Press.

[38] N Cao, C Wang, M Li, K Ren, and W. Lou. Privacy-Preserving Multi-keyword Ranked Search over Encrypted Cloud Data. In *IEEE INFOCOM*, 2011.

[39] P Chen. Software Behavior Based Trustworthiness Attestation For Computing Platform. *Journal of Software*, 7(1):55–60, 2012.

[40] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.

[41] T. Dierks. The transport layer security (TLS) protocol version 1.2. 2008.

[42] D Dolev and Andrew C Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, March 1983.

[43] Xue Dongliang, Wu Xiaolong, Gao Yunwei, Song Ying, Tian Xinhui, and Li Zhaopeng. TrustVP: Construction and Evolution of Trusted Chain on Virtualization Computing Platform. In *2012 Eighth International Conference on Computational Intelligence and Security (CIS)*, pages 623–630. IEEE, January 2012.

[44] Jan-Erik Ekberg and others. Mobile trusted module (MTM)–an introduction. 2007.

[45] Shawn Embleton, Sherri Sparks, and Cliff C Zou. SMM rootkit: a new breed of OS independent malware. *Security and Communication Networks*, 6(12):1590–1605, 2013.

[46] Abe Fettig and Glyph Lefkowitz. *Twisted network programming essentials*. O'Reilly Media, Inc., 2005.

[47] Keith Frikken and Mikhail Atallah. Privacy-Preserving Cryptographic Protocols. In *Digital Privacy*, pages 47–69. Auerbach Publications, November 2009.

[48] T. Garfinkel and M Rosenblum. When virtual is harder than real: Security challenges in virtual machine based computing environments. In *Proceedings of the 10th conference on Hot Topics in Operating Systems-Volume 10*, page 20. USENIX Association, 2005.

[49] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, Dan Boneh, Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. *Terra: a virtual machine-based platform for trusted computing*, volume 37 of *a virtual machine-based platform for trusted computing*. ACM, December 2003.

[50] Morrie Gasser, Andy Goldstein, Charlie Kaufman, and Butler Lampson. The Digital distributed system security architecture. In *Proceedings of the 1989 National Computer Security Conference*, pages 305–319, 1989.

[51] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[52] Eu-jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. Sirius: Securing remote untrusted storage. In *in Proc. Network and Distributed Systems Security (NDSS) Symposium 2003*, pages 131–145, 2003.

[53] Shafi Goldwasser and Silvio Micali. Probabilistic encryption and how to play mental poker keeping secret all partial information. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 365–377, New York, NY, USA, 1982. ACM.

[54] M R K Gupta, M R Pali, and S Singh. An Comparison with Property Based Resource Attestation to Secure Cloud Environment. 2012.

[55] Sven Ove Hansson. Risk. In Edward N Zalta, editor, *The Stanford Encyclopedia of Philosophy*. http://plato.stanford.edu/archives/spr2014/entries/risk/, 2014.

[56] D Harrison McKnight and Norman L Chervany. Trust and Distrust Definitions: One Bite at a Time. In *Lecture Notes in Computer Science*, pages 27–54. Springer Berlin Heidelberg, Berlin, Heidelberg, January 2001.

[57] Brian Hatch, James Lee, and George Kurtz. Hacking Linux Exposed: Linux Security Secrets & Solutions. *Computer*, 41(12):15–17, 2008.

[58] Yufan Ho, Pangfeng Liu, and Jan-Jan Wu. Server consolidation algorithms with bounded migration cost and performance guarantees in cloud computing. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 154–161. IEEE, 2011.

[59] D. Hubbard and Sutton M. Top Threats to Cloud Computing V1.0. *Cloud Security Alliance*, January 2010.

[60] J H Huh, J H Huh, J Lyle, J Lyle, C Namiluko, C Namiluko, A Martin, and A Martin. Managing application whitelists in trusted distributed systems. *Future Generation Computer Systems*, 27(2):211–226, 2011.

[61] iSecPartners. Open Crypto Audit Project: Truecrypt.

[62] Audun Jøsang. Subjective logic. *Draft book Available at: http://persons. unik. no/josang/papers/subjective_logic. pdf, visited*, 26, 2010.

[63] Audun Jøsang and Stéphane Lo Presti. Analysing the Relationship between Risk and Trust. In *Trust Management*, pages 135–145. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[64] E. Kangas and L.S. President. The Case for Email Security. *Published as a Lux Scientiae Article, available at http://luxsci. com/extranet/articles/email-security. html (accessed 1 May 2007)*, 2004.

[65] B Kauer. OSLO: Improving the security of Trusted Computing. In *Proceedings of the USENIX Security Symposium*, 2007.

[66] Lori M Kaufman. Data Security in the World of Cloud Computing. *IEEE Security & Privacy Magazine*, 7(4):61–64, 2009.

[67] Chunwen Li, Xu Wu, Chuanyi Liu, and Xiaqing Xie. An Implementation of Trusted Remote Attestation Oriented the IaaSCloud. In *link.springer.com.proxy1.lib.uwo.ca*, pages 194–202. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[68] Xiao-Yong Li, Li-Tao Zhou, Yong Shi, and Yu Guo. A trusted computing environment model in cloud architecture. In *2010 International Conference on Machine Learning and Cybernetics (ICMLC)*, pages 2843–2848. IEEE, June 2010.

