# A Framework For Secure Auditing Of Cloud

**A PROJECT REPORT**

*Submitted in partial fulfillment of the requirements for the award of the degree*

*of*
**BACHELOR OF TECHNOLOGY**
**in**
**COMPUTER SCIENCE AND ENGINEERING**

*Submitted by:*
**Himanshu Dogra, Sudhakar Verma**
**130001015, 130001035**

*Guided by:*
**Dr. Neminath Hubballi**


**INDIAN INSTITUTE OF TECHNOLOGY INDORE**
**November 2016**

# CANDIDATES' DECLARATION

We hereby declare that the project entitled **"A Framework For Secure Auditing Of Cloud"** submitted in partial fulfillment for the award of the degree of Bachelor of Technology in Computer Science and Engineering is an authentic work.

The project was supervised by **Dr. Neminath Hubballi**, Assistant Professor, Computer Science and Engineering, IIT Indore.

Further, we declare that we have not submitted this work for the award of any other degree elsewhere.

**Himanshu Dogra, Sudhakar Verma**
**Date:** 30 November 2016

# CERTIFICATE
# by BTP Guide

It is certified that the declaration made by the students is correct to the best of my knowledge and belief.

**Dr. Neminath Hubballi,**
**Assistant Professor,**
**Discipline of Computer Science and Engineering,**
**IIT Indore**

# PREFACE

Through this report, we have tried to give the detailed design on the architecture and implementation techniques of a Framework that would help Cloud Users to Securely Audit their Cloud Applications and therefore trust the Cloud Service Provider.

We have proposed a list of Service Level Agreements that would help a user ensure the Security and Integrity of his data and ensure Reliability on the Cloud Service Provider. We have also proposed a method to extend this trust to the hardware level with the use of Trusted Platform Module.

We have put our best efforts to explain the proposed framework in a lucid manner. We have also added the figures and screenshots to make setup, implementation and testing of the architecture more illustrative.

**Himanshu Dogra, Sudhakar Verma**

# ACKNOWLEDGEMENTS

We would like to express our gratitude towards Dr. Neminath Hubballi for constantly supporting and motivating us throughout the project and for always giving us new ideas and guiding us in the right direction.

We would also like to thank our families and friends for always being there with us and for being a constant source of motivation.

At last, we would like to thank all the contributors to The Xen Project and Linux Kernel, without whom this project would not have been possible.

**Himanshu Dogra, Sudhakar Verma**
**B.Tech IV Year**
**Discipline of Computer Science and Engineering**
**Indian Institute of Technology Indore**

# ABSTRACT

Using Cloud Services, users can remotely store their data and deploy applications and services on machines having superior resources as connectivity, storage or performance, without the hassle of local data storage and maintenance. This helps user reduce the cost on actual hardware and the costs that come on maintenance. However, since the users no longer have physical access of the hardware media containing the data makes data integrity and security a growing concern in Cloud Computing, especially for users with limited computing resources and knowledge of the cloud structure.

Ideally users should be able to use their data as if its is managed locally and should not waste much resources on regularly verifying the integrity. So a trusted Third Party Auditor(TPA) is required in order to maintain the Service Level Agreements (SLAs) agreed upon by the user and regularly apply them to the stored data in the cloud, such as the integrity is never tampered with. Introducing the third party between the user and the Cloud Service Provider (CSP) should bring in no new vulnerabilities towards user data security and privacy, and produce no additional online burden to user in terms of data handling or costs.

In this project, we propose a secure third party system supporting auditing. We propose a set of SLAs agreed upon by the User and the CSPs. We go on by discussing the methods in which these SLAs can be measured. We then show that these methods can be implemented as Daemons on the CSP and TPA side and the chain of trust of a running application in a Virtual Machine can be extended to the hardware level by the use of TPM module.

# Contents

# List of Figures

# Chapter 1

# Introduction

**Cloud Computing** is being widely used by the enterprises and single users alike. It has a long list of unprecedented advantages in the IT history: on-demand self-service, universal network access, location independent resource pooling, rapid resource elasticity, usage-based pricing and transference of risk. As a disruptive technology with vast implications, Cloud Computing is transforming the very nature of how businesses use information technology. The data is being centralized or outsourced to the Cloud. Storing the data remotely to the cloud is beneficial to both the user and the IT enterprises by relieving off the burden for storage management, universal data access with independent geographical locations, and reduction of capital expenditure on hardware, software, and personnel maintenances, etc.

While this technology is very convenient to the users, it has some serious concerns over the users' outsourced data. Since the Cloud Service Providers (CSP) are totally different entities with users having little or no control over them, the fate of the users' privacy depends on the trust between the two parties. This puts the security and the integrity of the user data at risk as explained in the following paragraphs. Even though the machines in the cloud are usually much more powerful than personal computing devices, they still face threats same as that of other devices both internal and external for data integrity and security.

A CSP may be motivated to to behave unfaithfully towards the cloud users regarding the status of their outsourced data. For example, CSP might reclaim storage for monetary reasons by discarding data that has not been or is rarely accessed, or even go to the extent of hiding this data loss so as to maintain a reputation. Also CSP might move the allocated machine to smaller machine if it sees the computing resources are rarely used for monetary gains. Thus, although outsourcing data to the cloud is economically attractive for long-term large-scale data storage, it does not immediately offer any guarantee on data integrity and availability. This problem, if not properly addressed, may impede the successful deployment of the cloud architecture.

Our Project aims at developing a framework for auditing the Cloud Data and Services that will ensure the integrity of the outsourced data. We propose a mechanism to measure Integrity, Security and Availability, and subsequently implement the required policies and protocols in the form of Service Level Agreements (SLAs) which are discussed in the later chapters of this report. We then extend the chain of trust of the running applications to the hardware level using Trusted Platform Module (TPM).

