D2.1.5
Final Reports on Requirements, Architecture, and Components for Single Trusted Clouds

Project number:	257243
Project acronym:	TClouds
Project title:	Trustworthy Clouds - Privacy and Resilience for Internet-scale Critical Infrastructure
Start date of the project:	1st October, 2010
Duration:	36 months
Programme:	FP7 IP

Deliverable type:	Report
Deliverable reference number:	ICT-257243 / D2.1.5 / 1.0
Activity and Work package contributing to deliverable:	Activity 2 / WP 2.1
Due date:	September 2013 – M36
Actual submission date:	30th September, 2013

Responsible organisation:	SRX
Editor:	Norbert Schirmer
Dissemination level:	Public
Revision:	1.0

Abstract:	cf. Executive Summary
Keywords:	Trusted computing, remote attestation, secure logging, confidentiality for commodity cloud storage, efficient resilience, tailored components

Editor

Norbert Schirmer (SRX)

Contributors

Johannes Behl, Stefan Brenner, Klaus Stengel (TUBS)

Nicola Barresi, Gianluca Ramunno, Roberto Sassu, Paolo Smiraglia (POL)

Alexander Büger, Norbert Schirmer (SRX)

Tobias Distler, Andreas Ruprecht (FAU)

Sören Bleikertz, Zoltan Nagy (IBM)

Imad M. Abbadi, Anbang Ruad (OXFD)

Sven Bugiel, Hugo Hideler, Stefan Nürnberger (TUDA)

Disclaimer

This work was partially supported by the European Commission through the FP7-ICT program under project TClouds, number 257243.

Executive Summary

This deliverable summarizes the technical work of WP 2.1, where the technical artefacts are integrated into two infrastructures, the Trusted Infrastructure Cloud and Trustworthy OpenStack. The Trusted Infrastructure Cloud is constructed from ground up with security and trustworthiness in mind, employing trusted computing technologies as a hardware anchor. With trusted boot and remote attestation we ensure that only untampered servers with our security kernel are started and that the sole way of administration is via the trusted channel from the management component TOM. Hence no administrator with elevated privileges is necessary and hence this functionality is completely disabled, abandoning the possibility for an administrator to corrupt the system. On the contrary, Trustworthy OpenStack is based on OpenStack which has a strong bias towards a scalable and decentralized architecture. We extend or embed new components into the OpenStack framework to improve its security, these are Access Control as a Service, Ontology-based Reasoner-Enforce, Remote Attestation Service, Cryptography as a Service, Log Service, Ressource-efficient BFT, and Simple Key / Value Storage. With these two infrastructures we can cover the needs of wide range of application scenarios. Trusted Infrastructure Cloud is especially attractive for private or community clouds with high security demands, while Trustworthy OpenStack is attractive for large-scale public clouds.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 TClouds — Trustworthy Clouds

TClouds aims to develop *trustworthy* Internet-scale cloud services, providing computing, network, and storage resources over the Internet. Existing cloud computing services today are generally not trusted for running *critical infrastructures*, which may range from business-critical tasks of large companies to mission-critical tasks for the society as a whole. The latter includes water, electricity, fuel, and food supply chains. TClouds focuses on power grids and electricity management and on patient-centric health-care systems as its main applications.

The TClouds project identifies and addresses legal implications and business opportunities of using infrastructure clouds, assesses security, privacy, and resilience aspects of cloud computing and contributes to building a regulatory framework enabling resilient and privacy-enhanced cloud infrastructure.

The main body of work in TClouds defines an architecture and prototype systems for securing infrastructure clouds, by providing security enhancements that can be deployed on top of commodity infrastructure clouds (as a cloud-of-clouds) and by assessing the resilience, privacy, and security extensions of existing clouds.

Furthermore, TClouds provides resilient middleware for adaptive security using a cloud-of-clouds, which is not dependent on any single cloud provider. This feature of the TClouds platform will provide tolerance and adaptability to mitigate security incidents and unstable operating conditions for a range of applications running on a clouds-of-clouds.

1.2 Activity 2 — Trustworthy Internet-scale Computing Platform

Activity 2 carries out research and builds the actual TClouds platform, which delivers trustworthy resilient cloud computing services. The TClouds platform contains trustworthy cloud components that operate inside the infrastructure of a cloud provider; this goal is specifically addressed by WP2.1. The purpose of the components developed for the infrastructure is to achieve higher security and better resilience than current cloud computing services may provide.

The TClouds platform also links cloud services from multiple providers together, specifically in WP2.2, in order to realize a comprehensive service that is more resilient and gains higher security than what can ever be achieved by consuming the service of an individual cloud provider alone. The approach involves simultaneous access to resources of multiple commodity clouds, introduction of resilient cloud service mediators that act as added-value cloud providers, and client-side strategies to construct a resilient service from such a cloud-of-clouds.

WP2.3 introduces the definition of languages and models for the formalization of user- and application-level security requirements, involves the development of management operations for

security-critical components, such as “trust anchors” based on trusted computing technology (e.g., TPM hardware), and it exploits automated analysis of deployed cloud infrastructures with respect to high-level security requirements.

Furthermore, Activity 2 will provide an integrated prototype implementation of the trustworthy cloud architecture that forms the basis for the application scenarios of Activity 3. Formulation and development of this integrated platform is the subject of WP2.4.

These generic objectives of A2 can be broken down to technical requirements and designs for trustworthy cloud-computing components (e.g., virtual machines, storage components, network services) and to novel security and resilience mechanisms and protocols, which realize trustworthy and privacy-aware cloud-of-clouds services. They are described in the deliverables of WP2.1–WP2.3, and WP2.4 describes the implementation of an integrated platform.

1.3 Workpackage 2.1 — Trustworthy Cloud Infrastructure

The overall objective of WP2.1 is to improve the security, resilience and trustworthiness of components and the overall architechture of an infrastructure cloud. The workpackage is split into four tasks.

- Task 2.1.1 (M01-M20) Technical Requirements and Architecture for Privacy-enhanced Resilient Clouds
- Task 2.1.2 (M07-M36) Adaptive Security by Cloud Management and Control
- Task 2.1.3 (M01-M36) Security-enhanced Cloud Components
- Task 2.1.5 (M18-M36) Proof of Concept Infrastructure

Task 2.1.1 and Task 2.1.5 follow each other with a slight overlapping. In Task 2.1.1 the requirements analysis took place mainly in the first year and we also identified the gaps and weaknesses of existing cloud solutions. From there we researched into components and architectures to improve security, resilience and trustworthiness of an infrastructure cloud. In Task 2.1.5 we continue to implement the designs into a prototype system. Tasks 2.1.2 and 2.1.3 identify sub-topics that are continuously worked on during building prototypes and doing research.

Figure 1.1 illustrates WP2.1 and its relations to other workpackages according to the DoW/Annex I.

Requirements were collected from WP1 which guided our requirements and gap analysis. Also requirements from the application scenarios in WP3.1 and WP3.2 were considered. Task 2.1.2 which is concerned about management aspects of the cloud infrastructure is strongly related to WP2.3 the overall management workpackage. The prototypes developed in Task 2.1.5 are input for the overall platform and prototype work of WP2.4 where the necessary interfaces and integration requirements are feed back to Task 2.1.5. The resulting platform and prototypes are employed by WP3.1 and WP3.2 for the application scenarios and are validated and evaluated in WP3.3.

In the first year we did an intensive gap and requirement analysis to identify the major shortcomings of commodity cloud offerings. In the second year we researched into components and mechanisms to overcome this shortcomings. Finally, in the third year we integrated the technical insights into our proof-of-concept prototypes and combined them into our TClouds platform. Besides significant research papers and results the work integrated into two infrastructures, the Trusted Infrastructure Cloud and the Trustworthy OpenStack. These infrastructures where evaluated in WP3.3 and are used as the core for the application scenarios.

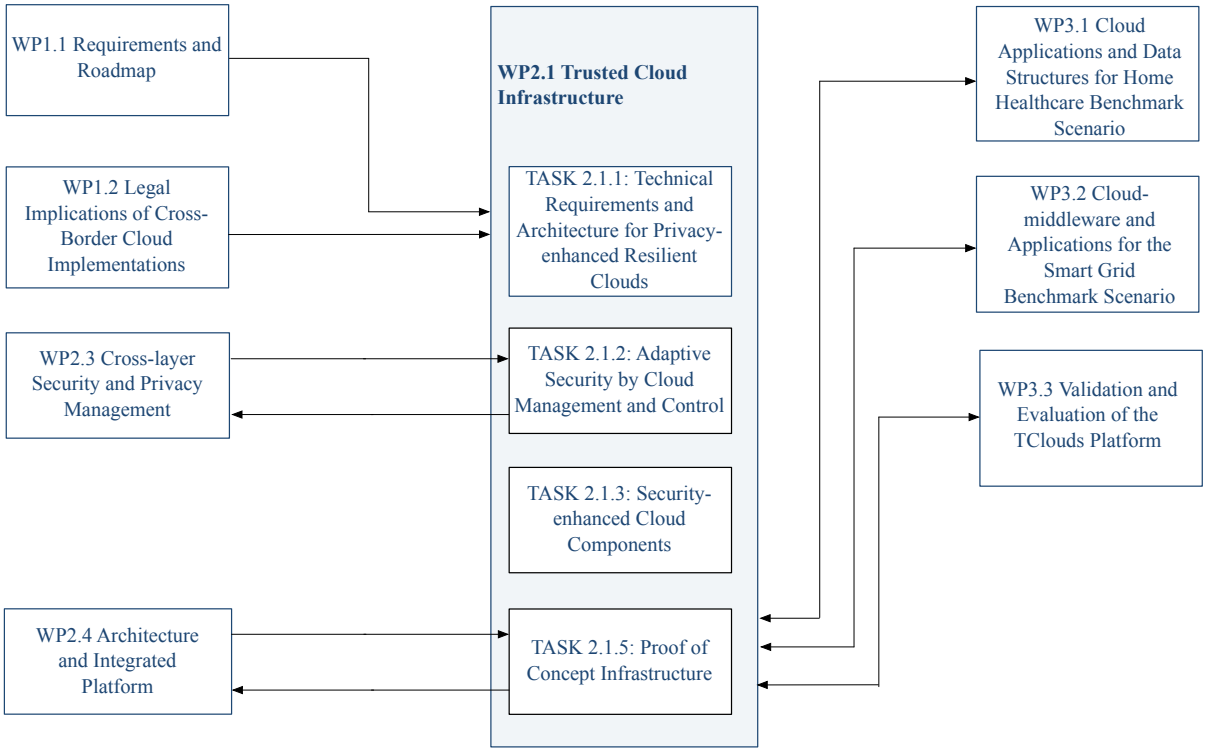


Figure 1.1: Graphical structure of WP2.1 and relations to other workpackages.

1.4 Deliverable 2.1.5 — Final Reports on Requirements, Architecture, and Components for Single Trusted CloudsPreliminary Description of Mechanisms and Components for Single Trusted Clouds

Overview. This deliverable summarizes the technical work of WP 2.1, which are integrated into two infrastructures, the Trusted Infrastructure Cloud and the Trustworthy OpenStack. The TrustedInfrastructure Cloud is constructed from ground up with security and trustworthiness in mind, employing trusted computing technologies as a hardware anchor. With trusted boot and remote attestation we ensure that only untampered servers with our security kernel are started and that the sole way of administration is via the trusted channel from the management component TOM. Hence no administrator with elevated privileges is necessary and hence this functionality is completely disabled, abandoning the possibility for an administrator to corrupt the system. On the contrary, Trustworthy OpenStack is based on OpenStack which has a strong bias towards a scalable and decentralized architecture. We extend or embed new components into the OpenStack framework to improve its security, these are Access Control as a Service, Ontology-based Reasoner-Enforce, Remote Attestation Service, Cryptography as a Service, Log Service, Ressource-efficient BFT, and Simple Key / Value Storage. With these two infrastructures we can cover the needs of wide range of application scenarios. Trusted Infrastructure Cloud is especially attractive for private or community clouds with high security demands, while Trustworthy OpenStack is attractive for large-scale public clouds.

Deviation from Workplan. This deliverable aligns with the DoW/Annex I, Version 4.

Target Audience. This deliverable aims at researchers and developers of secure cloud-computing platforms. The deliverable assumes graduate-level background knowledge in computer science technology, specifically, in virtual-machine technology, operating system concepts, security policy and models, basic cryptographic concepts and formal languages.

Relation to Other Deliverables. The workpackages and especially the year 3 deliverables of Activity 2 are closely related with each other, reflecting the integration efforts of Activity 2. Roughly speaking WP 2.1 provides resilience, privacy and security to individual infrastructure clouds (IaaS). WP 2.2 provides resilient middleware offering both infrastructure (IaaS) as well as platform (PaaS) services, following the cloud-of-cloud paradigm. WP 2.3 deals with the security management and finally in WP 2.4 all is integrated to the final TClouds platform. So to get the complete picture, the reader has to consider all deliverables of the different workpackages.

To help the reader to gain a clean view of the Activity 2 outcomes for the third and final year of the project, we provide here an overall view of the delivered TClouds platform and how to map it to the actual Activity 2 deliverables (or their parts or chapters) released during the third year. This view can be understood as the logical outline of all deliverables which is broken down to the content presented in the different deliverables. The following logical view starts from high level "big picture" (i.e. the latest concept of the TClouds platform), and moves down towards an in-depth and more concrete level, covering research and technical details of the integration of subsystems, updated research and technical details of single subsystems, and the actually delivered software with instructions for installation and configuration.

(Part 1) TClouds platform v2.x: definition of platform, comparison with Amazon AWS ecosystem, big picture of TClouds ecosystem and summary presentation tailored platform instantiations into two Activity 3 benchmark scenarios (for further details on the benchmark scenarios, see respectively D3.1.5 [D+13] and D3.2.4 [VS13]-D3.2.5 [Per13])

- D2.4.3 [R+13], Chapter 2

(Part 2) Integrated prototypes: research/technical details of the integration of subsystems to form IaaS alternatives – Trustworthy OpenStack and TrustedInfractructure Cloud – and PaaS modules/services – C2FS and Relational DB

- D2.1.5 (this document), Part I
- D2.2.4 [B+13a], Chapters 6 and 7
- D2.3.4 [B+13b], Chapters 3 and 5

(Part 3) Subsystems: research/technical details of single subsystems – only updates from previous deliverables

- D2.1.5 (this document), Part II
- D2.2.4 [B+13a], Chapters 2, 3, 4 and 5
- D2.3.4 [B+13b], Chapters 2, 4 and 6

(Part 4) Testing of prototypes and subsystems: test plans and results

- D2.4.3 [B+13b], Part II

(Part 5) Software details: instructions for installation, configuration and usage of prototypes (integrated subsystems)

- D2.1.4-D2.3.3 [BS+13]
- D2.4.3 [R+13], Appendix A

(Part 6) Software delivery: source code and/or binary code of prototypes and subsystems

- TClouds platform v2.0 (only subsystems for single cloud): D2.1.4-D2.3.3 [BS+13], companion tarball(s)
- TClouds platform v2.1 (complete): D2.4.3 [R+13], companion tarball(s)

Structure. Following this logical structure of the Activity 2 deliverables explained above this deliverable has two parts. In Part I we give a high-level view of the two infrastructures, Trustworthy OpenStack in chapter 2 and Trusted Infrastructure Cloud chapter 3. Part II describes on a more detailed technical level refinements or further developments and evaluations on individual sub-systems: In chapter 4 for the Remote Attestation Service, in chapter 5 for the Log Service, in chapter 6 for CheapBFT, in chapter 7 for the Key / Value Storage (memcached) and finally in chapter 8 for the confidentiality proxy for commodity cloud storage.

Part I

TClouds Prototypes for Single Trusted Cloud

Chapter 2

Trustworthy OpenStack

Chapter Authors:
Gianluca Ramunno, Roberto Sassu, Paolo Smiraglia (POL)
Sören Bleikertz, Zoltan Nagy (IBM)
Imad M. Abbadi, Anbang Ruad (OXFD)
Norbert Schirmer (SRX)
Johannes Behl, Klaus Stengel (TUBS)
Sven Bugiel, Hugo Hideler, Stefan Nürnberger (TUDA)

This chapter summarizes the high level objectives and the architecture of the Trustworthy OpenStack (cf. D2.4.2 [S+12a] Section 3.1). It is a consolidated version of the documentation presented in other, mostly previous deliverables, revised with the necessary updates reflecting the work of the last period of the TClouds project.