[69] Joseph CR Licklider. Memorandum for Members and affiliates of the Intergalactic Computer Network. *Apr*, 23:350–351, 1963.

[70] Qian Liu, Chuliang Weng, Minglu Li, and Yuan Luo. An In-VM Measuring Framework for Increasing Virtual Machine Security in Clouds. *Security & Privacy, IEEE*, 8(6):56–62, 2010.

[71] Qin Liu, Guojun Wang, and Jie Wu. An Efficient Privacy Preserving Keyword Search Scheme in Cloud Computing. *Computational Science and Engineering, 2009. CSE '09. International Conference on*, 2:715–720, August 2009.

[72] Y Liu, L Tan, and Q Yi. A trusted network platform architecture scheme on clouding computing model. In *Computer Science and Information Processing (CSIP), 2012 International Conference on*, pages 890–892. IEEE, 2012.

[73] Zhaoyu Liu, A W Joy, and R A Thompson. A dynamic trust model for mobile ad hoc networks. In *Proceedings. 10th IEEE International Workshop on Future Trends of Distributed Computing Systems, 2004. FTDCS 2004.*, pages 80–85. IEEE.

[74] Thomas Margoni, Mark Perry, and Karthick Ramachandran. Clarifying Privacy in the Clouds. *SSRN Electronic Journal*, 2011.

[75] Stephen Marsh and Mark R Dibben. *Trust, Untrust, Distrust and Mistrust – An Exploration of the Dark(er) Side*, volume 3477 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[76] J M McCune, B. Parno, A Perrig, M K Reiter, and H Isozaki. An execution infrastructure for TCB minimization. 2007.

[77] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *2010 IEEE Symposium on Security and Privacy*, pages 143–158. IEEE, 2010.

[78] David McGrew and John Viega. The Galois/Counter mode of operation (GCM). *Submission to NIST. http://csrc. nist. gov/CryptoToolkit/modes/proposedmodes/gcm/gcm-spec. pdf*, 2004.

[79] P Mell. The NIST definition of cloud computing (draft). *NIST special publication*, 2011.

[80] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing. *National Institute of Standards and Technology, Information Technology Laboratory*, Version 15, 10-7-09:2, 2009.

[81] Wang Meng, Hongxia Xia, and Huazhu Song. A Dynamic Trust Model Based on Recommendation Credibility in Grid Domain . In *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*, pages 1–4, 2009.

[82] X Meng, G Zhang, J Kang, Honghui Li, and et al. A new subjective trust model based on cloud model. *Networking*, 2008.

[83] Daniele Micciancio. A first glimpse of cryptography's Holy Grail. *Communications of the ACM*, 53(3):96–96, March 2010.

[84] M Mulazzani, S Schrittwieser, M Leithner, M Huber, and E Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *USENIX Security*, 2011.

[85] Mohammad Nauman, Sohail Khan, Xinwen Zhang, and Jean-Pierre Seifert. Beyond kernel-level integrity measurement: enabling remote attestation for the android platform. In *Trust and Trustworthy Computing*, pages 1–15. Springer, 2010.

[86] Q Ni, E Bertino, J Lobo, and C Brodie. Privacy-aware role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 13:24, 2010.

[87] Douglas F Parkhill. *Challenge of the computer utility*. Addison-Wesley, 1966.

[88] Siani Pearson. Privacy, Security and Trust in Cloud Computing. In *Kompetenz, Performanz, soziale Teilhabe*, pages 3–42. Springer London, London, June 2012.

[89] Siani Pearson, Yun Shen, and Miranda Mowbray. A Privacy Manager for Cloud Computing. In *Proceedings of the 1st International Conference on Cloud Computing*, pages 90–106, July 2009.

[90] Ronald Petrlic. Integrity Protection for Automated Teller Machines. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 829–834. IEEE, 2011.

[91] Martin Pirker, Ronald Toegl, Daniel Hein, and Peter Danner. A PrivacyCA for Anonymity and Trust. *link.springer.com*, 5471(Chapter 7):101–119, 2009.

[92] RA Popa, JR Lorch, D Molnar, and HJ Wang. Enabling Security in Cloud Storage SLAs with CloudProof. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, page 31. USENIX Association, 2010.

[93] J. Postel. RFC821: Simple mail transfer protocol. Technical report, 1982.

[94] Karthick Ramachandran, Hanan Lutfiyya, and Mark Perry. Chaavi: A Privacy Preserving architecture for Webmail Systems. In *CLOUD COMPUTING 2011, The Second International Conference on Cloud Computing, GRIDs, and Virtualization*, page 133 to 140.

[95] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[96] Craig H Rowland. Intrusion detection system. Technical report, 6 2002.

[97] Jordi Sabater and Carles Sierra. Review on Computational Trust and Reputation Models. *Artificial Intelligence Review*, 24(1):33–60, September 2005.

[98] R Sailer, X Zhang, T Jaeger, and L van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. *USENIX Security Symposium*, 2004.