# Chapter 2

# Problem Statement and Design Goals

We consider a cloud data storage service involving three different parties: the cloud user or application master, who has data files to be stored in the cloud and subsequent applications to run in the cloud; the cloud server, which is managed by the Cloud Service Provider (CSP) to provide data storage service and has significant storage space and computation resources; the Third Party Auditor (TPA), who has expertise and capabilities that cloud users do not have and is trusted to assess the cloud storage service reliability on behalf of the user upon request. Users rely on the CSP for cloud data storage and maintenance. They may also dynamically interact with the CSP to access and update their stored data for various application purposes. To save the computation resource as well as the online burden, cloud users may resort to TPA for ensuring the storage integrity of their outsourced data, while hoping to keep their data private from TPA.

We consider the existence of a semi-trusted CSP. Namely, in most of time it behaves properly and does not deviate from the prescribed protocol of execution. However, for their own benefits the CSP might neglect to keep or deliberately delete rarely accessed data files which belong to ordinary cloud users. Moreover, the CSP may decide to hide the data corruptions caused by server hacks or hardware failures to maintain reputation. We assume the TPA, who is in the business of auditing, is reliable and independent, and thus has no incentive to collude with either the CSP or the users during the auditing process. However, it harms the user if the TPA could learn the outsourced data after the audit.

To answer the SLAs posed by the user and not affect the privacy and security of the data stored by the user in the cloud our architecture design should achieve the following security and performance guarantees:

1. To allow TPA to verify the correctness/integrity of the cloud data on demand without retrieving a copy of the whole data or introducing additional online burden to the cloud users.

2. To allow the TPA to detect the breach of trust and hence check if the platform is trusted for customer applications or data.

3. To allow TPA to collect data from the CSP in order to produce sufficient data points and subsequently cater to cloud users' SLAs on integrity, security, availability and consistency.

# Chapter 3

# Background

A typical Cloud Server infrastructure consists of a Physical Server running a Hypervisor with one or more instances of Virtual Machines running on top of it. Along with these, there may be a Trusted Platform Module which is physically available as hardware chip on the motherboard, which forms the root of trust at a hardware level. Some servers may use the virtual implementation of such chips in the form of vTPMs. The following sub-sections discuss each of these components in detail.

## 3.1 Hypervisor

A **Hypervisor** is a software component running directly or indirectly on top of the hardware. It is responsible for the management of different Virtual Machines running on a single Physical Machine. Broadly, the Hypervisors can be classified into the following categories:

1. **Native or bare-metal Hypervisors** are the ones that directly run on the hardware. Some examples include Xen and Oracle VM Server.

2. **Hosted Hypervisors** are the ones that run inside an Operating System as a Process. VMware Workstation, QEMU are the common examples.

For this project we have chosen Xen Hypervisor because of its immense popularity as an Open Source Bare-Metal Hypervisor with wide availability of documentation and support.

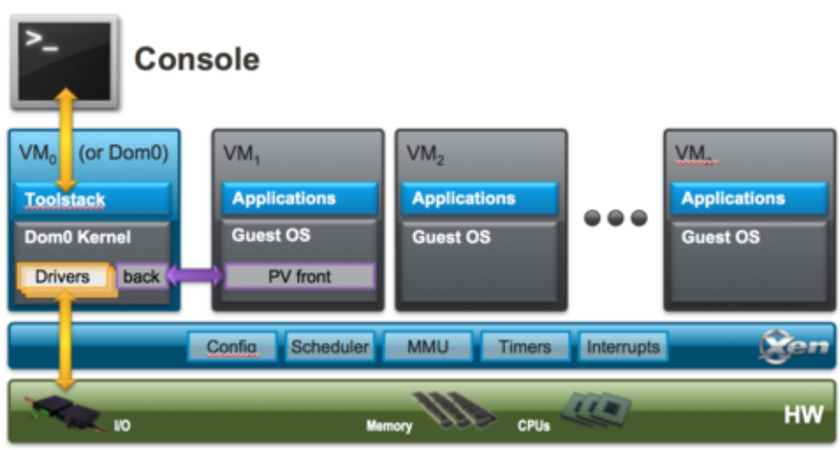


FIGURE 3.1: Xen Hypervisor Architecture

Figure 3.1 describes the architectures of Xen Hypervisor. It directly runs on the hardware and is responsible for handling CPU, Memory and interrupts. Various Virtual Machines(VMs)/domains run directly on top of Xen. The host domain (Dom0) consists all the device drivers and a control stack to manage other domains. Other domains (Guests) are denoted as DomU(s). Appendix A discusses the steps involved in setting up the Hyperviser and VMs.

## 3.2 Trusted Platform Module

A **Trusted Platform Module (TPM)** is a computer chip that is used to securely store artifacts to authenticate the platform (PC or laptop). It has the capability to store keys, certificates involved in integrity measurement of a machine at the time of boot. Along with this, it also provides some primitive cryptographic operations like random number generation, RSA key-pair generation and store hashes of the applications in a number of Platform Configuration Registers (PCRs). The only two operations supported on a PCR are Extend operation and Clear operation (on PCRs 16 to 23, used to reset the PCR value). Cryptographically, Extending a PCR P with a value v is defined in 3.1.

$$Extend(P, v) = SHA1(P \parallel v) \tag{3.1}$$

where $\parallel$ operation represents a concatenation of two byte arrays.

PCR extensions are used during the platform boot process and start within early-executed code in the Basic Input/Output System (BIOS) that is referred to as the Core Root of Trust for Measurement (CRTM).

Every TPM chip is uniquely identified by a built-in key, the Endorsement Key (EK), which is certified by the Device Manufacturer and stands for the validity of a TPM. Related to the EK are Attestation Identity Keys (AIKs). An AIK is created by the TPM and linked to the local platform through a certificate for that AIK. This certificate is created and signed by a certificate authority (CA). In particular, a privacy CA allows a platform to present different AIKs to different remote parties, so that it is impossible for these parties to determine that the AIKs are coming from the same platform. AIKs are primarily used during quote operations to provide a signature over a subset of PCRs as well as a 160-bit nonce. Quotes are delivered to remote parties to enable them to verify properties of the platform.

Once enabled in the BIOS, the host provides support for the TPM chip (eg. Kernel modules in Linux OS). One can interact with the TPM chip using the commands from tpm-tools or can use Trousers library to programmatically interact with the chip.