2.1 Motivation

One of the approaches of Workpackage 2.1 was to focus on existing software for cloud infrastructure, originally designed for high scalability, and to improve its overall security. Through proper security extensions (Cloud Nodes Verification/Remote Attestation, Advanced VM Scheduling, VM Images Transparent Decryption, Secure Logging, Tenant Isolation, VM Security Assessment and Log Resiliency) the security of chosen software, OpenStack, has been enhanced in various dimensions:

Trust / Integrity:

- The Cloud Nodes Verification/Remote Attestation extension enables users to trust that their virtual machines are actually deployed on computing nodes that satisfy their integrity requirements. Based on Trusted Computing technologies, e.g., a Trusted Platform Module, this extension verifies the configuration of the computing nodes and provides them for example to the cloud scheduler.
- The Advanced VM Scheduling extension matches user requirements (e.g., location restrictions or white lists of measurements for deploying a VM) with physical properties of computing nodes. The Cloud Nodes Verification/Remote Attestation extension on the computing nodes is employed by the scheduler to query the computing nodes in a trustworthy way.

Confidentiality: the VM Images Transparent Decryption extension provides disk encryption for volumes attached to virtual machines, as well as encryption of the VM images themselves.

These subsystems offer an API to cloud users to securely provide the encryption keys without giving the cloud provider access to them. This is an important improvement over current encryption schemes where the keys are under control of the cloud provider.

Audit: The Secure Logging extension allows to store logs of computing nodes selected by the Cloud Scheduler for VM deployment. It ensures confidentiality and integrity of the logs.

Isolation: the Tenant Isolation and VM Security Assessment extensions provide the isolation of the networks of each tenant through the proper network configuration (creation of Trusted Virtual Domains) and the validation of the resulting settings.

Resilience: The Log Resiliency extension provides fault tolerance to the Secure Logging. With special FPGA hardware, this solution can tolerate byzantine (i.e. arbitrary) failure modes of a certain amount of computing nodes. While traditional solutions require $3f+1$ replicas to tolerate f faults (i.e. 4 machines for 1 fault), Log Resiliency achieves the same with just $f+1$ active replicas backed up by f passive ones.

The software resulting from adding the Security Extensions to OpenStack is a key result of TClouds: called Trustworthy OpenStack (TOS), is one of the two alternatives offered by the TClouds Platform (see D2.4.3 [R+13], Chapter 2) at IaaS layer. The other one is Trusted infrastructure Cloud that is described in Chapter 3.

2.2 Architecture

Figure 2.1 illustrates the Trustworthy OpenStack architecture showing how the mentioned security extensions are implemented through the integration of a set of TClouds subsystems. Table 2.1 reports the correspondence between Trustworthy OpenStack Security Extensions and TClouds subsystems implementing them.

TOS Security Extension	**TClouds subsystem**
Cloud Nodes Verification/Remote Attestation	Remote Attestation Service
Advanced VM Scheduling	Access Control as a Service (ACaaS)
VM Images Transparent Decryption	Cryptography as a Service (CaaS)
Secure Logging	LogService (*)
Tenant Isolation	Ontology-based Reasoner/Enforcer
VM Security Assessment	Security Assurance of Virtualized Environments (SAVE)
Log Resiliency	Resource-efficient BFT (CheapBFT) (**)

(*) LogService is a subsystem actually belonging to Trustworthy OpenStack (i.e. at the IaaS layer) but it can also act as a service at PaaS layer for generic cloud applications.
(**) CheapBFT is not a subsystem actually belonging to Trustworthy OpenStack as it is a middleware (at PaaS layer) that can be used by generic applications.

Table 2.1: Mapping of Trustworthy OpenStack Security Extensions to TClouds subsystems

Note that the security improvements are conceived by the synergy of the careful selection and integration of TCloud subsystems, e.g.:

- The Remote Attestation Service and ACaaS together ensure that the selection of computing nodes matches the users requirements. Together with the LogService the scheduling decisions are securely stored for audit.

- The combination of CaaS with the ACaaS and Remote Attestation ensures that encrypted images will only be deployed on computing nodes that properly secure the encryption keys.

- The combined use of the Ontology-based Reasoner/Enforcer and Security Assurance of Virtualized Environments guarantee the isolation of the networks of the tenants via creation of Trusted Virtual Domains and validation of the network configuration being set.

- The integration of the Log Service with CheapBFT provides a fault tolerant implementation of the LogService within the cloud infrastructure.

In the following paragraphs a quick description of the subsystems will be given.

Remote Attestation Service is the subsystem responsible to assess the integrity of the nodes in the cloud infrastructure through techniques based on the Trusted Computing technology.

This service gives significant advantages in the cloud environment. First, it allows cloud users to deploy their virtual machines on a physical host that satisfies the desired security requirements – represented by five integrity levels. Requiring a higher level will give more confidence and trust into the used physical hosts. The highest integrity level means that a host is running only known software, i.e. a Linux distribution, and that all distribution packages are up-to-date.

Secondly, this service allows cloud administrators to monitor the status of the nodes in an efficient way and to take appropriate countermeasures once a compromised host has been detected. For instance, the administrators can isolate the host such that it can not attack other nodes of the infrastructure.

Further details can be found in Chapter 4.

Access Control as a Service (ACaaS) is a subsystem ensuring that user VMs are only executed on hosts matching their security requirements. Each node in the cloud infrastructure can be assigned a set of security properties expressed as pairs (key, value). When a user (i.e. a cloud tenant) specifies node requirements for the VM being started as pairs (key, value), the scheduler verifies the security properties of each node and selects for deployments only those matching the requested requirements for the VM.

ACaaS also provides an advanced scheduling criterion that allows to specify that a VM must not run on the same host where the VMs of specified users are currently running. This can guarantee a stronger isolation for a customer that could be offered by the cloud owner under a different Service Level Agreement. Further, the expected states of a host can be also defined as a scheduling criterion. Users can request ACaaS to deploy their VMs only on hosts with specific platform configurations (i.e. trusted properties): this is specified through a white list containing the digests of all files (binaries and configuration) that must be present on a target node. Also this capability is built on Trusted Computing and complements the scheduling of the VMs based on the integrity levels provided by the Remote Attestation Service.

Further details can be found in D2.4.2 [S+12a], Section 3.1.2.2.

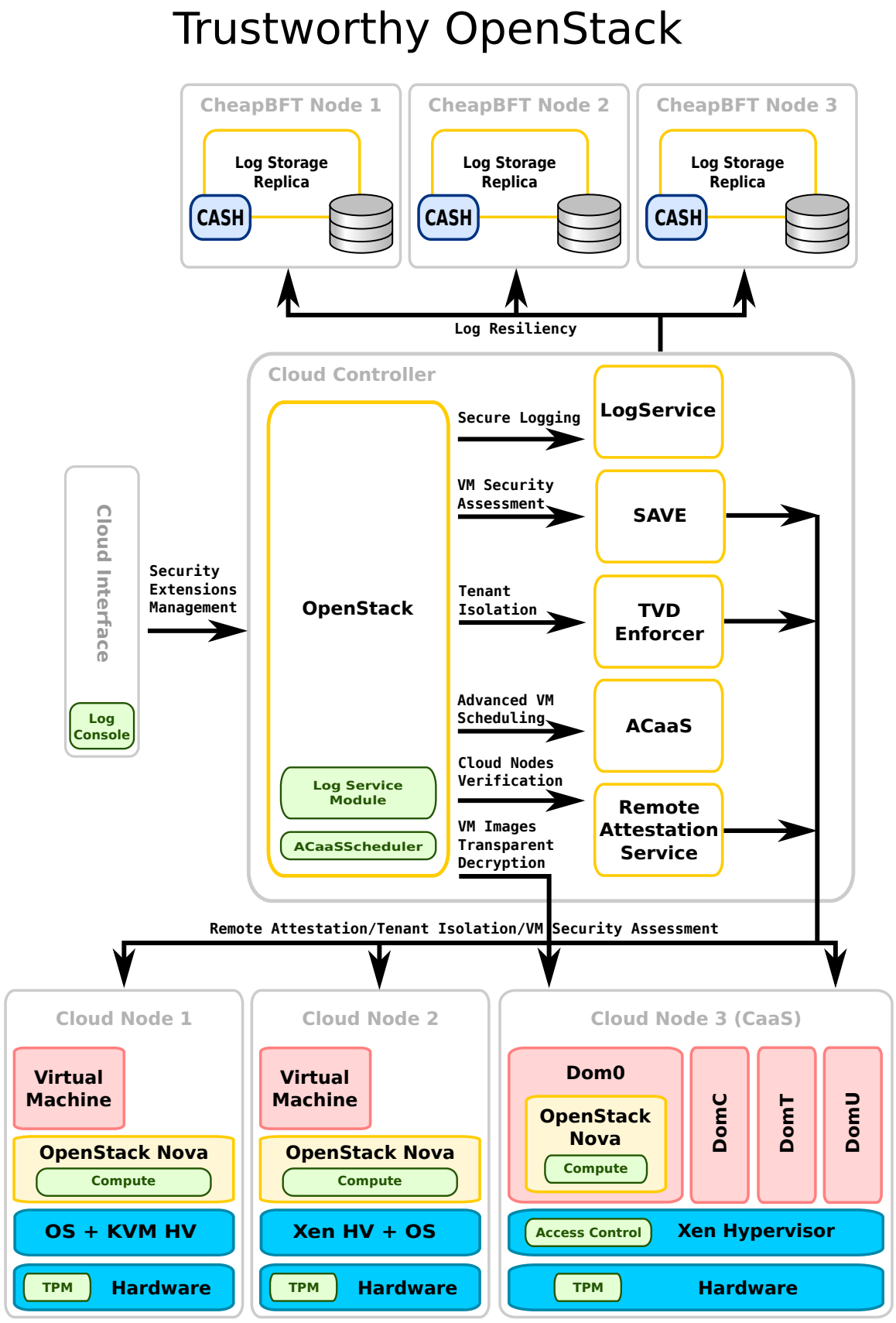


Figure 2.1: The Trustworthy OpenStack architecture

Cryptography-as-a-Service (CaaS) enables a VM to use an encrypted storage device transparently as if it were plaintext. On top of this capability CaaS provides the ability to bootstrap the VM from encrypted devices while still preserving confidentiality and integrity against the privileged administrator access and hence the cloud personnel and other customers. CaaS builds on Trusted Computing and uses the Trusted Platform Module to protect the customer encryption key for the VM image.

Further details can be found in D2.1.2 [S+12b], Chapter 3.

LogService is the subsystem that manages secure logging events in the TClouds cloud infrastructure. It provides confidentiality and integrity of each log entry and the forward integrity property, i.e. the capability to detect if log entries within a logging session have been deleted or reordered. Depending on the configuration, LogService can record all events produced by the components of Trustworthy Openstack, i.e. it operates at the IaaS layer, but it can also serve applications, thus operating at the PaaS layer. An auditor, internal or external, can always verify if a logging session has been tampered with, and if not, can decrypt the log entries and access to the log data.

Further details can be found in Chapter 5.

Resource-efficient BFT (CheapBFT) is a subsystem that implements the state machine replication scheme for Byzantine Fault Tolerance (BFT). This scheme normally entails very high resource consumption because $3f + 1$ actively operating replicas are required to tolerate f faults. CheapBFT, instead, is more efficient because, using a trusted device (a specialized FPGA module) and combining active and passive replicas, the number of actively involved replicas is lower: $f + 1$ in normal and error-free operations and $2f + 1$ under error condition. This subsystem stands at the PaaS layer of the TClouds Platform but it is also used in Trustworthy OpenStack (i.e. at IaaS layer) (see D2.4.3 [R+13], Chapter 2) to provide the LogService with resiliency. The LogService alone, in fact, is able to detect the corruption of log sessions but it cannot prevent it, i.e. it cannot guarantee availability of correct log data. This capability is, instead, provided by CheapBFT and the number (f) of faults tolerated depends on the chosen number of replicas ($2f + 1$).

Further details can be found in Chapter 6.

Ontology-based Reasoner/Enforcer (Enforcer) is a subsystem that enhances the capability provided by the standard OpenStack network management component (called Quantum) of isolating the tenants' virtual networks. The Enforcer builds on an enhancement of Libvirt that allows to define and configure a *Trusted Virtual Domain* (TVD): this is an aggregation of virtual machines that share a virtual network (at OSI level 2), which is isolated from similar networks of the other TVDs.

Further details can be found in D2.3.4 [B+13b] Chapter 6.

Security Assurance of Virtualized Environments (SAVE) is a subsystem developed for extracting configuration data from multiple virtualization environments to analyze their security properties. In particular SAVE support the specification of security policy that the cloud configuration must satisfy. It can be used with ACaaS to validate the correct deployment of the VMs according to the security properties requested by the customer. It can also be used with the Enforcer ACaaS to validate the isolation of the cloud customers.

Further details can be found in D2.3.4 [B+13b] Chapter 2.

Chapter 3

Trusted Infrastructure Cloud

Chapter Authors:Norbert Schirmer (SRX)

This chapter summarizes the high level objectives and the architecture of the Trusted Infrastructure Cloud (cf. D2.1.1 Chapter 12). It is a consolidated version of the documentation presented in other, mostly previous deliverables, revised with the necessary updates reflecting the work of the last period of the TClouds project.

3.1 Motivation

Cloud computing promises on-demand provisioning of scalable IT resources, delivered via standard interfaces over the Internet. Hosting resources in the cloud results into a shared responsibility between cloud provider and customer. In particular the responsibility for all security aspects is now shared. As the cloud provider hosts the customers resources and data, insiders like cloud administrators can access unprotected data of the customers. Moreover, as all cloud customers use the same resources, the infrastructure is shared among multiple tenants, which may be competitors. Hence proper isolation of cloud customers becomes of crucial importance for the acceptance of cloud offerings. With the Trusted Infrastructure Cloud we tackle these challenges by the following key properties:

- *Trust in remote resources* is established by building on top of Trusted Computing technologies, providing verifiable integrity of the remote components.
- *Protection against insider attacks* is achieved, as the administration is completely controlled by the infrastructure itself. All data is encrypted and there are no administrators with elevated privileges.
- With *Trusted Virtual Domains (TVD)* we provide trustworthy isolation of virtual computing, storage and networking resources as well as pervasive information flow control. TVDs are employed for isolation of tenants and for separation of security domains of a single tenant.

3.2 Architecture

In the Trusted Infrastructure Cloud a central management component, called TrustedObjects Manager (TOM), manages a set of TrustedServers (TS) which run a security kernel, which runs the virtual machines (VM) of the users. A virtual machine consists of the operating system (OS) and applications (App). This is depicted in Figure 3.1.

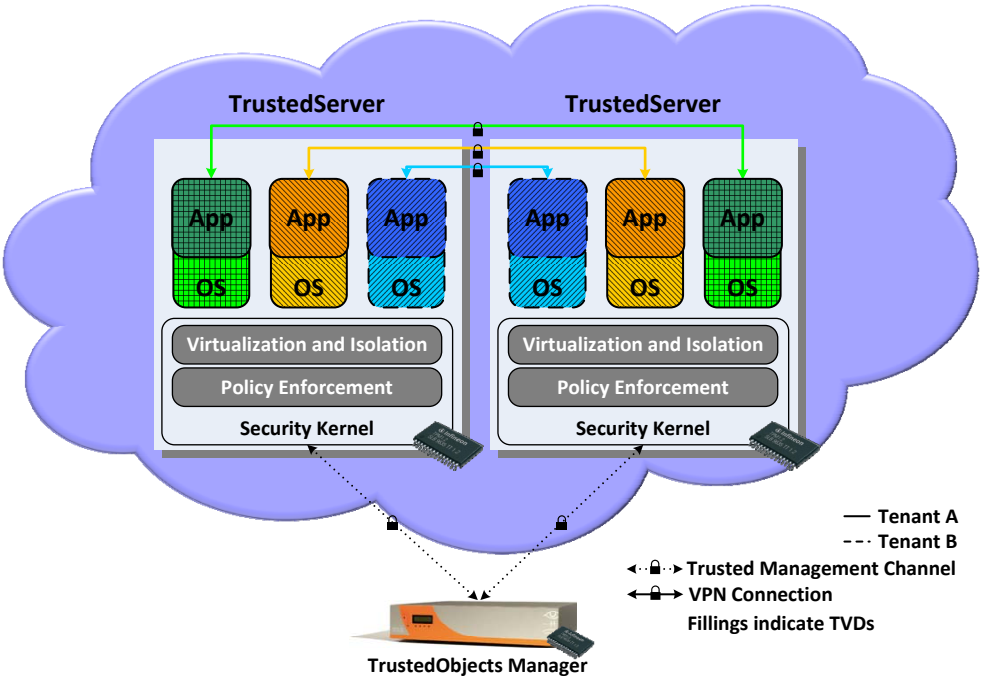


Figure 3.1: Trusted Infrastructure Cloud Architecture

Hardware security Anchor and Trusted Boot of Security Kernel The TrustedServer as well as the TOM, are equipped with a hardware security module (HSM) or Trusted Platform Module (TPM) [tcg]. When started, the HSM is employed for trusted boot, ensuring the integrity of the software (in particular of the security kernel). Moreover, the hard drives are encrypted by a key that is stored within the HSM. Via this sealing, the local hard drives can only be decrypted in case the HSM has crosschecked the integrity of the component. Hence only an untampered security kernel can be booted and can access the decrypted data. Once the security kernel is booted it enforces the security policy and the isolation.