[99] N Santos, R Rodrigues, K P Gummadi, and S Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *USENIX Security*, 2012.

[100] B. Schneier. Snake Oil. Crypto-Gram Newsletter (http://www.schneier.com/crypto-gram-9902.html#snakeoil) [Online on 05th September 2011], 1999.

[101] Bruce Schneier. *Liars and Outliers*. Wiley, 2012.

[102] S P Shapiro. The social control of impersonal trust. *American journal of Sociology*, 1987.

[103] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared-memory. *ACM SIGARCH Computer Architecture News*, 20(1):5–44, 1992.

[104] Ben Smith, Rick Grehan, and Tom Yager. Byte-unixbench: A Unix benchmark suite. Technical report, 2011.

[105] Piotr Sztompka. *Trust: A sociological theory*. Cambridge University Press, 1999.

[106] TCG. Trusted Computing Group (TCG) and the TPM 1.2 Specification. In *Trusted Computing Group*, 2005.

[107] V. Toubiana, A Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy preserving targeted advertising. In *17th Annual Network and Distributed System Security Symposium, San Diego, CA, USA*. Citeseer, 2010.

[108] G W Van Blarkom, J J Borking, and JGE Olk. Handbook of privacy and privacy-enhancing technologies. *Privacy Incorporated Software Agent (PISA) Consortium, The Hague*, 2003.

[109] Luis M Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2008.

[110] D Wallom, M Turilli, and G Taylor. mytrustedcloud: Trusted cloud infrastructure for security-critical computation and data managment. *. . .)*, 2011.

[111] Meng Wang, Xiaoqiao Meng, and Li Zhang. Consolidating virtual machines with dynamic bandwidth demand in data centers. In *INFOCOM, 2011 Proceedings IEEE*, pages 71–75. IEEE, 2011.

[112] Shouxin Wang, Li Zhang, Na Ma, and Shuai Wang. An Evaluation Approach of Subjective Trust Based on Cloud Model. In *2008 International Conference on Computer Science and Software Engineering*, pages 1062–1068. IEEE.

[113] Y Wang and Vijay Varadharajan. Trust/sup 2/: developing trust in peer-to-peer environments. In *2005 IEEE International Conference on Services Computing (SCC'05) Vol-1*, pages 24–31 vol.1. IEEE.

[114] S.D. Warren and L.D. Brandeis. The right to privacy. *Harvard Law Review*, pages 193–220, 1890.

[115] Harold B. Westlund. NIST reports measurable success of Advanced Encryption Standard - News Briefs - National Institute of Standards and Technology - Brief Article. *Journal of Research of the National Institute of Standards and Technology*, 2002.

[116] Tao Xie, Xiao Qin, and A Sung. SAREC: a security-aware scheduling strategy for real-time applications on clusters. *2005 International Conference on Parallel Processing (ICPP'05)*, pages 5–12, 2005.

[117] Zhen Xu, Aimin Yu, and Wensi Yang. Real-time remote attestation of IaaS cloud. In *2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems (CCIS)*, pages 292–297. IEEE, January 2012.

[118] Zheng Yan. Security via Trusted Communications. In *link.springer.com*, pages 719–746. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[119] A.C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science*, pages 160–164. Citeseer, 1982.

[120] Chern Har Yew. *"Architecture Supporting Computational Trust Formation"*. PhD thesis, ir.lib.uwo.ca.

[121] A Yu and D Feng. Real-Time Remote Attestation with Privacy Protection - Springer. *Trust*, 2010.

[122] P.R. Zimmermann. *The official PGP user's guide*. MIT Press, May 1995.

Curriculum Vitae

Karthick Ramachandran

Education

University of Western Ontario London, ON, Canada

- Master of Science, Computer Science

  September 2007 - April 2009

Thangavelu Engineering College, University of Madras Chennai, India

- Bachelors in Engineering, Computer Science (First Class with Distinction)

  July 2000 - April 2004

Work History

- **Senior Software Developer** Aug 2014 - Present

  Polar Securities, Toronto, ON

- **Research Assistant/Teaching Assistant** September 2009 - Aug 2013

  University of Western Ontario, London, ON

- **Assistant Systems Engineer** November 2004 - July 2007

  Tata Consultancy Services Ltd., Pune, India.

Selected Publications

- On Subjective and Dynamic Trust for Privacy Policy Enforcement in Cloud Computing. Karthick Ramachandran, Hanan Lutfiyya and Mark Perry. In TSP 2013, Proceedings of IEEE HPCC 2013, November, 2013
- A Privacy Preserving Solution for Webmail Systems with Searchable Encryption. Karthick Ramachandran, Hanan Lutfiyya and Mark Perry. In International Journal On Advances in Security 5.1 and 2 (2012): 36-45. 2012
- Clarifying Privacy in Cloud Computing. Karthick Ramachandran, Mark Perry and Thomas Margoni. In Cyberlaws2011, Mar, 2011
- Decentralized Approach to Resource Availability Prediction using Group Availability in a P2P Desktop Grid. Karthick Ramachandran, Hanan Lutfiyya and Mark Perry. In Future Generation Computer Systems, October 2010