## 3.3 Virtual Trusted Platform Module (vTPM)

A vTPM is a user-mode process which provides TPM support to a running VM. The VM interacts with this vTPM in a same way it would with a hardware TPM. Multiple instances of vTPMs can exist on a Physical Machine with a single hardware TPM, thus providing each Guest VM with its own vTPM that it can use.

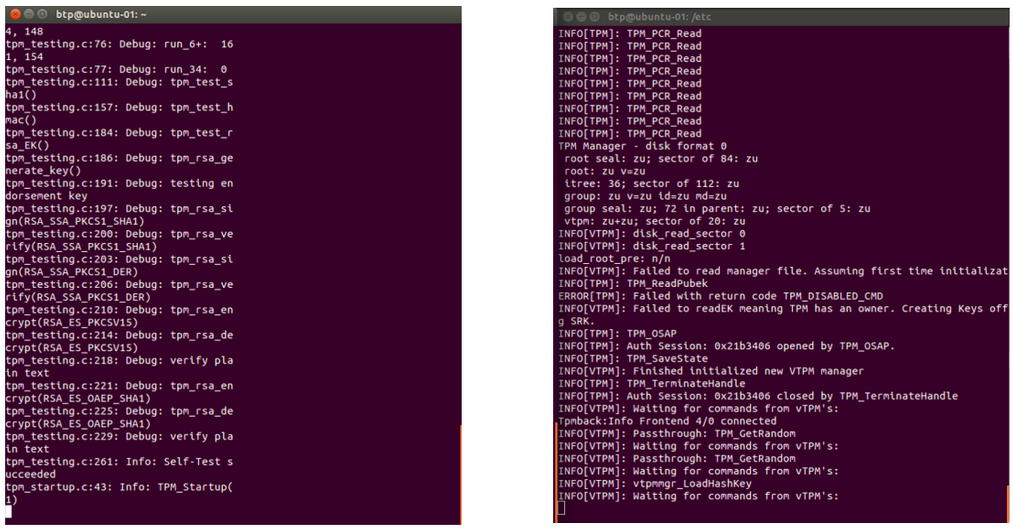


FIGURE 3.2: vtpmmgr and vTPM interaction

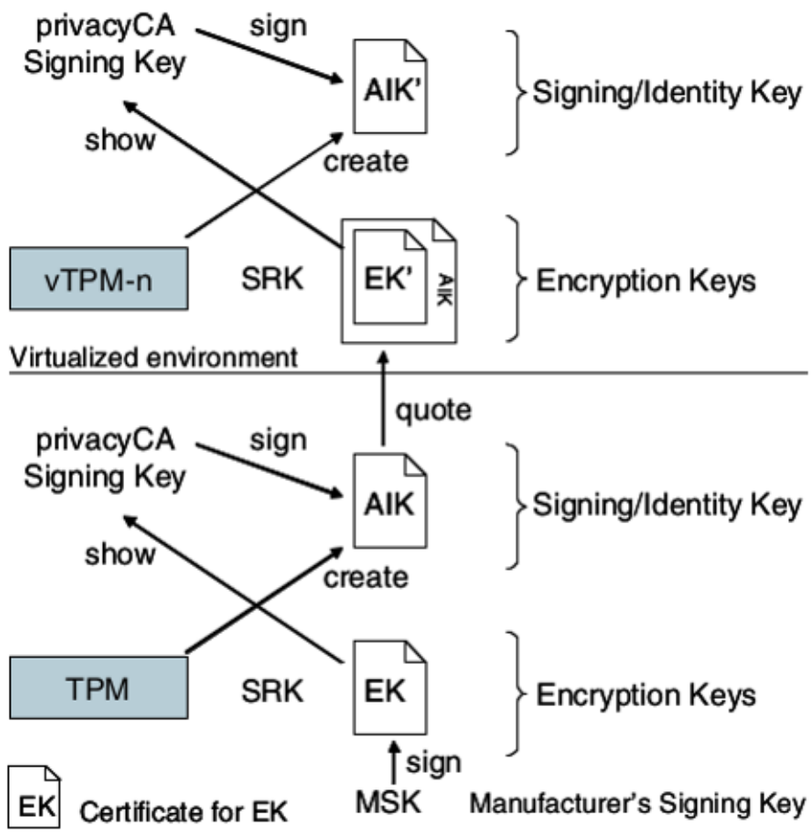


FIGURE 3.3: Certifying vTPM EK using TPM AIK

A vtpmmgr process acts as a server receiving requests from the vTPMs. This manager is responsible to directly talk to the hardware TPM chip. This interaction is shown in Figure 3.2. Trust establishment in a vTPM is done by certifying the EK of a vTPM with an AIK of the TPM as described in the Figure 3.3.

# Chapter 4

# Architecture of the Framework

This chapter discusses the proposed architecture involving a Cloud Service Provider (CSP), Cloud User and a trusted Third Party Authority (TPA) as shown in Figure 4.1. We start by discussing the mechanism through which the root of trust of any running application can be extended to the hardware. We then go on to propose various Service Level Agreements and ways in which they can be measured.

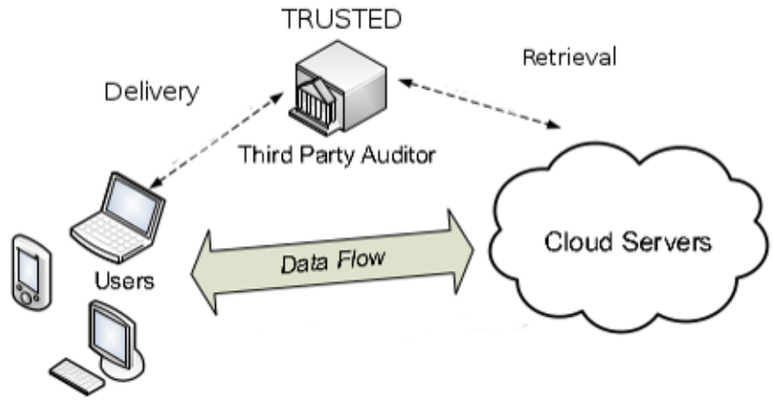


FIGURE 4.1: Framework Architecture consisting of a Third Party

## 4.1 Process Overview

We design our solution as a collection of Daemons that work on the Cloud Server, securely collecting the data needed to answer a consumer's SLAs. This process periodically collects logs, signatures, and their corresponding timestamps which is then transmitted to the Third Party Auditor for secure storage and processing. The TPA stack consists of signature verification scheme, decrypting data and further processing into a format that can be shown to the User.