Trusted Management The TOM is in charge to deploy configuration data (including key material and security policies) and VMs on the TrustedServers (cf. D2.3.1 Chapter 6). Security services within the security kernel of the TrustedServer handle the configuration and ensure that the security policies are properly enforced. Encrypted communication of TOM and TrustedServer is via the Trusted Management Channel (TMC) which ensures the integrity using the remote attestation feature of the HSM before transmitting any data. This ensures that the TOM will only transmit data to untampered TrustedServers. Moreover, the TrustedServers are bound to a distinct TOM during their deployment (cf. D2.1.2 Chapter 6), such that a TrustedServer only accepts this TOM as a management component. All administrative tasks on the TrusedServer are performed via the TMC, there is no other management channel for an administrator (like an ssh-shell with elevated privileges).

Trusted Server In Figure 3.2 the architecture of a TrustedServer is presented in more details. The security kernel comprises the isolation kernel and the hypervisor to provide isolation between virtual machines and a set of security services managing file encryption, TVDs, VPN, the confidentiality proxy for cloud storage (S3 Proxy) , remote attestation and the TMC. All

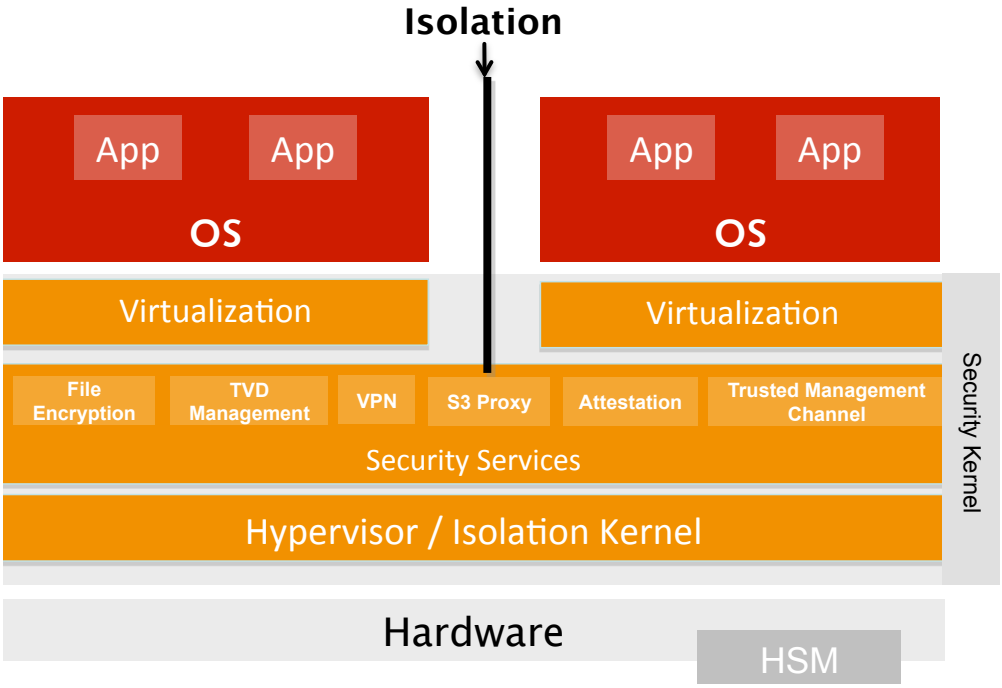


Figure 3.2: TrustedServer Layers

these services are part of the security kernel and belong to the trusted computing base of the TrustedServer. These are the components that are measured by the HSM during the trusted boot. Only if all of them are untampered, the server will boot up. In contrast, everything happening inside the VMs (OS and Apps) does not belong to the Trusted Computing Base. This means that a misbehaving (or infected) VM has no negative impact on the security guarantees of the TrustedServer. For example, the isolation properties of TVDs are not depending on the VM but only on the security kernel.

Trusted Virtual Domains Trusted Virtual Domains (TVD) [CLM+10] allow to deploy isolated virtual infrastructures upon shared physical computing and networking resources. By default, different TVDs are isolated from each other. Communication is restricted to virtual machines within the same TVD and data at rest is encrypted by a TVD specific key. Remote communication between components of the same TVD over an untrusted network are secured via virtual private network (VPN). Only virtual machines of the same TVD, which have access to the same TVD key, are able to communicate and decrypt data. A TrustedServer can simultaneously run various VMs attached to different TVDs. Figure 3.1 illustrates that each tenant runs his own set of TVDs (indicated by the fillings of the VMs), ensuring isolation of tenants. A single tenant (cf. Tenant A) can also run distinct TVDs (indicated by the fillings of the VMs), to isolate domains within his organisation, e.g. to isolate the 'human ressources' department from 'product devolopment' department.

Cloud Storage Encryption The TrustedServer also supports the transparent encryption of commodity cloud storage like Amazon S3 (cf. chapter 8) which is hosted in another cloud, outside of the VM and the Trusted Infrastructure Cloud. From within a VM the plaintext data is accessible via the ordinary file system. Not the VM but the security kernel takes care of

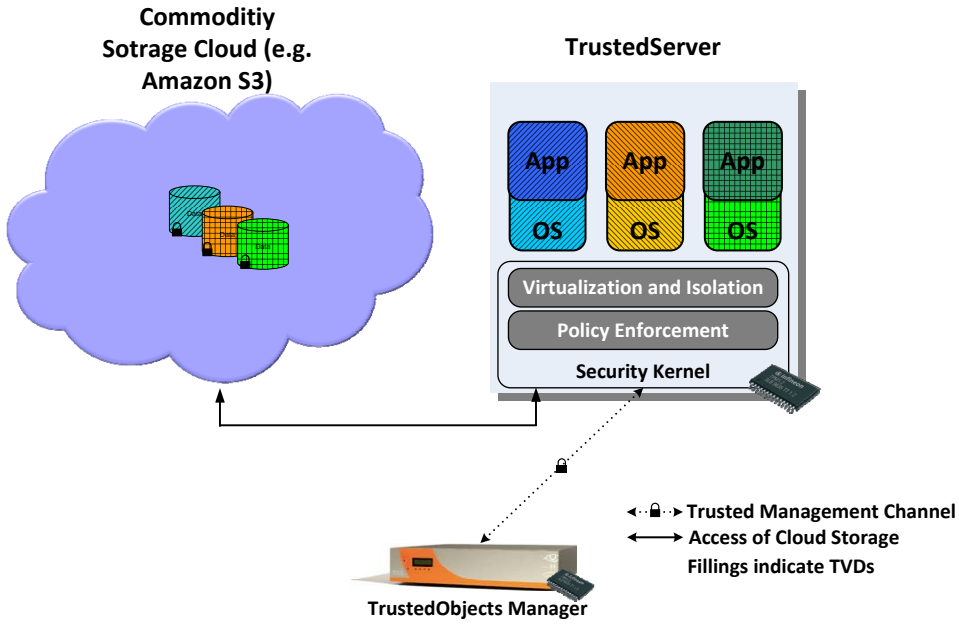


Figure 3.3: Transparent Encryption of Commodity Cloud Storage

the TVD specific encryption of the data before it moves it to the commodity cloud storage (cf. Figure 3.3). That way we can securely extend the boundaries of the TVDs to commodity cloud storage services.

Beyond the Cloud: Trusted Endpoints The concept of TVDs within the Trusted Infrastructure Cloud allows the trustworthy isolation of domains within the cloud infrastructure. This means that information flow is confined within the TVD. To securely access the TVDS from outside the cloud this concept can also be extended to the endpoints, e.g. desktop / laptop computers of the user to obtain complete end-to-end security. This is depicted in Figure 3.4 on the example of a TrustedDesktop. We laverage this scenario in our demonstration of the smart lighting scenario. While updates of the sensitive data in the cloud can only be made from within the TVD from a TrustedDesktop, reading the data is open for public access. To also support mobile devices as trusted endpoints for TVDs was TClouds research that was described in D2.1.2 Chapter 5.

3.3 Conclusion

The Trusted Infrastructure Cloud is constructed from ground up with security and trustworthiness in mind, employing trusted computing technologies as a hardware anchor. With trusted boot and remote attestation we ensure that only untampered servers with our security kernel are started and that the sole way of administration is via the trusted channel from the management component TOM. Hence no administrator with elevated privileges is necessary and hence this functionality is completely disabled, abandoning the possibility for an administrator to corrupt the system. The costs of the security measures of the Trusted Infrastructure Cloud can be

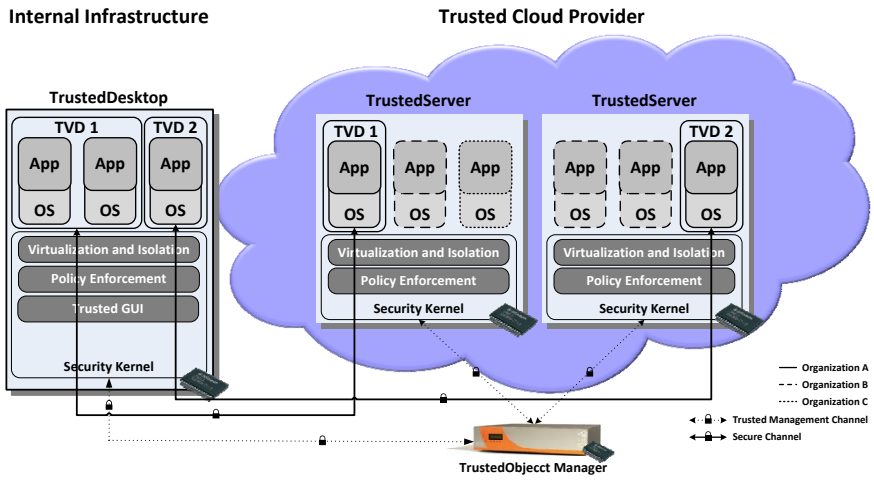


Figure 3.4: Extending the Trusted Infrastructure to Endpoints

measured in two dimensions: the cost of additional hardware (e.g. Trusted Computing) and the cost of ressources, like performance or bandwidth overhead. The Trusted Infrastructure Cloud is designed to work with "custom of the shelf hardware". The hardware components like TPM or native cryptographic support of the CPU (e.g. Intel AES NI) are nowadays standard. For example, a TPM chip accounts to only 1-2% of the hardware costs.

Regarding the ressource costs we have additional efforts for trusted boot / remote attestation and the encryption (VPN and files). The costs for trusted boot and remote attestation are negligible, as these only account during bootup of the TrustedServer, which is a rare event as a server is supposed to have a long uptime. During the operation of the server there is no additional cost. Both the network encryption (VPN) as well as the disk encryption leverage the native cryptographic support of the processor [int10] which minimizes the performance overhead. The bandwidth overhead of the VPN which is based on IPSEC is about 10%. Altogether this cost figures illustrate that the security benefits of the Trusted Infrastructure Cloud come with only low additional costs.

Part II

TClouds Subsystems

Chapter 4

Remote Attestation Service

Chapter Authors: Roberto Sassu, Nicola Barresi, Gianluca Ramunno (POL)

During the second year of the project POL developed a new subsystem, the *Remote Attestation Service* that has been delivered at the end of that year. A quick overview on such service will be given in Section 4.1. The service has been updated during the third year with the following enhancements: generation of smaller integrity reports (see Section 4.2), possibility to define new analysis types (see Section 4.3) and support for Ubuntu distributions (see Section 4.4).

4.1 Architecture summary

From Section 3.1.2.1 of deliverable D2.4.2 [S+12a] we recall here a summary of the architecture of the first version of the *Remote Attestation Service* delivered at the end of the second year.

The *Remote Attestation Service* (*RA Service*) is a cloud subsystem responsible to assess the integrity of the nodes in the cloud infrastructure through techniques introduced by the Trusted Computing technology.

This service gives significant advantages in the cloud environment. First, it allows cloud users to deploy their virtual machines in a physical host that satisfies the desired security requirements, represented by five integrity levels. Requiring a higher level will give more confidence and trust into the used physical hosts.

Second, this service allows cloud administrators to monitor the status of the nodes in an efficient way and to take appropriate countermeasures once a compromised host has been detected. For instance, the administrators can isolate the host such that it can not attack other nodes of the infrastructure.

A high-level architecture of *RA Service* is depicted in Figure 4.1 and it consists of two main components (*OpenAttestation* and *RA Verifier*), interacting with the *OpenStack* modules. The *OpenStack* Nova Scheduler takes users' requirements as input and selects the proper host where to deploy a virtual machine depending on the platforms evaluation done by the *RA Service*.

OpenAttestation. This framework, developed by Intel, enables the *OpenStack Nova Scheduler* to retrieve and verify the integrity of cloud nodes such that the former can select a host that meets the users requirements. The framework handles the remote attestation protocol through two submodules that act as the endpoints: *HostAgent* collects the measurements done by the attesting platform, generates and sends the integrity report to the verifier; *Attestation HTTPS Server* (implemented by three main Java components: *HisWebService*, *HisAppraiser* and *AttestationService*) verifies the integrity report received from a cloud node and assigns to the latter an integrity level. *OpenAttestation* has been enhanced by POL to support the reporting of the

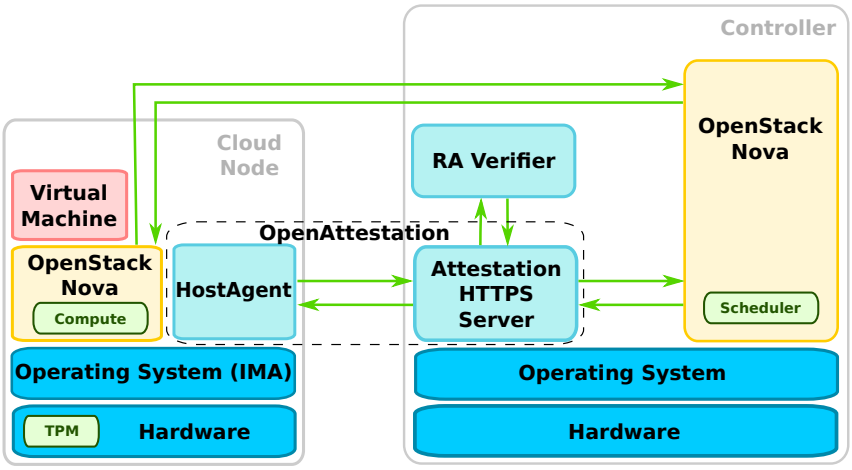


Figure 4.1: *Remote Attestation Service* architecture

measurements taken by *Integrity Measurement Architecture* (IMA), a subsystem of the Linux Kernel running on cloud nodes.

RA Verifier. This component analyses the measurements performed by IMA. In particular, it verifies whether the digest of binary executables and shared libraries are present in a database of known values and whether the packages these files belong to are up to date. The first check allows to detect possibly malicious software that may have been executed before verification, while the second check allows to identify loaded applications with known vulnerabilities that may be exploited by an attacker. Details of the *RA Verifier* can be found in Chapter 4 of deliverable D2.1.2 [S+12b].