## 4.2 Trusting AIK

The cloud user specifies the data to be attested but the actual attestation is done by the TPA, thus reducing the load on the user side. On the server side, we extract the EK certificate which is used to generate a new AIK. First, we generate a proof file containing the public part of AIK, EK cert and other intermediate certificates. This is the file which is published and sent to TPA that will claim that the platform is trusted. TPA will generate a nonce which will be encrypted and whose successful decryption at server side will prove the validity. We validate the AIK proof file testing the cert chain for proper root of trust.

In case of successful completion of the test, we generate a challenge file which is decrypted on the server side and sent back to the challenger to compare it with the secret. The verifier then knows that the platform is in a trusted state.

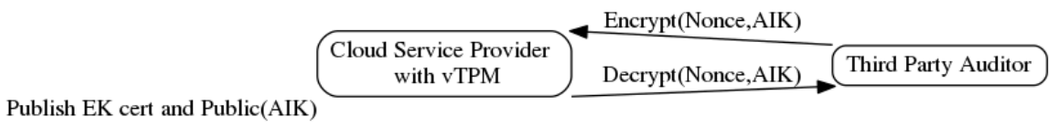


FIGURE 4.2: Trusting AIK

## 4.3 Attestation

To attest an application, TPA requests for a set of PCRs and sends a nonce. The CSP will request the TPM to sign the PCRs and nonce with AIK. The CSP sends this data to the TPA which then knows that the PCRs are valid (as they are signed by a key lying in the chain of trust).

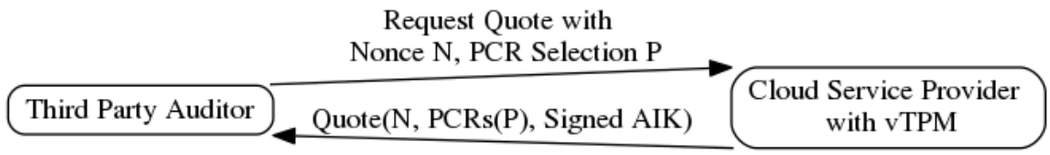


FIGURE 4.3: Attestation

The TPA will check for AIK signatures and compare the PCRs to the known state.

## 4.4 Service Level Agreements

There can be various SLAs that act as questions that a Cloud User can ask the CSP at any point in time. These SLAs act as a pre-defined agreement between the two parties whose answers will reflect whether or not the CSP fulfills its promises to the User.

Here is a list of some sample SLAs along with the method by which they can be measured:

**Availability SLA**

| **SLA** | **Method** |
|---|---|
| Availability of a service | Monitored by periodically pinging the service |

**Security SLAs**

| **SLA** | **Method** |
|---|---|
| Logged in users | Reading /var/log/auth.log |
| User login time | Monitored periodically |
| Processes spawned during the session | Periodically check for processes associated with Users. |
| Privileges changed | PID associated with processes changing privileges in auth.log |
| Files accessed | Using auditd daemon, reading /var/log/audit/audit.log |

**Integrity SLAs**

| | |
|---|---|
| New programs installed | Changes made to $PATH during a session |
| FDs associated with the processes | Monitored periodically from /proc/<pid>/fd |
| Files modified/corrupted | Checksum |
| Backup frequency | User Decided |

# Chapter 5

# Implementation and Testing

Implementation and testing of the framework was done on a system containing a TPM chip running Xen 4.7. We enabled Xen Security Modules to get vTPM support for the DomU instance. The detailed setup procedure is described in Appendix 1. Linux Kernel 3.13.0 was used as the Dom0 and 3.9.1 as DomU.

## 5.1 Attestation

We used our Laptop as a verifier (TPA) for the attestation purposes. The attestation programs were written in C Language which were further automated using Bash Scripts. Three different attestation scripts were written namely, Publish, Challenge and Respond as shown in Figure 5.1. Each of their working is as follows:

1. **Publish** script runs on the CSP containing the TPM chip. It will publish a proof file of AIK for the verifier to read.

2. **Challenge** script is run by the verifier over the published proof and a secret. This will encrypt the secret using the AIK proof file which is sent back to the CSP.

3. **Response** script is then run on the CSP side which reads the encrypted challenge and tries to decrypt it using the private AIK. This decrypted challenge is sent back to the verifier to check for the validity.

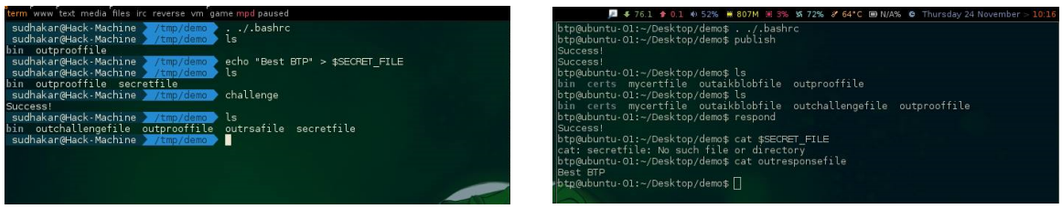


FIGURE 5.1: **Trusting AIK**: Left side is the verifier and right one is the CSP

Once the AIK is trusted, actual attestation is done as follows:

1. Verifier supplies a hash value along with the PCRs it wants to read.
2. Use TPM's **Quote** functionality to sign the extended PCRs using the AIK and sent to the verifier.
3. Since the AIK is controlled by the TPM, it will only sign the correct PCRs.

An example of one such attestation for PCR 10 is shown in Figure 5.2.

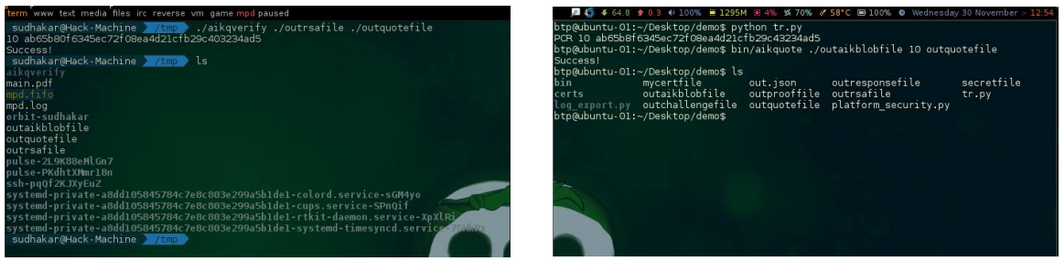


FIGURE 5.2: **Example of Attestation**: Left side is the verifier and right one is the CSP

## 5.2 SLA Daemons

Processes are running in two different contexts. One on the CSP and one on the TPA.
On the CSP stack:

1. Setup rules for *auditd* to look for events. For every *event* encountered it generates *records* for the event.
2. A separate process checks the log location for generated logs in realtime, processes them for particular rules and extracts meaning ful information in *JSON* format to be exported to the software stack on TPA with proper encrypted channels.

On the TPA stack:

1. A controller listens in the channel for the logs.
2. Over time it decrypts and processes the logs for *events* and *records*. Rules are implemented in the process to verify all SLA.
3. On user intervention it can generate a short as well as detailed report of events and SLAs.

# Chapter 6

# Conclusion and Future Scope

Due to various reasons which may be advantageous to the Cloud Service Provider, the CSP may deviate from its normal behavior and in the process, may violate certain promises it must fulfill. Thus, concerning about security is an important factor in Cloud Computing. We described and analyzed the design of protocols for the integrity measurement in the Cloud Platform by constructing a set of SLAs along with the ways in which they can be answered. To reduce the resource utilization on the user machine, we introduce a trusted Third Party Auditor in order to answer the SLAs. Our framework runs on both the CSP side as Daemons to collect and format the logs and on the TPA side to periodically receive the logs from the CSP and parse them into a user-understandable format.

We extend the root of trust of the running applications to the hardware and thus to a Certificate Authority by the means of Trusted Platform Modules (and vTPM for Virtual Machines). A verifier can then measure the integrity of a running application by retrieving a number of PCRs from the CSP and comparing them to a known state. The PCR values are trusted as they are signed by the AIK which is signed by a Certificate Authority.

However for a complete end-product, many of these protocols must be implemented on the hypervisor level. vTPM migration in Xen on remote hosts will allow much more independence to the CSP to use this framework.

# Bibliography


[1] The Xen Project: *https://www.xenproject.org/*.

[2] The Linux Kernel Arhcives: *https://www.kernel.org/*.

[3] Trusted Computing Group: *https://www.trustedcomputinggroup.org/*.

[4] Attestation and Authentication Protocols Using the TPM: *https://www.cylab.cmu.edu/tiw/slides/segall-attestation.pdf*.

[5] Danev, Boris and Masti, Ramya Jayaram and Karame, Ghassan O. and Capkun, Srdjan. Enabling Secure VM-vTPM Migration in Private Clouds. In *Proceedings of the 27th Annual Computer Security Applications Conference* (ACSAC '11). ACM, New York, NY, USA, 187-196.

[6] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. 2006. vTPM: virtualizing the trusted platform module. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15* (USENIX-SS'06), Vol. 15. USENIX Association, Berkeley, CA, USA.

[7] Peter M. Mell and Timothy Grance. 2011. SP 800-145. *the NIST Definition of Cloud Computing*. Technical Report. NIST, Gaithersburg, MD, United States.

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

[9] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. 2007. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM conference on Computer and communications security* (CCS '07). ACM, New York, NY, USA, 598-609.

[10] Qian Wang, Cong Wang, Jin Li, Kui Ren, and Wenjing Lou. 2009. Enabling public verifiability and data dynamics for storage security in cloud computing. In *Proceedings of the 14th European conference on Research in computer security* (ESORICS'09), Michael Backes and Peng Ning (Eds.). Springer-Verlag, Berlin, Heidelberg, 355-370.

[11] Shucheng Yu, Cong Wang, Kui Ren, and Wenjing Lou. 2010. Achieving secure, scalable, and fine-grained data access control in cloud computing. In *Proceedings of the 29th conference on Information communications* (INFOCOM'10). IEEE Press, Piscataway, NJ, USA, 534-542.

[12] Configuring Virtual TPM (vTPM) for Xen 4.3 Guest Virtual Machines: *https://mhsamsal.wordpress.com/2013/12/05/configuring-virtual-tpm-vtpm-for-xen-4-3-guest-virtual-machines/*.

# Appendices

# Appendix A

# Installing and Configuring vTPM with Xen Hypervisor

With these basic steps we can have the vTPM installed for our guest (DomU)

1. Enable TPM in BIOS
2. Install a host operating system -Dom0
3. Install Xen hypervisor
4. Build Dom0 kernel
5. Build kernel and filesystem for DomU
6. Configure vTPM manager and vTPM
7. Boot DomU

## A.1 Enable TPM in BIOS

Clear all the keys and enable TPM from the BIOS.