In TClouds the *RA Service* is integrated with *OpenStack* through the *TrustedFilter* scheduler filter, introduced by Intel developers in Folsom and modified by POL to support the newly defined integrity levels: this is a portion of the *Trustworthy OpenStack* prototype v1 delivered at the end of the second year. However the *RA Service* is independent from *OpenStack*, therefore it can be used with other cloud frameworks.

4.2 Update: Integrity Reports Optimization

This optimization mitigates the scalability issue that may arise in large cloud deployments due to the significant size of integrity reports sent by *Cloud Node*s to the *Controller* node (reported respectively with CN and CT abbreviations in the sequence diagrams). Indeed, the *OpenAttestation* component with POL customizations developed during the second year generates integrity reports of about 140 KB while the original version produced reports of 4 KB.

The reason for this increased size is the inclusion in the reports of the IMA measurements that allow to have a more comprehensive information of the integrity status of *Cloud Node*s. However, sending large reports to the verifier will cause the saturation of the resources available at the *Controller* node (that performs the report verification). In particular, this issue affects the storage, as the integrity reports are saved in a database, and the network because the *Controller* periodically receives data from thousands of nodes.

In the version 2.x of the *Trustworthy OpenStack* prototype, we modified *OpenAttestation* in a way that *Cloud Node*s include in an integrity report only the IMA measurements that were not

previously sent to the verifier. It is task of the latter to reconstruct the whole report from the partial ones before it determines, together with *RA Verifier*, the integrity of the sender.

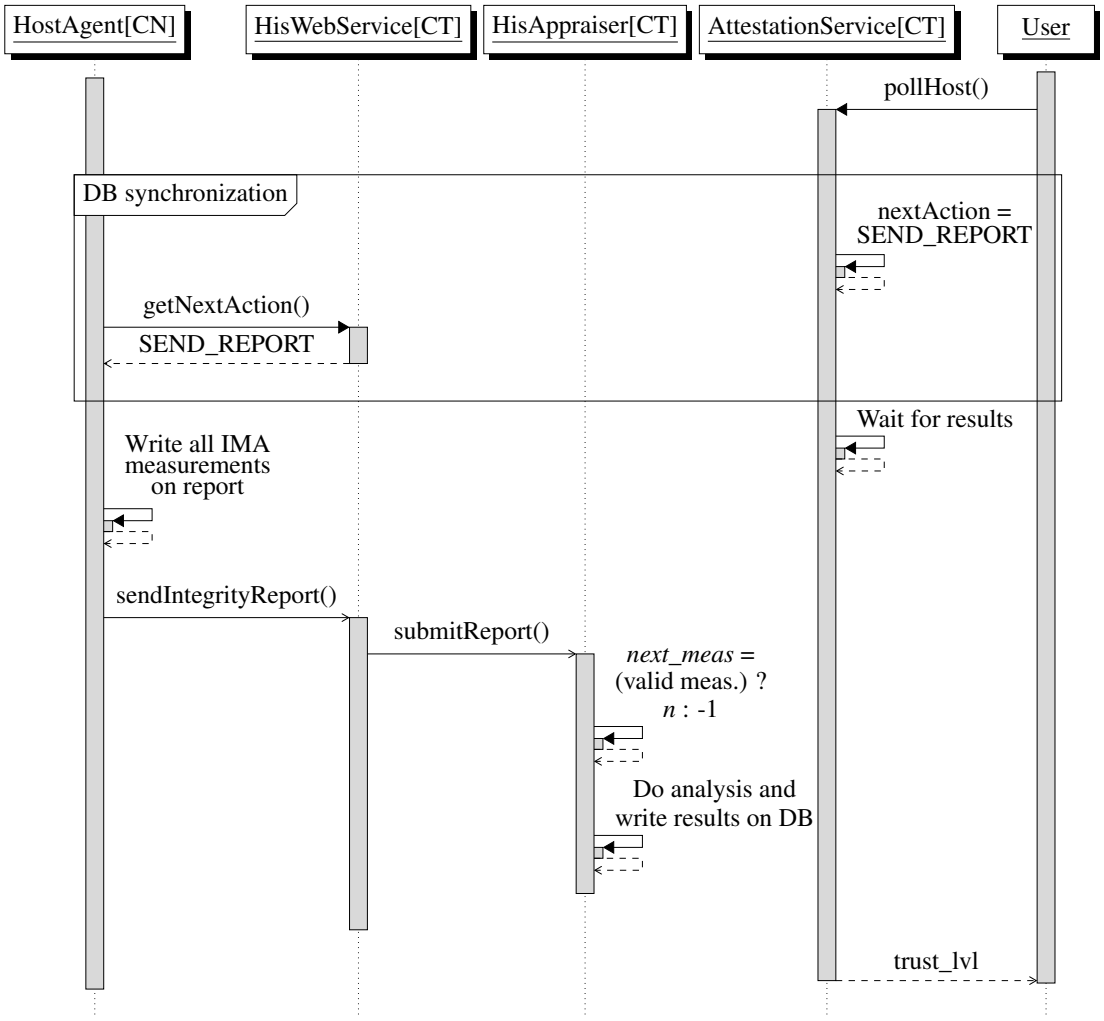


Figure 4.2: Optimization mechanism if the Controller can't find a report for the node

As in the version 1.0 of the *Trustworthy OpenStack* prototype, the remote attestation protocol is initiated by the *OpenStack* scheduler that, after extracting users' integrity requirements from VM specifications, tries to find a suitable *Cloud Node*s to start requested VMs. In the version 2.x, POL introduced an optimization mechanism that allows the reconstruction of the whole report by the *Controller* from partial ones. This mechanism depends on whether the target *Cloud Node* previously sent an integrity report or not. The two possible cases are described below through sequence diagrams, where actors are: *HostAgent*, running on a *Cloud Node*; *Attestation HTTPS Server*, running on the *Controller* node (split into its main three Java classes: *HisWebService*, *HisAppraiser* and *AttestationService*); *User* (the *OpenStack* scheduler), acting in the *Controller* node.

Case 1: no previous reports received. If the target *Cloud Node* did not send any report at the time of an attestation request (see Figure 4.2), the *Controller* sets `nextAction` (a value in the database that indicates the next action that should be performed by a *HostAgent*) equal to `SEND_REPORT` to ask the *Cloud Node* to include in the report all the measurements produced by IMA. In this case, since the required information is present in the report sent by the *Cloud*

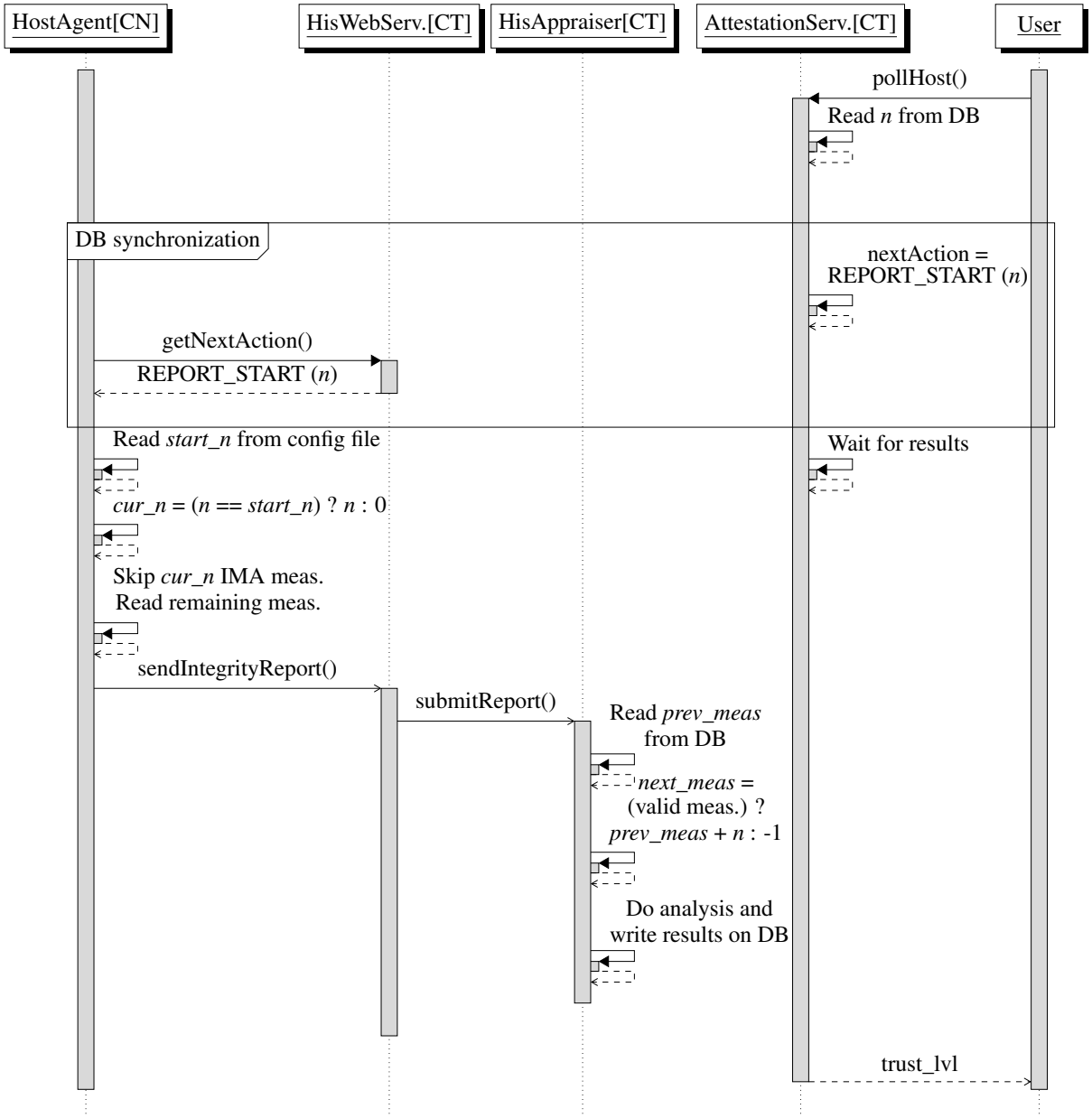


Figure 4.3: Optimization mechanism if the *Controller* already owns a report for the node

Node, the *Controller* can launch the analyses requested by the *User* without doing additional work.

Case 2: reports available. If the target *Cloud Node* already sent one or more integrity reports, the *Controller* node can rely on them for subsequent validations and will request to report only the new measurements. In this case, the *Controller* (see Figure 4.3) sets `nextAction` to `REPORT_START` and the next measurement (in the IMA measurements list) that it expects to receive.

The *Cloud Node* verifies whether the number of the first measurement requested by the *Controller* (`n`) matches the number of measurements it actually sent (`start_n`). If so, the sender skips `n` measurements and fills the report with the remaining measurements; on the other side, the *Controller* will reconstruct the whole report from the partial ones before verifying all

IMA measurements.

If the *Controller*'s request does not match sender's expectations, some parts of the whole report may have been lost. In this case the *Cloud Node* sends the entire list of measurements produced by IMA, so that the *Controller* will be able to complete the integrity verification.

In any case, the `TPM_Quote()` operation covers all measurements: in fact the TPM signature is performed as usual, i.e. it covers all PCRs, and among them `PCR#10`, where all IMA measurements are accumulated.

As a result of this work, we experienced that, while the first integrity report sent by a *Cloud Node* is large, the subsequent ones are comparable in size to those generated by the original unmodified version of *OpenAttestation*. Thus, the new version of the *Remote Attestation Service* offers now the same features of the previous one while requiring lower resources to perform its tasks.

4.3 Update: Definition of New Analysis Types

The main goal of *OpenAttestation* developers is to provide a generic SDK for the remote attestation which vendors of cloud software stacks can extend and integrate into their products. However, even if it is extensible, *OpenAttestation* offers limited functionalities, i.e. it only performs a very basic integrity verification: it checks the presence of TPM register (PCR) values, collected from the attesting platform in a white list, and compares the current report with the previous one to find what PCRs have been changed since the last verification.

POL enhanced *OpenAttestation* to include and verify IMA measurements with the *RA Verifier* tool. The latter assigns to each host an integrity level among five depending on whether the digest of software executed on a *Cloud Node* is in a database of known values and the related packages are up to date or not.

While the *RA Service* is strongly tied to *RA Verifier* to determine the integrity status of a platform, it would be useful to allow users define their own analysis types in *OpenAttestation*, so that an integrity report can be evaluated from different perspectives. For example, a different analysis method could be used to verify whether software executed belongs to a trusted configuration, meaning that digests of binaries and libraries are part of a set of packages known to be secure (i.e. they were not affected by serious vulnerabilities in the past).

In this context, during the third year of the project POL further enhanced *OpenAttestation* to define and perform user-defined analyses on integrity reports generated by *Cloud Node*s. This feature has been made available by defining a new API command `analysisTypes` and modifying the parameters of `PollHosts`, the command to request a remote attestation for the specified hosts. Both commands can be called through a RESTful interfaces exposed respectively by the *HisAppraiser* and *AttestationService* modules of the *Controller*.

```
{
"analysisTypes": [{
                "deleted": "false",
                "inputParameters": "[1-4],>|>=",
                "maxOutputSize": "10",
                "module": "IMA",
                "name": "IMA_LEVEL",
                "outputParameters": "[tT]rue|[fF]alse",
                "URL": "python /usr/bin/ra_verifier.py -t openattestation",
                "version": "1"
                }]
}
```

Listing 4.1: Sample response to GET method on `analysisTypes`

The Listing 4.1 shows a sample response to the GET method executed on the newly introduced `analysisTypes` command. Among others, the most important fields are: `inputParameters` and `outputParameters` contain a regular expression for the input passed to the analysis software and for the output returned by the latter; `module` and `version` define respectively name and version of the analysis software; `name` contains a type of analysis among those supported by the analysis software; `URL` contains path name and parameters of the program to be executed for the defined analysis type. Since some parameters (e.g. the analysis to perform) cannot be included in the URL, as they vary depending on users' input, *OpenAttestation* passes the required information through three environment variables set just before the execution of the verification script. The first two environment variables are:

`ANALYSIS`: types of analyses to be performed by the verification script and associated parameters;

`IMA`: absolute pathname of the file with IMA measurements extracted from the integrity report.

while the third one, `OS`, is explained in Section 4.4.

```
{
        "hosts": [
                        "node-109",
                ],
        "analysisType": "IMA_LEVEL,1,>="
}



{
        "hosts": [
                        {
                        "URL": "http://node-108/OAT/report.php?id=23",
                        "details": [
                                        {
                                        "analysis_name": "IMA_LEVEL,1,>=",
                                        "result": "true"
                                        }
                                ],
                        "host_name": "node-109",
                        "trust_lvl": "trusted",
                        "vtime": "2013-02-20T20:01:14.067+01:00"
                        }
                ]
}
```

Listing 4.2: Sample request and response to POST on `PollHosts`

The Listing 4.2 illustrates a sample request and response to the POST method executed on the command `PollHosts`. Such request, made for the host `node-109`, tells *OpenAttestation* to perform the analysis `IMA_LEVEL`, supported by *RA Verifier*. The input parameters `1` and `>=` after the analysis name mean that the result of the verification will be positive only if the integrity level determined for the `node-109` host is greater or equal to `1` (the complete level identifier is `l1_ima_digest_notfound`). From the response, it is possible to see in the `details` field that the result of the analysis was positive.