## A.2 Install a host operating system -Dom0

Download and install latest stable release of Ubuntu from *https://www.ubuntu.com/download*.
We used Ubuntu 14.04

## A.3 Install Xen hypervisor

To enable vTPM default installation from apt won't work. Build it from the source instead.
Install build dependencies from apt-get.

```
# apt-get install bridge-utils build-essential libncurses-
  dev python-dev uuid uuid-dev libglib2.0-dev libyajl-dev
  bcc gcc-multilib iasl libpci-dev mercurial flex bison
  libaio-dev build-essential gettext libpixman-1-dev bin86
  gawk bridge-utils iproute libcurl3 libcurl4-openssl-dev
  bzip2 module-init-tools transfig tgif texinfo texlive-
  latex-base texlive-latex-recommended texlive-fonts-extra
  texlive-fonts-recommended pciutils-dev mercurial make gcc
   libc6-dev zlib1g-dev python python-dev python-twisted
  libncurses5-dev patch libvncserver-dev libsdl-dev libbz2-
  dev e2fslibs-dev git-core uuid-dev ocaml ocaml-findlib
  libx11-dev bison flex xz-utils libyajl-dev
```

Download xen latest source code release. We used Xen 4.7.

```
$ wget http://bits.xensource.com/oss-xen/release/4.7.0/xen-4.7.0.tar.gz
```

Extract and add the following line to *Config.mk*

XSM_ENABLE ?=y

In the xen source directory now configure

```
# ./configure --enable-vtpm-stubdom --enable-vtpmmgr-stubdom
```

If any package is missing then install them from apt-get and continue.

```
# make all
# make install
```

On successful build and installation you can find xen images in */boot*. Also check for *vtpmmgr-stubdom.gz* and *vtpm-stubdom.gz* in *lib* of xen.

## A.4 Build Dom0 kernel

After installing Xen change kernel of Dom0 in order to disable direct TPM access from Dom0 and enable Xen in the kernel. For compatibility reasons choose the same kernel as the one already installed.
Download from *https://www.kernel.org/*. We used *Linux 3.13.0*

```
$ wget https://www.kernel.org/pub/linux/kernel/v3.0/linux-3.13.tar.gz
```

Extract and configure

```
$ tar xf linux-3.13.tar.gz
$ cd linux-3.13.tar.gz/
$ sudo make menuconfig
```

Disable *TPM Hardware Support* in *Character Devices* of *Device Drivers*. Add the following lines in .config after successful menuconfig.

```
CONFIG_ACPI_PROCFS=y
CONFIG_XEN=y
CONFIG_XEN_MAX_DOMAIN_MEMORY=32
CONFIG_XEN_SAVE_RESTORE=y
CONFIG_XEN_DOM0=y
CONFIG_XEN_PRIVILEGED_GUEST=y
CONFIG_XEN_PCI=y
CONFIG_PCI_XEN=y
CONFIG_XEN_BLKDEV_FRONTEND=y
CONFIG_XEN_NETDEV_FRONTEND=y
CONFIG_XEN_KBDDEV_FRONTEND=y
CONFIG_HVC_XEN=y
CONFIG_XEN_FBDEV_FRONTEND=y
CONFIG_XEN_BALLOON=y
CONFIG_XEN_SCRUB_PAGES=y
CONFIG_XEN_DEV_EVTCHN=y
CONFIG_XEN_GNTDEV=y
CONFIG_XEN_BACKEND=y
CONFIG_XEN_BLKDEV_BACKEND=y
CONFIG_XEN_NETDEV_BACKEND=y
CONFIG_XENFS=y
CONFIG_XEN_COMPAT_XENFS=y
CONFIG_XEN_XENBUS_FRONTEND=y
CONFIG_XEN_PCIDEV_FRONTEND=y
```

Build and install the images and modules.

```
$ sudo make modules_prepare
$ sudo make
$ sudo make modules_install
$ sudo make install
$ sudo cd /boot
$ sudo mkinitramfs -o initrd.img-3.13.0 3.13.0
$ sudo update-grub
```

This should install all the modules and new kernel. Reboot and choose the newly installed kernel in GRUB. Newer kernels don't disable the *tpm_tis* driver even when configured in .config. To get past this add the following line in */etc/modprobe.d/blacklist.conf*.

```
blacklist tpm_tis
```

This is absolutely necessary that no driver or program should access the hardware TPM from Dom0 while booting guest machines or vTPM manager.

## A.5 Build kernel and filesystem for DomU

Most mainline linux kernels don't support vTPM. Download modified kernel source tree from *https://github.com/virt-cloud/domuKernel* Make *menuconfig* and enable *Integrity Measurement Architecture(IMA)* and TPM in menu.
Also add the following lines in *.config*

```
CONFIG_XEN=y
CONFIG_PARAVIRT_GUEST=y
CONFIG_PARAVIRT=y
CONFIG_XEN_PVHVM=y
CONFIG_XEN_MAX_DOMAIN_MEMORY=128
CONFIG_XEN_SAVE_RESTORE=y
CONFIG_PCI_XEN=y
CONFIG_XEN_PCIDEV_FRONTEND=y
CONFIG_XEN_BLKDEV_FRONTEND=y
CONFIG_XEN_NETDEV_FRONTEND=y
CONFIG_INPUT_XEN_KBDDEV_FRONTEND=y
CONFIG_HVC_XEN=y
CONFIG_XEN_FBDEV_FRONTEND=y
CONFIG_XEN_DEV_EVTCHN=y
CONFIG_XEN_XENBUS_FRONTEND=y
CONFIG_TCG_TPM=y
CONFIG_TCG_XEN=y
```

and then build

```
$ sudo make
```

Build a filesystem using *debootstrap*. First build an empty disk and mount it.

```
$ sudo dd if=/dev/zero of=domu.img bs=1024K count=10240
$ sudo /sbin/mkfs.ext4 domu.img
$ sudo mount -o loop domu.img /mnt/
```

Install a base system.

```
$ sudo debootstrap --arch amd64 jessie /mnt/
```

Depending on the filesystem you might have to uninstall *systemd*. Add *hvc0* as a *tty* in the newly built filesystem so that it is spawned at boot time. Also configure

credentials for users, network and filesystem according to your preferences.
Now build the kernel modules in this filesystem and generate initial ram disk.

```
$ cd linux-3.9.1
$ sudo make modules_install INSTALL_MOD_PATH=/mnt
$ sudo cp .config /mnt/boot/config-3.9.1
$ sudo chroot /mnt
$ apt-get install initramfs-tools
$ mkinitramfs -o initrd.img-3.9.1-domU 3.9.1
```