Figure 4.4 details the steps (performed by the *HisAppraiser* module of the *Controller* node) corresponding to the action *Do analysis and write results on DB* represented in the sequence diagrams of Figures 4.2 and 4.3; these steps depend on some previously executed actions also represented in those diagrams. During the execution of the function `submitReport` called by *HisWebService*, before executing the action *Do analysis and write results on DB*, *HisAppraiser* verifies the integrity of IMA measurements from the sender, eventually reconstructing the

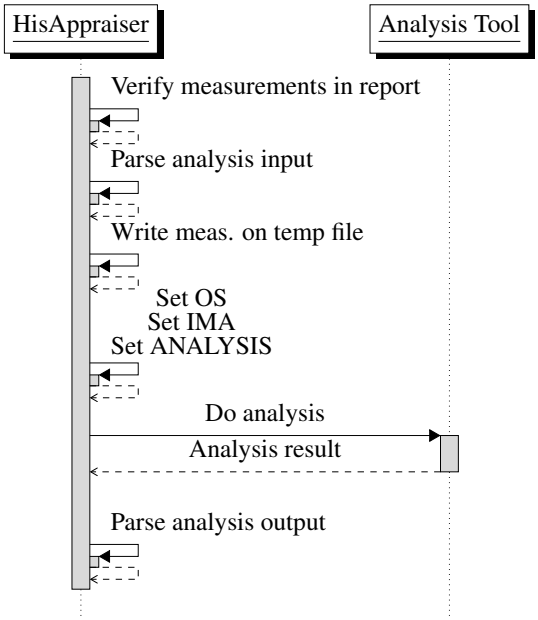


Figure 4.4: *Controller* behaviour on analysis request

whole report as described in Section 4.2. Then *HisAppraiser* uses the data associated to the `analysisType` previously specified by *User* when calling the command `PollHosts`: see the sample request in Listing 4.1 where the analysis type to be performed is `IMA_LEVEL >=1`. *HisAppraiser* first parses the analysis input by using the regular expression defined with `inputParameters` and sets the three environment variables mentioned above. Secondly, the *Controller* calls the analysis tool from `URL` and parses its output by using the regular expression defined with `outputParameters`.

4.4 Update: Support for Ubuntu distributions

The *Remote Attestation Service* has been enhanced to support Ubuntu distributions, in addition to Fedora distributions. This new feature is required as the TClouds *Trustworthy OpenStack* prototype is built on top of the Ubuntu 12.04 LTS distribution.

We performed the following modifications with respect to the version released last year:

Creation of new Cassandra DB comparator: we provide a new library for the Cassandra database to sort versions of packages from the oldest to the most recent. This library comes from the source code of the dpkg tool, which is used by the Ubuntu package manager (APT) to determine when a package must be updated. The library has been implemented by wrapping existing functions from dpkg with the function `filevercmp_deb()`, which is called by Cassandra to perform the ordering.

Database structure modifications: in the previous version of the *Remote Attestation Service*, two column families (like tables in relational databases) were defined: `FilesToPackages`, to store the mapping between a digest and the information associated to the file the digest was taken from (full pathname, list of linked libraries, libraries aliases and packages that contain that file); `PackagesHistory`, to store the update type of each package released by the distribution vendor. In the current version we added a new column family called `PackagesHistoryDEB` that, similarly to the latter, contains the history of packages

released for Ubuntu distributions only. With the addition of this new column family, we can set in the Cassandra database configuration the correct comparator for each column family storing packages history, i.e. the RPM comparator for `PackagesHistory` and the DEB comparator for `PackagesHistoryDEB`.

Enhancement of *DB insert* scripts: the *DB insert* scripts have been enhanced to also process DEB packages. In particular, the following functions have been modified:

- Parsing of metadata from packages header (source package name, source package version and release, processor architecture of the binary package);
- Extraction of files from packages (through rpm2cpio and cpio for RPMs, through dpkg for DEBs);
- Writing of the collected data to the database.

To minimize the changes to the existing scripts, we moved package-specific code to separate python libraries (one for RPMs and one for DEBs) which are called from the main *DB insert* script depending on the package being processed.

Modified interface between *OpenAttestation* and *RA Verifier*: to perform the verification, it is necessary to pass the distribution name to *RA Verifier* so that the latter can select the proper column family for the packages history query. *OpenAttestation* provides this information to *RA Verifier* (in addition to those introduced in Section 4.3) by setting the environment variable `OS` – taken from the Measured Launch Environment (MLE) associated to the platform being attested – before calling the verification script.

Chapter 5

Log Service

Chapter Authors: Paolo Smiraglia (POL)

LogService is a subsystem in the Trustworthy OpenStack platform offering secure logging features. It allows the generation of secure log entries in order to track the events occurring in the platform at different layers with the purpose of increasing the trustworthiness of the whole cloud infrastructure. The LogService, here depicted in Figure 5.1, is composed of four modules:

LogCore: the trusted entity collecting all cryptographic material necessary to preserve the confidentiality of the tracked information and to execute the integrity verification of the logs.

LogStorage: the module playing the role of storage. The read and write operations could be performed over plain text files or by interfacing the LogStorage with the fault tolerant storage system CheapBFT [KBS+12].

LogConsole: it is a web based management console to access the LogService features. It is available in form of standalone web application or as a dedicated administration tab integrated within the Trustworthy OpenStack dashboard.

LogService Module: it is a software module that developers must use in their applications in order to have secure logging features.

LogService has been already presented in Chapter 9 of the deliverable D2.1.2 [DKSP+12]. Therefore, this chapter will contain only the descriptions of the enhancements implemented and available in the latest version.

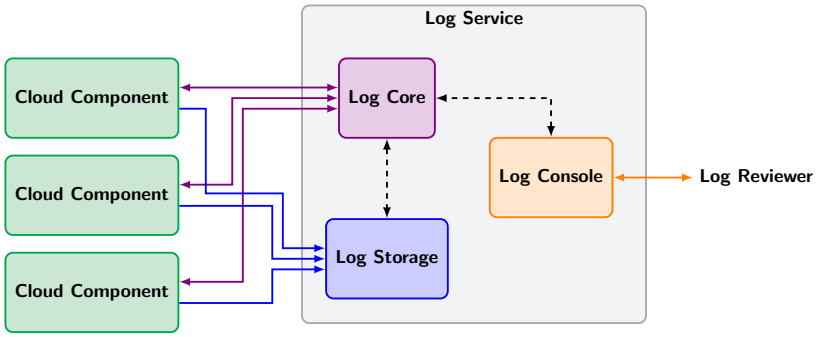


Figure 5.1: LogService high level view

5.1 New features

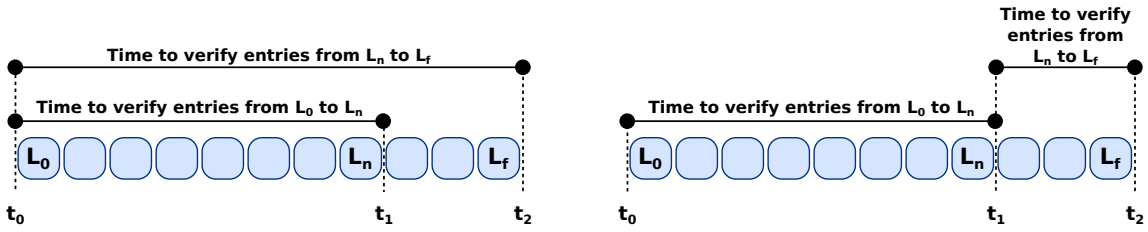
The new version of the LogService introduces three new features. The first aims to mitigate some issues related to the scalability of the service, while the second enhances the security in the communication among the entities composing it. Finally, the third feature is the replacement of the core library which the LogService is founded on.

5.1.1 Incremental and Asynchronous verification

The verification of a logging session is the process allowing the identification of potential corruptions at client side. We consider a session corrupted if some entries are deleted or properly replaced by an attacker, in order to hide a malicious behaviour. In the first version of the LogService, the verification process was affected by scalability issues related to two aspects.

The first is the synchronous serving of the verification requests that locked the LogConsole (or the OpenStack dashboard) and hence made it not usable until the end of process. To avoid this, the new LogService serves the verification request through a message queuing system. Thus, each time that a verification requests is received, it is added to the queue in order to be subsequently scheduled without locking the dashboard.

The second aspect causing issues in the LogService scalability was the absence of a mechanism allowing the *incremental verification* of the logging sessions. For instance, considering the case where immediately after the end of the verification of a session containing the entries $[L_0, \ldots, L_n]$ the new entries $[L_{n+1}, \ldots, L_f]$ are generated, in absence of a mechanism allowing the incremental verification, to verify the new entries will be necessary to re-verify all the entire session (see Figure 5.2(a)). Such problem is now mitigated because the LogService implements a simple caching system that stores for a limited period of time the metadata (last computed hash, ...) related to logging session verification. Such approach allows the verification only of the *session delta* represented by the entries that have been added during the last verification (Figure 5.2(b)).



(a) Without incremental verification

(b) With incremental verification

Figure 5.2: Logging session verification procedure

5.1.2 Secure communication

To enforce the security in the communication among LogService modules and also with the components interacting with the LogService, all the connections are over HTTPS with both client and server authentication. In addition, each module filters the incoming connections using white lists that are filled in with the subject of the X509 certificates identifying the authorised entities.

5.1.3 New core library

The library being the core of the LogService has been renewed. The definition of a new library was necessary due to the monolithic code structure, the high number of external dependencies, the low adaptability and finally, the low testing coverage of the old core library. Therefore, a new library called `libseclog` has been designed and implemented. Such library has been newly developed and represents a deep restyling of the `libsklog` library used as core in the old version of the LogService. The main feature of the new library is the provision of a high level API allowing the developers to access to different secure logging schemes through the same set of functions. The current version of the `libseclog` only supports the scheme proposed by Schneier and Kelsey [SK99].

5.2 Design and implementation

5.2.1 Building blocks

Log Storage

The Log Storage is the component of the LogService playing the role of storage system. It is deployed as RESTful web application implemented through the framework Python Tornado. The Log Storage provides two methods: **`/store?<ARGS>`** and **`/retrieve?<ARGS>`**. The former allows the clients to store a block of log entries related to a specific logging session, while the latter makes the clients capable to retrieve all log entries contained in a specific logging session. A logging session is identified through a tuple of three values that will be encoded in the URL. The values in the tuple are: the ID of the machine generating the log entries (**`mid`**), the ID of the logging session (**`sid`**) and the human comprehensible label describing the session (**`label`**). Figure 5.3 shows how the URL will be generated after the encoding.

```
# example of /store
/store?mid=<MACHINE_ID>&sid=<SESSION_ID>&label=<LABEL>

# example of /retrieve
/retrieve?mid=<MACHINE_ID>&sid=<SESSION_ID>&label=<LABEL>
```

Figure 5.3: LogSorage example API call

The current implementation of the Log Storage supports two storage mechanisms. The first uses as storage medium plain text files in the local file system, while the second exploits the capabilities of CheapBFT [KBS+12], a fault tolerant storage framework. About the second mechanism, the Log Storage acts as pass through service between the clients and the CheapBFT endpoint. From the client point of view, using plain text files or CheapBFT is totally transparent. A graphical representation of the Log Storage architecture is depicted in Figure 5.4.

Log Core

The Log Core is the trusted entity of the LogService subsystem. Its role consists in collecting the cryptographic material necessary to perform the integrity verification of the logging session, as well as, to provide the client a trusted dump of the session content. It has been implemented using Python Tornado and runs as standalone web application accessible through a RESTful API

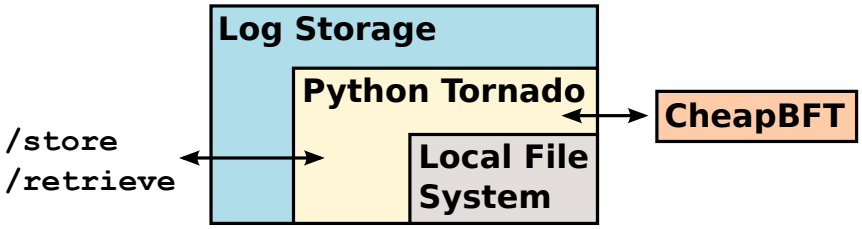


Figure 5.4: Log Storage architecture

Method	Description
`/initialize`	Requests the initialization of a new logging session.
`/retrieve`	Retrieves a list containing the tuples identifying the initialized logging sessions.
`/verify?<ARGS>`	Requests the verification of a logging session identified by the tuple (*machine_id*, *session_id*, *label*) encoded in **<ARGS>**.
`/verify/<ID>/update`	Requests the update of the verification identified by **<ID>** (incremental verification)
`/verify/<ID>/dump`	Requests a trusted dump of content included in the verification identified by **<ID>**.
`/verify/<ID>`	Retrieves the details about a specific verification identified by **<ID>**.
`/verify/status`	Retrieves information about the status of the verification still active.

Table 5.1: Log Core API

described in Table 5.1. In the context of the Log Core, the verification of a logging session is managed through an object identified by an **ID**. In addition to the information about the status, the verification object contains also the plain text version of the log entries (the dump) included in the logging session which it is related to.

As previously mentioned, the new version of the LogService serves the verification requests asynchronously. Such functionality is implemented through Celery [**?**], a tasks queuing framework for Python. The internal Log Core design is depicted in Figure 5.5.

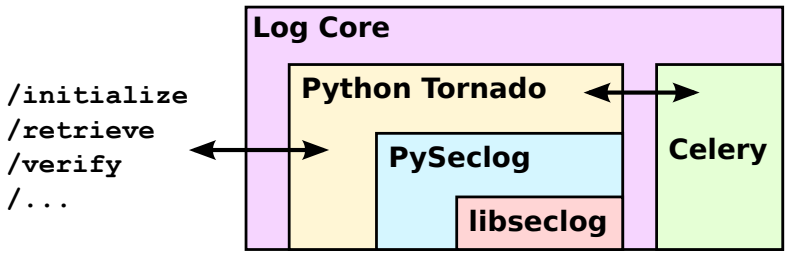


Figure 5.5: Log Core architecture

Log Console

The Log Console is the web based log management console provided by the LogService. Such component is available in two forms. The first one is a standalone web application accessible via

web browser. The second one is a log administration panel within the OpenStack dashboard.

Log Service Module

The LogService Module is a software module that must be included within the application aiming to interact with the LogService. The current release is in form of a Python package that provides two classes (**`LogStorageClient`**, **`LogCoreClient`**) for implementing client applications for both Log Core and Log Storage. Moreover, within the package it is also available a Python Logging Handler (**`SecureLoggingHandler`**), allowing the developers to implement applications performing log using the LogService capabilities.

5.2.2 Integration in OpenStack "Folsom"

LogService is fully integrated in Trustworthy OpenStack which is a security enhanced version of standard OpenStack "Folsom". The integration includes the definition of a Python Logging Handler[1] to manage secure logging features in the OpenStack core service (Nova) and the definition of a logging administration tab in the dashboard (Horizon).

Core Service (Nova)

Figure 5.6 shows the details about the integration of the LogService in the Trustworthy OpenStack core service. To perform logging, Trustworthy OpenStack uses the Python Logging framework. Therefore, the integration of the LogService in Trustworthy OpenStack consists in including the **`SecureLoggingHandler`** provided by the Log Service Module, within the Trustworthy OpenStack logging handler list.

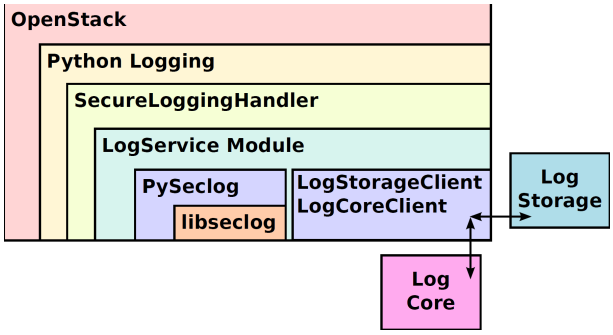


Figure 5.6: LogService integration in OpenStack Folsom

The new handler could be configured through a configuration flags group called **`secure_logging`**. This group includes several configuration flags that are listed and described in the Table 5.2.

Figure 5.7 depicts a snippet of the main Nova configuration file. In detail, the snippet highlights how the **`secure_logging`** group could be used.

Dashboard Service (Horizon)

The integration of the LogService administration tab in the Trustworthy OpenStack dashboard (Horizon) includes the definition of some additional settings organised in two dictionaries. The first one, `LOG_CLI_SSL_OPTS`, includes the SSL settings while the second one, `LOG_CORE`,

[1] Python Logging Handlers - http://docs.python.org/2/library/logging.handlers.html

Configuration flags group name		`secure_logging`
Flag	**Value**	**Description**
`use_secure_log`	`BOOLEAN`	Enables the secure logging handler.
`logcore_address`	`STRING`	Specifies the Log Core listening address. It could be expressed as an IP address or a FQDN.
`logcore_port`	`INTEGER`	Specifies the Log Core listening port.
`logcore_cert`	`STRING`	Specifies the path of the certificate containing the Log Core's public key.
`logstorage_address`	`STRING`	Specifies the Log Storage listening address. It could be expressed as an IP address or a FQDN.
`logstorage_port`	`INTEGER`	Specifies the Log Storage listening port.
`cert`	`STRING`	Specifies the path of the certificate containing the Node's public key.
`key`	`STRING`	Specifies the path of the file containing the Node's private key.
`cacert`	`STRING`	Specifies the path of the file containing the CA certificate.
`freq`	`INTEGER`	Specifies the log entries uploading frequency expressed in number of entries.
`size`	`INTEGER`	Specifies the size of the logging session expressed in number of entries.
`debug`	`BOOLEAN`	Enables debugging for the secure logging handler.
`secure_log_services`	`STRING`	Specifies the Nova services (comma separated values) for which the secure logging handler is enabled.

Table 5.2: Configuration flags of the `secure_logging` group

```
[secure_logging]
use_secure_log = True
logcore_address = logcore.example.com
logcore_port = 50001
logcore_cert = /path/to/file/logcore.pem
logstorage_address = logstorage.example.com
logstorage_port = 50002
cert = /path/to/file/novaservices.pem
key = /path/to/file/novaservices.key
cacert = /path/to/file/cacert.pem
freq = 100
size = 500
debug = True
secure_log_services = nova-scheduler,nova-api
```

Figure 5.7: Nova configuration file

contains the information necessary to interact with the LogCore. Details about the settings included in the two dictionaries are givens in Tables 5.3 and 5.4, while an example of Horizon configuration file is in Listing 5.8.

Options dictionary name		LOG_CLI_SSL_OPTS
Option	**Value**	**Description**
certfile	STRING	Specifies the client X.509 certificate file path.
keyfile	STRING	Specifies the path of the file containing the private key.
ca_cert	STRING	Specifies the CA X.509 certificate file path.

Table 5.3: LOG_CLI_SSL_OPTS dictionary settings

Options dictionary name		LOG_CORE
Option	**Value**	**Description**
host	STRING	Specifies the LogCore hostname. It could be in form of an IP address or a FQDN.
port	INTEGER	Specifies the port where the LogCore is listening.
timeout	INTEGER	Specifies the timeout for the requests.
api_version	STRING	Specifies the API version that has to be used. (not used in this version)

Table 5.4: LOG_CORE dictionary settings

```
# Logging Tab settings
LOG_CLI_SSL_OPTS = {
    "certfile": "/path/to/file/logconsole.pem",
    "keyfile": "/path/to/file/logconsole.key",
    "ca_certs": "/path/to/file/cacert.pem"
}
LOG_CORE = {
    "host": "logcore.example.com",
    "port": 50001,
    "timeout": 60,
    "api_version": "0.1",
}
```

Figure 5.8: Horizon configuration file

Chapter 6

Cheap BFT

Chapter Authors: Johannes Behl, Stefan Brenner (TUBS)
Tobias Distler, Andreas Ruprecht (FAU)

6.1 Introduction

The advent of cloud computing further amplified a trend that has been ongoing for roughly two decades now: More and more of our life, be it our private or our business life, happens in the net of all nets, happens in the Internet, happens online. By now, we use online services for virtually everything. We communicate via online services, we obtain news via online services, we make money via online services, we transfer money via online services, we even elect our government via online services. This development has brought a lot of benefits, but also at least one major drawback: By now, we completely depend on these online services. We depend on their availability, on their correct functioning, and in general on their trustworthiness.

However, with the increasing number of services, with their increasing complexity, and with their increasing importance, faults in hardware and software become more and more crucial. Preventing faults from getting in productive systems is one important approach in order to ensure that services do not fail. Nevertheless, fault prevention will in most cases never be completely effective in the sense that it could guarantee fault-free services. Therefore, measures have to be taken to keep services available even if their implementation is faulty, that is, measures for fault tolerance have to be applied.

One question that arises in that context is: What kind of faults are supposed to be tolerated, thus which class of faults is considered? One common answer is: faults leading to crashes. That is, a service shall remain available even if a part of the system crashes, if it just stops its execution. A technique to implement such a crash fault tolerance is service replication. Instead of hosting only a single instance of a service, multiple instances are used. When one instance crashes, the other instances can still provide the correct service.

Although tolerating crash faults is an important approach to implement high available services, considering the vast range of system malfunctioning, this very restricted fault model seems not sufficient for a lot of applications, particularly critical applications as regarded by the TClouds project. Random hardware errors can lead to corrupted messages and state, malicious attacks can compromise the system in various ways, and also ordinary bugs can cause a system to behave different than intended and not only to crash. In other words, there are innumerable kinds of faults that can entail a system to behave arbitrarily incorrect. If the availability of services has to be ensured also in this comprehensive fault model, services has to be implemented in such a way that they are able to tolerate arbitrary, also called Byzantine faults.

Byzantine fault tolerance (*BFT*) would be hence a highly desirable property for many service implementations. Despite that, BFT systems are not widely employed so far. One reason is their

high cost. Standard BFT systems require $3f + 1$ running service instances, called replicas, to tolerate only f arbitrary fault. Systems employing a trusted submodule in a hybrid fault model can lower these costs to $2f + 1$ actively running instances.

One objective of *CheapBFT* [KBC+12], a BFT system developed in the context of TClouds, was to reduce the costs further and thereby the obstacles preventing BFT from its wider adoption. CheapBFT makes use of a novel trusted FPGA-based hardware module, named *CASH*, and introduces passive replication in the field of BFT. As a result, CheapBFT provides Byzantine fault tolerance with only $f + 1$ replicas actively executing incoming requests as far as their are no detected errors. Only when an error occurs, additional f replicas, regularly provided with status updates so far, are activated. After the causing error has been removed, f replicas are put back in the passive mode, leaving again only $f + 1$ active replicas.

As a side effect, the different roles of replicas introduced with CheapBFT result in a unevenly distributed load across the participating machines. Active replicas have to execute all incoming requests while the passive ones just keep their state more or less up to date. However, cloud providers as well as cloud customers usually prefer evenly distributed load for several reasons. From the cloud providers perspective the hardware can be used more efficiently when load is distributed evenly across the available hardware. Cloud customers, on the other hand, want evenly distributed load because they usually pay for each machine hour according to the resources provided by these machines and not according to their actual utilization.

Therefore, we introduce *RotatingCheap* as an improvement of CheapBFT to tackle this problem. Briefly, RotatingCheap is another Byzantine fault-tolerant protocol based on the concepts of CheapBFT. Thus, it works with a reduced number of replicas by employing CASH as trusted submodule and with different roles assigned to replicas. Contrary to CheapBFT, however, RotatingCheap quickly rotates the roles among the replicas during the execution. As a consequence, replicas perform the active role for some requests and the passive one for others. This eventually leads to an evenly distributed load across all underlying machines as our evaluation shows.

6.2 Background and Related Work

In the following, we briefly explain the basic concepts behind Byzantine fault-tolerant (*BFT*) systems and describe and discuss more in detail the existing BFT protocols on which basis RotatingCheap was developed.

6.2.1 Basics of BFT Systems

A well-known and often applied fault model is the crash-stop model. This model assumes that systems only fail by crashing, that is, either they operate correctly or they operate not at all, in particular, they do not response to any request or give any other sign of life. Compared to this relatively simple fault model, the Byzantine fault model is more generic and broader. It assumes that faulty systems can behave in any arbitrary way. For instance, a system in this model can lose data, get into an invalid state or deliberately provide wrong and misleading messages to other system components.

In order to tolerate arbitrary incorrect behavior, a certain BFT system usually consists of multiple replicas, each running an instance of the considered application and executing all incoming requests according to a specific (agreement) protocol that determines a total order on the requests. Starting from the same state, running a deterministic application, and executing the same requests in the same order ensures that all replicas remain in an equal state and hence

return, if correct, the same result for each request. This enables the detection of faulty replicas by comparing the results as long as a particular quorum of replicas functions properly. Under the basic Byzantine fault model, it can be shown that this quorum has to comprise $2f + 1$ replicas out of $3f + 1$ if the system is supposed to tolerate up to f faulty replicas.

6.2.2 CheapBFT

This leads to a well-known problem of many existing BFT protocols: their high resource utilization stemming from the $3f + 1$ active replicas required to tolerate f faults. This reduces the practicability of these protocols and prevents them from being actually used. For that reason, a lot of research has been undertaken to lower the number of replicas involved in BFT systems and their employed protocols and thereby to lower the entailed resource footprint.

One representative of these BFT systems with reduced resource usage is *CheapBFT*, which has been developed in the context of TClouds. CheapBFT achieves a reduction of the required resources not only by employing a novel FPGA-based trusted submodule named *CASH*, but also by introducing a mixed active and passive replication scheme to the field of Byzantine fault tolerance. Using CheapBFT, only $f + 1$ replicas are required which actively execute incoming requests in normal case. For the rare cases where the system is subject to an error, f passive replicas are hold as warm stand-by.

Briefly, CheapBFT works as follows: The trusted submodule CASH is used to assign consecutive counters to all messages sent by participating replicas. The counters are unforgeably bound to single messages by signing them with certificates which are generated through CASH. Certificates comprise the ID of the particular CASH module, the counter unique to each message, and a message authentication code (MAC) which is calculated over the message content, the module ID, and the counter. When a replica receives a signed message, it is only allowed to handle the message if it has a valid MAC, which is verified by the receiver through its own CASH module, and if its assigned counter is the follower to the counter of the last message received from the sending replica.

In total, this prevents so-called equivocation which is a situation where one replica sends different messages to different other replicas at the same phase of the protocol and where the receiving replicas do not detect this misbehavior of the sending replica, or in other words, where a replica states different things although not permitted to do so. In CheapBFT, if a replica tried to make differing statements, this would be detected by at least one receiving replica since the validation of the message certificate would fail if it was altered by the sender or the receiver would witness a gap in the sequence of counters because each counter value is only allocated once by the CASH modules. In order to enforce a safe mapping of counter values to messages, creating a certificate must be the only way to change the counter value. Moreover, replicas have to save the most recent counter value received from each other replica to be able to reconstruct the correct sequence of values. If, and only if, a message to be verified contains the right and expected counter value the replica will increment its saved value for the particular sender.

Preventing equivocation allows a BFT system to reduce the number of required replicas from $3f + 1$ to $2f + 1$ and thereby to reduce the resource footprint. To further reduce this footprint, CheapBFT distinguishes between a normal mode, which is executed when no errors are present, and an error mode, which is activated when a replica is suspected to behave erroneous. Running in normal mode, CheapBFT employs a novel BFT protocol called *CheapTiny*, which only requires $f + 1$ active replicas executing requests. Additional f replicas are kept up-to-date by being provided with the state changes caused by the request execution. Therefore, using CheapTiny, f out of $2f + 1$ replicas run in a passive mode in which they only receive and

apply state changes, which in most cases requires significantly less resources than the active execution of requests. Whether a replica runs in active or in passive mode corresponds to its current role in the protocol. All in all, CheapTiny distinguishes three roles: Besides active and passive role, a replica can also perform as the so-called leader. The leader is an active replica which is responsible for distributing client requests and for driving the protocol, for instance, by proposing an order in which the requests have to be executed. During normal operation, there is exactly one replica which possesses the leader role.

Using $f + 1$ active replicas allows CheapTiny to detect up to f errors, however, this protocol is not able to mask them. Thus, in the presence of errors, CheapTiny can not make any progress. For that reason, if CheapTiny detects an error, a so-called transition protocol is initiated which is responsible for activating the f passive replicas and for reaching a consistent state on all replicas. After a consistent state has been reached, the MinBFT protocol is established for the further execution. MinBFT is a traditional BFT protocol with a hybrid fault model, that is, it employs $2f + 1$ active replicas which enables it not only to detect up to f errors but also to mask them. As in the case of CheapTiny, one replica is assigned with the leader role. To enable the transition back to CheapTiny, MinBFT is executed for a limited time. After that, the system will try to fall back to CheapTiny and thus to the normal mode with $f + 1$ active and f passive replicas.

6.2.3 Spinning

Spinning [GSVL09] is a protocol for Byzantine fault tolerance which enhances PBFT [CL99]. Like PBFT, Spinning requires $3f + 1$ replicas and like most BFT protocols, Spinning assigns to one of these replicas the leader role. Here, each phase during the execution of the protocol in which a particular replica continuously holds the leader role is called a *view*. The situation where the leader role is assigned to an other replica is called *view change*.

In contrast to PBFT, Spinning changes the leader periodically on a regular basis after a fixed number of requests and not only when it fails. Thus, it deliberately enforces periodic view changes. This is done to mitigate denial of service attacks and to balance system load more evenly across the involved machines. In order to prevent faulty replicas from becoming the leader over and over again, a blacklisting mechanism is used. If a leader is suspected to be faulty in view $v - 1$, for example because it does not introduce new requests to the agreement, it will be pushed to a blacklist of length f during the transition to view v. All replicas on this list will continue to participate at the protocol, albeit they will not be able to become the leader.

6.2.4 Comparison of CheapBFT and Spinning

CheapBFT introduces passive replication to Byzantine fault-tolerant systems and thereby demonstrates how the resource usage of such systems can be further lowered. However, the fixed assignment of roles to the replicas during normal operation entails an unbalanced distribution of the overall load between the replicas. Active replicas, and particularly the leader, have to exchange considerably more messages than passive replicas in order to come to an agreement about which requests have to be executed in which order. This also leads to a higher load for their CASH modules, which are used to certify and to verify messages. Additionally, the execution of requests consumes in most of the cases a way more CPU cycles than the application of status updates. All in all, CPU, network interface, and CASH module are significantly more utilized on active replicas than on passive ones.

Since also PBFT encounters load imbalances because a replica holds the leader role as long as no problems are encountered, Spinning initially tackles balancing of system load across

replicas by rotating the leader role at predetermined points in the protocol. Spinning requires, however, $3f + 1$ active replicas to tolerate f faults. Compared to CheapBFT, this not only means more active replicas utilizing resources but also a higher resource usage for each single replica since replicas have to communicate with all other actively participating replicas during protocol execution. Following to Spinning, the same authors presented EBAWA [SVCBL10], which also reduces the number of replicas to $2f + 1$ by employing a trusted submodule, but unlike CheapBFT, all $2f + 1$ replicas participate in the protocol actively, even in the absence of errors.

6.3 RotatingCheap

In this section, we present RotatingCheap, a BFT protocol which is based on CheapBFT and its novel CheapTiny protocol but which achieves a better load distribution by constantly rotating the roles of all participating replicas.

6.3.1 Initialization

To initialize the system, unique IDs in the range from 0 to $2f$ are assigned to each of the $2f + 1$ replicas. These IDs also determine the initial roles of the replicas (as in CheapTiny, three roles are distinguished: leading active replica, active but following replicas, and passive replicas):

- ID 0 will be active and leader
- ID $1 - f$ will be active replicas
- The remaining f replicas will be passive

6.3.2 Communication

Clients can introduce messages into the system by sending a $\langle \text{REQUEST}, m \rangle$ message, signed by their key, to an arbitrary, for instance geographical close replica. Here, m contains the command to be executed, the ID of the client, as well as a client-specific sequence number which is used by replicas to recognize already executed request, that is, to filter duplicates. After sending a request, a client waits until equal responses from $f + 1$ replicas have been received, before sending another request.

Agreement Phase

When a replica receives a message from a client, it verifies its authenticity first. If the request is valid and the receiving replica is leader of the current round, it sends a $\langle \text{PREPARE}, m, v, mc_L \rangle$ message to all active replicas. The PREPARE message contains the message m from the client, the current round number v, and the certificate mc_L from the CASH submodule which in turn contains the current counter value. In case the replica is not the leader when the request is received, the message is buffered and scheduled for agreement when the replica will become leader the next time.

When an active replica receives a PREPARE message, it checks the validity of the certificate by means of its CASH submodule and whether the certificate contains the next expected number. This ensure that messages are only accepted if there are no gaps in message order. After a successful validity check, the active replica sends a $\langle \text{COMMIT}, m, v, mc_L, mc_p \rangle$ message to all other active replicas. m, v, mc_L are taken from the PREPARE message. mc_P is created by signing the concatenation of m and mc_L via the local CASH submodule.

A replica receiving a COMMIT message again checks the correctness of the certificate and continuity of the message order. If an active replica got $f + 1$ valid commit messages subjecting a request m (one from each active replica) a round is considered agreed and execution of this request is started.