## A.6 Configure vTPM manager and vTPM

vTPM manager is a domain by itself which stores and manages the sessions for vTPM of guest machines. So it requires a hard disk image and configuration. Also it should be the first domain to boot up after Dom0.

```
$ sudo dd if=/dev/zero of=/var/vtpmmgr-stubdom.img bs=16M count=1
```

Verify the presence of */usr/local/lib/xen/boot/vtpmmgr-stubdom.gz* and save the following config file to */var/vtpmmgr.cfg*. If */usr/local/lib/xen/boot/vtpmmgr-stubdom.gz* is located someplace else change the config to reflect that.

```
kernel="/usr/local/lib/xen/boot/vtpmmgr-stubdom.gz"
memory=16
disk=["file:/var/vtpmmgr-stubdom.img,hda,w"]
name="vtpmmgr"
iomem=["fed40,5"]
```

Now we create a vTPM. Like vTPM manager this too requires a hard disk image and config file. This should be booted before its respective DomU.

```
$ sudo dd if=/dev/zero of=/var/vtpm.img bs=8M count=1
```

Verify the presence of */usr/local/lib/xen/boot/vtpm-stubdom.gz* and save the following config file to */var/vtpm-DomU.cfg*. If */usr/local/lib/xen/boot/vtpm-stubdom.gz* is located someplace else change the config to reflect that.

```
kernel="/usr/local/lib/xen/boot/vtpm-stubdom.gz"
memory=8
disk=["file:/var/vtpm-DomU.img,hda,w"]
name="domu-vtpm"
vtpm=["backend=vtpmmgr,uuid=<Use uuidgen to generate a new
   UUID>"]
```

## A.7 Boot DomU

Create a DomU config file in */var/DomU.cfg*

```
kernel = <Location of your vmlinux file from DomU kernel>
ramdisk = <Location of the initrd.img created earlier>
vcpus = '1'
memory = '1024'
root = '/dev/xvda1 ro'
disk=['tap:aio:<<Location of the domu.img created earlier>>,
   xvda1,w']
name = <Name>
vif = [ '','bridge=xenbr0']
dhcp = "dhcp"
on_poweroff = 'destroy'
on_reboot = 'restart'
on_crash = 'restart'
extra = 'console=hvc0 xencons=tty'
vtpm=["backend=domu-vtpm"]
```

With all the configuration done, start the vtpmmgr

```
$ sudo xl create -c /var/vtpmmgr.cfg
```

Start the vTPM

```
$ sudo xl create -c /var/vtpm-DomU.cfg
```

Start DomU

```
$ sudo xl create -c /var/DomU.cfg
```

This should boot up DomU. To check successful vTPM deployment in DomU do

```
$ sudo apt-get install tpm-tools trousers
$ sudo tcsd
$ sudo tpm_version
```

would result apt output about the vTPM.

# Appendix B

# Code Constructs and Development

To set up the development environment for TPM and its libraries.

```
$ sudo apt-get install tpm-tools trousers libtspi-dev gcc
```

Start up *tcsd*

```
$ sudo modprobe tpm_tis
$ sudo tcsd start
```

To verify proper setup run *tpm_getpubek* from *tpm-tools*. After this take ownership of the TPM. For development and ease using *Well Known Secret* is recommended. It is just 20 bytes of zeroes.

```
$ tpm_takeownership -z
```

Include libraries

```
#include <tss/tss_error.h>
#include <tss/platform.h>
#include <tss/tss_defines.h>
#include <tss/tss_typedef.h>
#include <tss/tss_structs.h>
#include <tss/tspi.h>
#include <trousers/trousers.h>
```

GCC flags for tss is *-ltspi* . This flag should be added for proper linking.

## B.1 Connecting to the TPM

This has to be done in every program before doing anything

```
TSS_HCONTEXT hContext=0;
TSS_HTPM hTPM = 0;
TSS_RESULT result;
TSS_HKEY hSRK = 0;
TSS_HPOLICY hSRKPolicy=0;
TSS_UUID SRK_UUID = TSS_UUID_SRK;
BYTE wks[20] = TSS_WELL_KNOWN_SECRET; // Place to put the
   well known secret
// Pick the TPM you are talking to in this case the system
   TPM (which you connect to with NULL)
result =Tspi_Context_Create(&hContext);
result=Tspi_Context_Connect(hContext, NULL);
// Get the TPM handle
result=Tspi_Context_GetTpmObject(hContext, &hTPM);
//Get the SRK handle
result=Tspi_Context_LoadKeyByUUID(hContext,
   TSS_PS_TYPE_SYSTEM, SRK_UUID, &hSRK);
//Get the SRK policy
result=Tspi_GetPolicyObject(hSRK, TSS_POLICY_USAGE, &
   hSRKPolicy);
// Then we set the SRK policy to be the well known secret
result=Tspi_Policy_SetSecret(hSRKPolicy,TSS_SECRET_MODE_SHA1
   ,20, wks);
```

## B.2 Generate and Publish an AIK

To prove that a system has a valid AIK we assume that it also has an EK certificate. This EK certificate must have a certificate chain which ends in a root certification key issued by an controlled by a Globally trusted CA.

```
TSS_HKEY          hPCA;
/* Create dummy PCA key */
result = Tspi_Context_CreateObject(hContext,
   TSS_OBJECT_TYPE_RSAKEY,TSS_KEY_TYPE_LEGACY|
   TSS_KEY_SIZE_2048,&hPCA);
memset (n, 0xff, sizeof(n));
result = Tspi_SetAttribData (hPCA, TSS_TSPATTRIB_RSAKEY_INFO
   ,TSS_TSPATTRIB_KEYINFO_RSA_MODULUS, sizeof(n), n);
TSS_HKEY          hAIK;
/* Create AIK object */
initFlags = TSS_KEY_TYPE_IDENTITY | TSS_KEY_SIZE_2048;
BYTE              *blob;
UINT32            blobLen;
result = Tspi_TPM_CollateIdentityRequest(hTPM, hSRK, hPCA,
   0, "",hAIK, TSS_ALG_AES,&blobLen,&blob);
Tspi_Context_FreeMemory (hContext, blob);

/* Output file with AIK pub key and certs, preceded by 4-
   byte lengths */
result = Tspi_GetAttribData (hAIK, TSS_TSPATTRIB_KEY_BLOB,
   TSS_TSPATTRIB_KEYBLOB_PUBLIC_KEY, &blobLen, &blob);
```

## B.3 Challenging and response for AIK verification

Once the CSP publishes its AIK claim, the TPA can then verify the certificate chain by itself and encrypt some secret data with the AIK published. The system that created the AIK takes the encrypted challenge, load the AIK using *Tspi_Context_LoadKeyByBlob* and the decrypt using *Tspi_TPM_ActivateIdentity*
A properly decrypted response can be verified by the TPA for authenticity of the AIK and its platform.