Execution Phase

During the execution of a request, the application creates a response r for the client and a representation of the state changes that were caused by request execution. This state change representation is denoted by u. After the execution, an active replica creates an $\langle \text{UPDATE}, r, v, u, C \rangle$ message that contains the response r, the round number v, the state change u, the set of COMMIT messages C as well as the PREPARE message from the leader and sends it to all passive replicas. Finally, the response r is sent to the client and the transition to the next agreement round is initiated. The policy for the selection of the role of the next agreement round is described in the next section.

If a passive replica receives an UPDATE message, it checks its signature first. As soon as $f + 1$ correct UPDATE messages are available from all active replicas, the contained state change is applied to the current state and the next agreement round is started. In order to ensure the correct ordering of messages from the replicas, it is checked whether the sequence numbers of the PREPARE and COMMIT messages in C contain no gaps and match the expected value.

This is necessary because in an agreement round, messages are exchanged between active replicas that are not received by the passive replicas. Sending PREPARE and COMMIT messages cause an increment of the according counter from the active replicas. Without special treatment, in the next round, a former passive replica would notice a gap in counter values when it receives a message from a replica that has been active in the last as well as the current round. Because of the fact that a correct UPDATE message must contain the $f+1$ COMMIT messages of all active replicas including the current counter values, passive replicas are able to update the counter values for the other replicas, thereby prevent the occurrence of such gaps.

Transition to the Next Agreement Round

Switching the role of a replica during the transition from agreement round k to $k + 1$ is done as follows: Assuming the replicas arranged in a circle ordered by their ID, roles rotate on this circle clockwise by f positions after an agreement round is considered finished.

Active replicas including the leader consider an agreement round finished, as soon as they have sent the UPDATE message and the client response. Passive replicas consider a round finished when they have received at least $f + 1$ corresponding UPDATE messages and applied the respective state changes.

Formally, let L_k be the ID of the leader, A_k the ID set of active and P_k the ID set of passive replicas in round k, f the amount of tolerable failures and $n = 2f + 1$ the total number of replicas, then:

- $L_{k+1} = L_k + f (mod\ n)$
- $A_{k+1} = \{a + f (mod\ n) \mid a\ \epsilon\ A_k$
- $P_{k+1} = \{p + f (mod\ n) \mid p\ \epsilon\ P_k$

This is also illustrated in Figure 6.1 for three successive agreement rounds for $f = 1$. Figure 6.2 shows the role change policy for $f = 2$.

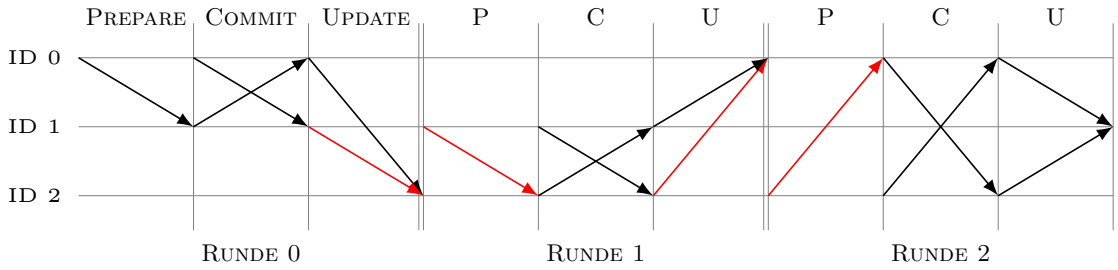


Figure 6.1: Three agreement rounds in RotatingCheap for $f = 1$; roles in round 3 eventually are equal to round 0. Communication with clients is not illustrated for simplicity. Red messages can be batched for optimization purposes.

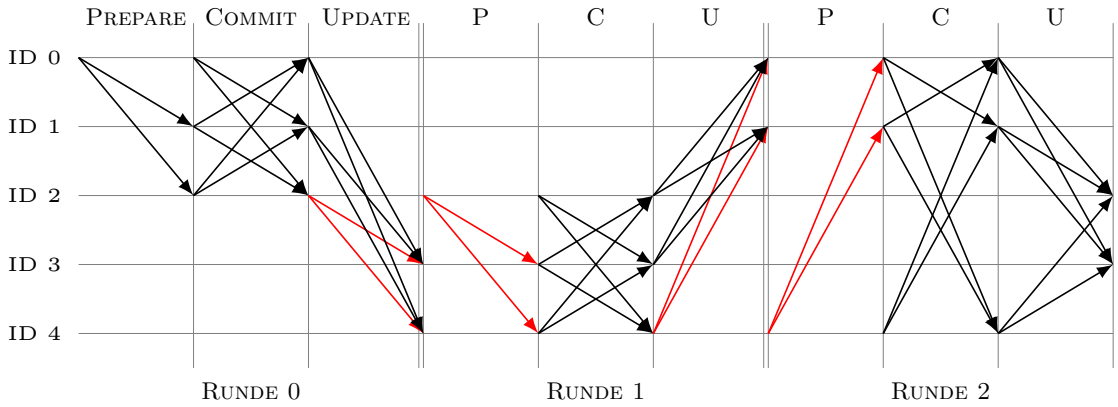


Figure 6.2: Three agreement rounds of RotatingCheap for $f = 2$. Communication with clients is not illustrated. Red arrows illustrate messages that could be batched for optimization.

Checkpoints

In order to be able to check whether an agreement round was already finished in case of a failure, all replicas save all messages they sent as well as all received UPDATE messages from other replicas. Since this approach is not scalable for long system uptimes, checkpoints are periodically created after a certain number of rounds.

A $\langle \text{CHECKPOINT}, v, snap, mp_{cp} \rangle$ message contains the most recent (finished) round number v, the hash value of the replicas state $snap$ and is signed with the certificate mc_{cp} from the CASH submodule. Then, a replica sends the CHECKPOINT to all active and passive replicas.

When a replica receives a CHECKPOINT message, it verifies the certificate and checks the counter value. As soon as a replica received $f + 1$ correct matching CHECKPOINT messages, the checkpoint is considered stable and the replica can remove old PREPARE, COMMIT, and UPDATE messages from its log.

Skipping Rounds

In case a replica gets leader which has not received any request from clients during the time it was in active and passive state, a timer with the interval t is started while waiting for requests. If the replica receives a request during this interval, the timer is stopped and normal agreement is started as described above. However, if the timer times out, the replica sends a SKIP message to all active replicas. This message is agreed on like any other request.

6.4 Evaluation

In this section we evaluate RotatingCheap with respect to the formulated objective to reach a better distribution of load among the replicas. We only consider the case $f = 1$, that is, we use $2f + 1 = 3$ replicas. The replica machines are commodity 4-core machines (2.3 GHz, 8 GB RAM) equipped with an FPGA that realizes the CASH submodule. In addition, there are three client machines based on 12 cores (2.4 GHz, 24 GB RAM) each, on which the clients are equally distributed. All machines are interconnected with switched Gigabit Ethernet. As a convention for this section, a load value of 1.0 represents 100% load on one core, hence, 2.0 represents 100% load on two cores and so forth.

In order to be able to compare RotatingCheap to CheapBFT, the original implementation of CheapBFT has been slightly modified. Instead of multiple agreement rounds in parallel, which CheapBFT is able to do, our modified version only supports one agreement round at a time. This modified version is called *BlockingCheap* from now on. That means, like it is done in RotatingCheap, before the leader starts round $n + 1$ it has to wait until the UPDATE messages for round n have been sent. For both protocols, we define a maximum batching size of 20 requests, that is, up to 20 requests can be ordered within a single agreement round.

Clients send all requests to exactly one replica of their choice and wait until they receive a response. They send the next request only after a successful validation of the response. For RotatingCheap, replicas are selected by clients such that every replica receives approximately the same amount of requests. In contrast, for BlockingCheap, the clients will send all requests to the replica with ID 0.

For this evaluation, a micro benchmark is used which allows to set the size of requests, responses, and the state changes. Therefore, besides measuring throughput for empty messages, we are able to simulate read (4 kB sized responses) and write access (4 kB sized requests and updates).

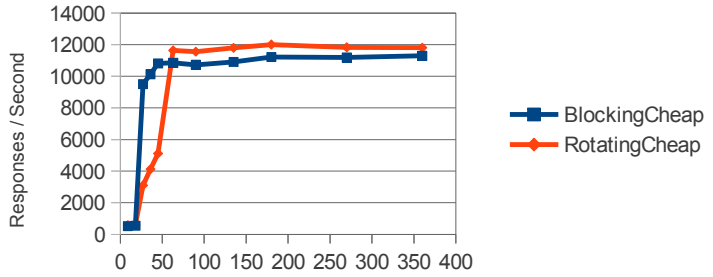
At the beginning of each experiment, we allow the system to warm up for 90 seconds, which prevents artifacts caused by Just-in-time-compilation from showing up in the results. After the warm up phase, we measure throughput on the client and server side as well as the network utilization for 60 seconds.

6.4.1 Throughput

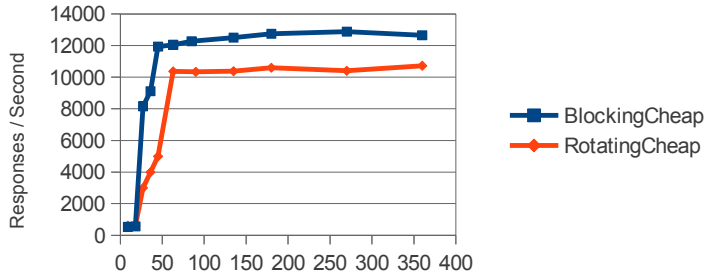
First, the possible throughput of the system is measured, that is, the maximum amount of responses to client requests per second. For this purpose, the amount of client instances is increased step by step from 0 to 360 clients.

In Figure 6.3, the results are illustrated for both, BlockingCheap and RotatingCheap. In the experiment with empty messages (Figure 6.3(a)), RotatingCheap is slightly ahead of BlockingCheap, which is caused by a better distribution of requests to the replicas. However, when it comes to read access (Figure 6.3(b)), BlockingCheap achieves about 15-20% better throughput. This is caused by delays due to the hashing of response messages for which RotatingCheap is more susceptible than BlockingCheap. The evaluation of write access (Figure 6.3(c)) shows that both protocols achieve approximately the same throughput. Note, however, that this is only about one half of the read throughput.

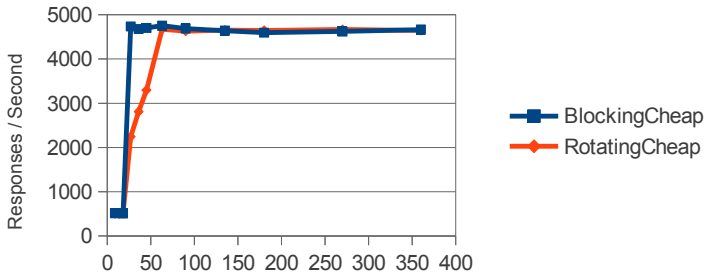
All experiments show that throughput rapidly increases with the number of clients until a certain threshold is reached from which on the throughput remains static. This can be explained by further measurements: Because of the modular architecture of the implemented prototype, a message has to pass many layers which interact asynchronously using queues. Thus, every layer



(a) minimum throughput for 0 kB messages



(b) read access throughput for 4 kB responses



(c) write throughput for 4 kB requests and update messages

Figure 6.3: Comparison of possible throughput using RotatingCheap and BlockingCheap for variable amount of client instances.

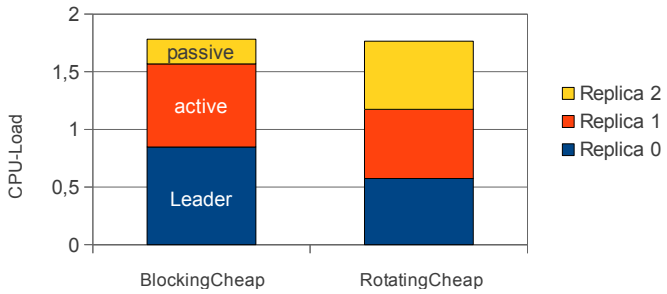
introduces its own delays. In the case of empty messages, the time between receiving a request and until the UPDATE messages are transferred finally, is about $1.7ms$. Because of the selected batch size of 20 requests, the maximum amount of requests per second is $\frac{20req}{1.7ms} \approx 11.764$. This load value is already reached when 20 or more clients simultaneously interact with the system.

This also explains why the throughput of RotatingCheap increases slower than for BlockingCheap below 60 clients. Since all requests are evenly distributed across all replicas, a batch in RotatingCheap can only be filled when any of the three replicas accept 20 or more clients. For BlockingCheap batches are already filled with only 20 clients.

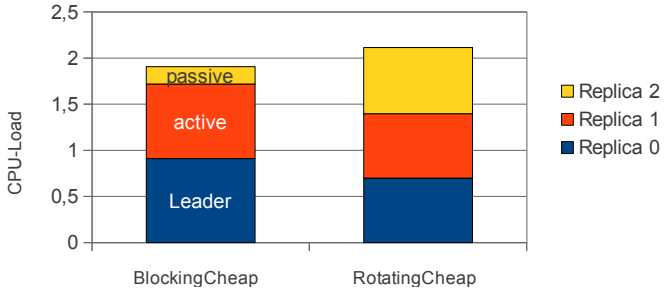
6.4.2 CPU Load

In this section the CPU load of the replicas is evaluated in order to see the difference in load between active and passive replicas as well as the leader replica.

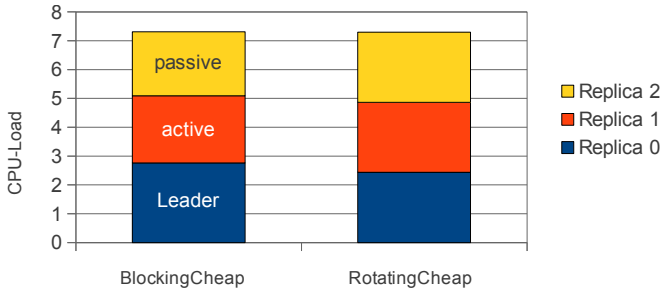
For BlockingCheap, we expect higher load on active replicas compared to passive ones. Furthermore, the leader replica is expected to be higher loaded than the non-leader replicas. In contrast, in RotatingCheap all replicas are expected to be evenly loaded.



(a) CPU load for 0 kB messages and 180 clients



(b) CPU load for 4 kB responses and 180 clients



(c) CPU load for 4 kB requests and update messages with 180 clients

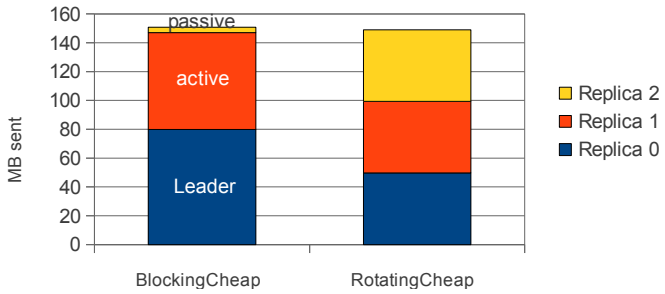
Figure 6.4: Comparison of CPU load of RotatingCheap and BlockingCheap using 180 connected clients. The values are averaged across 60 seconds.

Figure 6.4 shows the results for all three above mentioned test cases for 180 connected clients each. The results are average values which are normalized to 10.000 requests per second.

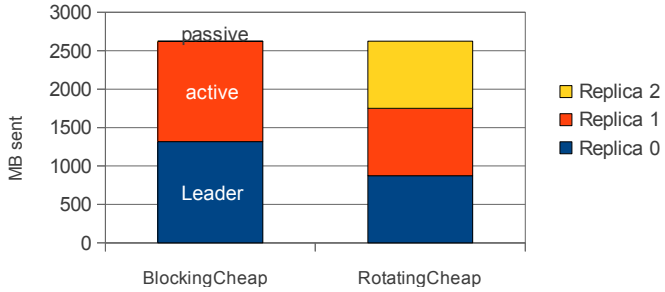
In Figure 6.4(a), the experiment with empty messages is illustrated. The leader replica reaches about 0.85 load while the other active replica only reaches about 0.72 load and the passive replicas show a very little load of approximately 0.21. In contrast, using RotatingCheap load is distributed evenly; all replicas show a load of about 0.57 here. This is what we expect, since all replicas are "simultaneously" leader, active, and passive.