## B.4 Quoting and response for PCR verification

If the system has booted to a known state, the state of PCRs is same. This can be verified by the TPA using quote from the CSP. The TPA can specify the list of PCRs it wants to verify and a nonce for freshness. Quote operation gives out an response of the PCRs signed with the previously published AIK.

```
result = Tspi_Context_LoadKeyByBlob (hContext, hSRK, bufLen,
    buf, &hAIK); CKERR;
free (buf);

if (pass) {
result = Tspi_Context_CreateObject(hContext,
   TSS_OBJECT_TYPE_POLICY,
TSS_POLICY_USAGE, &hAIKPolicy); CKERR;
result = Tspi_Policy_AssignToObject(hAIKPolicy, hAIK);
result = Tspi_Policy_SetSecret (hAIKPolicy,
   TSS_SECRET_MODE_PLAIN,
strlen(pass)+1, pass); CKERR;
}

/* Create PCR list to be quoted */
tpmProp = TSS_TPMCAP_PROP_PCR;
result = Tspi_TPM_GetCapability(hTPM, TSS_TPMCAP_PROPERTY,
sizeof(tpmProp), (BYTE *)&tpmProp, &tmpbufLen, &tmpbuf);
   CKERR;
npcrMax = *(UINT32 *)tmpbuf;
Tspi_Context_FreeMemory(hContext, tmpbuf);
npcrBytes = (npcrMax + 7) / 8;
result = Tspi_Context_CreateObject(hContext,
   TSS_OBJECT_TYPE_PCRS,
TSS_PCRS_STRUCT_INFO, &hPCRs); CKERR;

/* Also PCR buffer */
buf = malloc (2 + npcrBytes + 4 + 20 * npcrMax);
*(UINT16 *)buf = htons(npcrBytes);
for (i=0; i<npcrBytes; i++)
buf[2+i] = 0;

for (i=2; i<ac-1; i++) {
char *endptr;
long pcr = strtol (av[i], &endptr, 10);
if (pcr < 0 || pcr > npcrMax || *av[i] == 0 || *endptr != 0)
    {
fprintf (stderr, "Illegal PCR value %s\n", av[i]);
exit (1);
}
result = Tspi_PcrComposite_SelectPcrIndex(hPCRs, pcr); CKERR
   ;
```

```
++npcrs;
buf[2+(pcr/8)] |= 1 << (pcr%8);
}

/* Create TSS_VALIDATION struct for Quote */
valid.ulExternalDataLength = sizeof(chalmd);
valid.rgbExternalData = chalmd;

/* Perform Quote */
result = Tspi_TPM_Quote(hTPM, hAIK, hPCRs, &valid); CKERR;
quoteInfo = (TPM_QUOTE_INFO *)valid.rgbData;
```

Once the quote file generated, the TPA can then verifiy the correct signature and values of the PCRs by the public part of AIK published earlier.

# Appendix C

# Logging and processing

Logging and auditing is based on *auditd*. A daemon runs on the CSP to process the logs generated for events. A sample code to export the logs

```
import auparse
import audit
import json
def walk_log(au):
    event_cnt = 1
    au.reset()
    while True:
        if not au.first_record():
            print "Error_getting_first_record"
            sys.exit(1)
        out_dict = {}
        out_dict['record_count'] = au.get_num_records()
        out_dict['records'] = []
        record_cnt = 1
        while True:
            record = {}
            record['type'] = '%d(%s)' % (au.get_type(),
               audit.audit_msg_type_to_name(au.get_type()))
            record['field_count'] = au.get_num_fields()
            record['line'] = au.get_line_number()
            record['file'] = au.get_filename()
            event = au.get_timestamp()
            if event is None:
                print "Error_getting_timestamp_aborting"
                sys.exit(1)
            record['time'] = "%d.%d:%d" % (event.sec, event.
               milli, event.serial)
            record['host'] = none_to_null(event.host)
            au.first_field()
            record['event'] = {}
            while True:
                record['event'][au.get_field_name()] = au.
                   get_field_str(), au.interpret_field()
                if not au.next_field(): break
            out_dict['records'].append(record)
            record_cnt += 1
            if not au.next_record(): break
        print json.dumps(out_dict)
        event_cnt += 1
        if not au.parse_next_event(): break
au = auparse.AuParser(auparse.AUSOURCE_FILE,'/var/log/audit/
   audit.log')
```

This will generate a *JSON* dump of logs.

```
{"records":
[{"field_count": 12, "host": "(null)", "file": "/var
   /log/audit/audit.log", "time":
   "1479295201.43:84", "line": 17, "type": "1101(
   USER_ACCT)", "event": {"auid": ["4294967295", "
   unset"], "exe": ["\"/usr/sbin/cron\"", "/usr/sbin
   /cron"], "ses": ["4294967295", "unset"], "uid":
   ["0", "root"], "res": ["success", "success"], "
   hostname": ["?", "?"], "pid": ["7819", "7819"], "
   terminal": ["cron", "cron"], "addr": ["?", "?"],
   "acct": ["\"btp\"", "btp"], "type": ["USER_ACCT",
    "USER_ACCT"], "op": ["PAM:accounting", "PAM:
   accounting"]}}],
"record_count":
1}
```

Depending on the event type and rules set these *JSON* objects can be parsed using even the simple libraries.