We measure a similar behavior when it comes to 4 kB response messages, as can be seen in Figure 6.4(b). The load on the leader replica is about 0.91 while active non-leader replicas show about 0.8 load. The passive replica, however, shows a load of 0.21 only. In contrast, the load on each replica when using RotatingCheap is about 0.59. Compared to Figure 6.4(a) the higher load on active replicas here is due to the fact that larger messages are used which causes the hashing to be more time consuming. In total, using RotatingCheap the load is a little bit higher, which is caused by different handling of messages that have to be hashed.

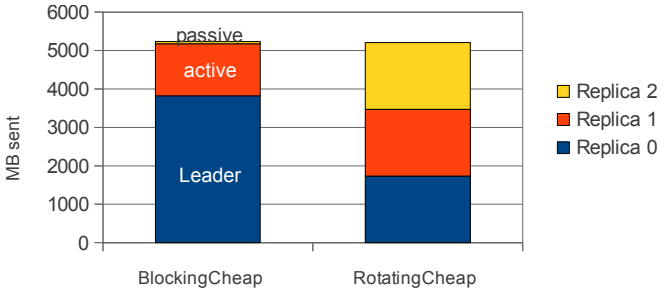
Finally, Figure 6.4(c) shows the results for write access. Here, the load even for BlockingCheap is already distributed evenly across the replicas. While the leader (2.76) is slightly



(a) Data volume sent, 0KB messages, 180 Clients



(b) Data volume sent, 4KB responses, 180 Clients



(c) Data volume sent, 4KB requests/updates, 180 Clients

Figure 6.5: Comparison of data volume sent over the network by the replicas for a range of 60 seconds using 180 clients. Values are normalized to a throughput of 10.000 requests per second.

higher loaded than the active replica (2.33), also the passive replica shows surprisingly high load (2.22). This is due to the fact that 4 kB sized update messages are also hashed on the passive replicas for the purpose of verification. However, still using RotatingCheap, the slight differences in load are flattened by the protocol and all replicas reach approximately the same load of about 2.43. Compared to the other experiments, total load is much higher here because of the additional effort for hashing large protocol messages.

6.4.3 Network Load

As shown in the following, the inhomogeneous distribution of load is even more noticeable for network load. Here, for BlockingCheap, active replicas are much higher loaded because active replicas have to send PREPARE, COMMIT, and UPDATE messages for every agreement round. In contrast, passive replicas mostly receive UPDATE messages only and send CHECKPOINT messages. In RotatingCheap, we expect the inhomogenity to be flattened again.

Figure 6.5 shows the data volume sent from each replica for all three test cases as described above for a time period of 60 seconds. The values are normalized to 10.000 responses per second.

For empty messages, as it is shown in Figure 6.5(a), we measured the expected behavior.

While using BlockingCheap, the active replicas send 53% (leader) and 45% respectively, only 2% of the data volume are sent by the passive replicas. In RotatingCheap, the load again is distributed quite evenly across the replicas. The total volume of data sent is almost identical here because for each agreement round the same amount of protocol messages has to be sent.

When we simulate read access, as shown in Figure 6.5(b), the differences become even more significant because of the large response messages. While active replicas send about 50% of the total amount of data volume, passive ones have a negligible share. RotatingCheap again provides evenly distributed load across all replicas.

An even more significant contrast is illustrated in Figure 6.5(c) which shows the results for write access. Here, the leader in BlockingCheap by far causes the highest portion of data volume to be sent, while the other active replica only reaches 26% and the passive replica is almost negligible. RotatingCheap here shows evenly distributed data volume sent by each replica. The huge difference in load between active replicas in BlockingCheap is caused by the fact that the leader adds the client request to the prepare message which, as a result, gains about 4 kB size.

6.4.4 Result

In summary, the goal of evenly distributed load comparing all replicas with each other apparently can be reached by rotating the roles of replicas using RotatingCheap. This is not only true for CPU load, but also for network data volume sent by the replicas. As the evaluation has shown when using BlockingCheap, load will concentrate on the active replicas while the load is evenly distributed when using RotatingCheap in all three scenarios (empty messages, read access, write access). While throughput for empty messages is even higher using RotatingCheap compared to BlockingCheap, however, for write access both protocols reach approximately the same throughput and for read access we measured about 15-20% penalty.

6.5 Conclusion

In this work, we introduced a new Byzantine fault-tolerant protocol named *RotatingCheap* which enhances CheapBFT by allowing to achieve an even load distribution among participating replicas of service implementations.

RotatingCheap employs the concepts developed for and implemented in CheapBFT. It uses the trusted hardware-based submodule CASH and the mixed active and passive replication scheme to reduce the number of required active replicas to $f + 1$ in order to tolerate up to f faults. In addition to CheapBFT, it uses a dynamic assignment of replica roles to balance load differences induced by the different work necessary to perform those roles. The assignment which replicas serve as active and which as passive ones is changed after every protocol round. This mechanism eventually distributes the overall load among all replicas as the evaluation of a prototype implementation clearly shows.

Chapter 7

Tailored VMs: Key / Value Store

Chapter Authors: Klaus Stengel (TUBS)

7.1 Introduction

The rising popularity of cloud computing due to better cost effectiveness and scalability results in an increased demand to put also security critical infrastructure on cloud platforms. Unfortunately, current cloud infrastructure is still often perceived to be less secure than dedicated and locally managed hardware. This hinders companies to move large-scale critical applications such as trading, healthcare information systems and smart-lighting to the cloud.

Apart from the security aspects it is also crucial to leverage the pay-per-use model for computing power and disk storage to operate the application in a cost-effective manner. Particularly when implementing such applications on top of Infrastructure-as-a-Service (IaaS) clouds, this means that only a minimal amount of computing power and disk storage should be required. This leads to designs with many instances of simple, but highly specialized software components that can be added or removed dynamically depending on current demand. Application agnostic examples for this kind of services are in-memory caches like Memcached [Fit04] or Amazon's Dynamo file system [DHJ+07]. Interestingly, this trend away from monolithic systems towards solutions with collaborating services mostly happens on the application and middleware tier but not so much on the operating system level. Specialized services are still being developed around the same commodity operating systems (i.e., Linux and Windows). This is a real problem in terms of effectiveness, as these systems waste resources for features that are not required for the specific use case at hand. Regarding the aforementioned security and reliability aspects, the *Trusted Computing Base (TCB)* of such commodity systems is also unnecessary large, resulting in a considerable amount of vulnerabilities. Apart from the usual stack and buffer overflows, these often stem from unforeseen interactions with less commonly used features. Reducing a commodity operating system kernel to a bare minimum in terms of features also has its limitations, as was demonstrated recently with Linux: A minimal configuration to run a predetermined service on a specific hardware configuration resulted in about 80% reduced code size, while only half of the known security issues from the Common Vulnerabilities and Exposures database (CVE) [1] could be eliminated [KTD+13].

Specialized library operating systems have the potential to vastly improve the situation, which was shown in previous work like the the *Mirage Unikernel* [MMS+13] and *Drawbridge* [PBWH+11]. Such systems posses a small code base and a low resource footprint from the beginning and are ideal for running inside virtual machines as provided by IaaS clouds.

[1] http://cve.mitre.org

In combination with an implementation language with a better type system, many of the security issues found in current commodity systems are eliminated by design and result in a resource efficient and secure solution.

In our Tailored Memcached subsystem, we therefore follow a clean slate approach by evaluating the design and implementation space of using Haskell as a pure functional programming language. Tailored Memcached explores how specialized runtime environments with fine-grained tailoring and the benefits of a programming language that facilitates verification work together to implement secure cloud services.

In the remainder of the chapter, we first provide a short overview of related work in the area of operating system development in safe programming languages, library operating systems and software tailoring. After that, we revisit security and reliability issues in current application stacks that led to the idea of library operating systems for the cloud in Section 7.4. This also motivates the use of Haskell, a pure functional programming language for our Tailored Memcached. Next, we focus on the key aspect of our work, which comprises software tailoring in Section 7.5 and possible strategies to verify the correctness of certain implementation features in Section 7.6. The discussion of these subjects provides the basis for the description of the initial architecture of Tailored Memcached, which can be found in Section 7.7. Finally, Section 7.8 draws conclusions and outlines the road ahead.

7.2 Related Work

As Tailored Memcached touches a large number of different research subjects, we want to begin with a short overview of each of these areas. First, we look into more popular, safe programming languages as an alternative to implement services for a virtualized environment. Second, we detail different approaches in the area of aspect-oriented programming to implement distinct features of an application in an independent and reconfigurable fashion. The final part of this section is concerned about the use of functional programming languages in operating systems.

7.2.1 Programming languages

Looking at the area of fully type-safe systems programming languages that can be used on bare hardware, contemporary research strongly focuses on popular imperative, object-oriented languages like Java and C#. A major disadvantage from a security perspective is, that these languages were designed with a machine-independent bytecode representation in mind, thus depending on a complex Just-In-Time compiler (JIT) as an integral part of the runtime system to execute. Albeit some projects, like *Singularity* with its *Bartok* compiler [HL07] and *Maxine* [WHVDV+13], also implement the JIT in a safe programming language instead of C/C++, they still significantly increase the code size and possible attack surface. Dynamically generated machine code also precludes some of the common security measures to mitigate vulnerabilities like Data Execution Prevention (DEP). On top of that, the process of generating the platform-specific machine code is also prone to bugs that might be exploited by determined attackers [RI11]. There are also approaches to statically translate Java programs into machine code, but these either target user-level applications (e.g. *GCJ* [Fou13b] and *Excelsior JET* [LLC13]), or are solutions specifically aiming at small real-time and embedded systems (e.g., *KESO* [SSWSP10]) with different feature sets.

7.2.2 Aspects

Another topic of interest is how the fine-grained tailoring of the system can be accomplished. For imperative programming languages exists a large body of research concerning aspect-oriented methods to create highly modular and reconfigurable programs. Popular practical examples are the *AspectJ* [Fou13a] and *AspectC++* [SGSP02] compilers and the *CiAO* [LSHSP12] operating system as a working demonstration of the application of these methods. In contrast to this, the functional programming community seems to pay surprisingly little attention to the developments in this area and how these techniques are applicable to these languages. Nevertheless, some notable results for the Haskell language exist in the form of the *UUAGC* compiler [SAS+99], which uses aspects to describe transformations on the internal program representations in a modular fashion. The more general and theoretical paper on *EffectiveAdvice* [OSC10] contains an alternative approach to UUAGC and can guarantee different levels of isolation between program code weaved together by aspects.

7.2.3 Operating Systems

There is also existing work regarding operating system implementation in Haskell or other related functional programming languages. Probably the first instances of such systems were *hOp* [CB04] and the closely related *House* [HJLT05], which concentrate on a monadic framework for low-level hardware interaction and provide traditional user-/kernel-space separation with a defined set of system calls.

A similar approach was used for prototyping the *seL4* Microkernel [KEH+09], which is implemented in a simplified variant of Haskell. It can be used both as input to a theorem prover for verification, as well as executed directly for simulation purposes. However, this prototype was not supposed to serve as the final result, as the necessary runtime support and performance characteristics were deemed incompatible with the goal of providing a high speed and fully verified Microkernel. The performance-optimized realization of the seL4 kernel for actual use is a complete rewrite in the C programming language, which has to be verified again whether it is equivalent to the Haskell model.

Another approach from industry, Galois' Haskell Lightweight Virtual Machine (HaLVM) [Gal13a], is designed to interface with the paravirtualized environment of the Xen Hypervisor [BDF+03] instead of managing physical hardware. It belongs to the family of library operating systems as it does not differentiate between user- and kernel-space and the kernel is linked immediately to the application and required libraries at compile-time. As this technical realization is very much in line with the service architecture we propose in Section 7.7, we based our own prototype on HaLVM. Unfortunately, the basic HaLVM platform is not flexible enough, in the sense that it only supports Xen as the Hypervisor platform and not any other often encountered on IaaS clouds. As the major design goal is of Tailored Memcached is the ability to tailor the software stack to a given application, meaning that additional mechanisms are required to keep the code base small. The Mirage approach [MMS+13] also provides a relatively small and efficient operating systems library based on *O'Caml*, but is not particularly oriented towards reconfigurability and correctness. With Tailored Memcached we try to improve upon these results both in terms of security and flexibility regarding the service configuration.

7.3 Current Software Stacks

Developing a service with today's standard tools typically results in a large software stack when deploying the result on an IaaS cloud. While the actual logic encoded in the service is usually small and can be measured in ten thousands lines of code, the supporting software stack is multiple orders of magnitudes larger. For example, using the popular Java 7 platform to implement ones service entails a dependency on a complex runtime system with a huge standard library, which contains many features the application does not need. The runtime system also depends on a commodity operating system that has been growing in features and size for at least 15 years. Moreover, major parts of the entire platform are written in C and C++, which are languages that contain lots of known pitfalls, especially regarding memory management. Mistakes in that area often allow attackers to crash or even remotely execute code in networked services.

For a preliminary evaluation of our prototype implementation described in Section 7.7.3, we determined the TCB of the Hotspot Java Virtual Machine of OpenJDK 7 and the Linux 3.2 kernel. The amount of code that has to be trusted is already in the order of millions lines of code and does not even include any library functions and administrative tools also necessary to boot the platform.

7.4 Security and Reliability

The particular use case we have in mind for such tailored systems, security sensitive cloud services, has very diverse characteristics what security actually means. This can range from Memcached[Fit04], which typically does not enforce any sort of access control at all, up to services which allow fine-grained control over who may access which record. As a basic rule, exposed services should not contain any issues that might allow unauthorized parties to trigger actions that were not intended. While it is conventional wisdom to verify every piece of incoming data before actually processing it, we still see plenty violations of this rule in practice. Modern system programming languages, like Java or C#, already enforce memory safety and thus prevent common buffer overflow issues. There is, however, no special support for expressing additional constraints that must be taken into account when handling data. Necessary data conversions for middleware layers, like escaping to prevent SQL injections or cross-site-scripting attacks, are not enforced by any platform on a language level. An elegant solution for many of the problems described above lies in the type system of the programming language.

7.4.1 Type Safety

The Curry-Howard-Isomorphism [SU06] provides the theoretical foundation how the type system of a language can be leveraged to statically guarantee certain properties of a computer program. In practice, the need to provide type information is often seen as a nuisance instead of being used for this purpose. The reason for this is that the type systems of the current widespread statically typed languages are geared towards memory safety and enabling compiler optimizations instead of the concerns of the application. However, extending the expressive power of a type system often comes at the cost that type checking and inference is no longer a mechanical task that can be done by the compiler without further assistance. There are ongoing efforts like ATS [CX05] and Idris [Bra11] to remedy the situation by designing languages that try to find a useful compromise between an entire theorem proving framework and traditional type systems. These are often

based on a functional language at the core with some form of *dependent types*, as these are closer related to mathematical models than the more widespread imperative languages. Such languages are still in an experimental stage, so we have chosen Haskell as a programming language because it provides a mature basis with many extensions to the type system. These help us to encode properties in the type system in a similar fashion to dependent types, as demonstrated in [McB02].

As Haskell conveniently captures functions with side effects in a purely functional model using Monads, it becomes much easier to reason about the behavior of code fragments. Unfortunately, the full Haskell programming language is unsound, so following this approach still has certain loopholes. From a theoretical standpoint, the primary issue is that there are no proper limitations on recursive functions. The type system does not enforce functions to return eventually by demanding proof that a termination condition is present and guaranteed to become true at some point. This is most visible by the fact that the keyword `undefined` is a valid value for any type, similarly to `null` in other languages. However, such constructs are not used that often in typical Haskell programs, as variables are principally immutable and have to be assigned with their final value right away.

Another issue regarding the reasoning about Haskell code is that it allows access to functions written in C and potentially unsafe memory operations that may disrupt unrelated parts of the program. A relatively new development to remedy this issue is the introduction of the "Safe" language extension in the *Glasgow Haskell Compiler (GHC)*, the currently most popular Haskell compiler. Actual application code can be restricted to call only safe modules and functions that do not perform such potentially dangerous operations. This does not solve the problem entirely as the language still has to interact with lower levels at some point. As such functions must be explicitly trusted by placing additional keywords, it helps to make these more discoverable for manual auditing.

Despite these limitations, we believe that writing software in Haskell and properly using the type system will still produce much safer and reliable results. In Section 7.6 we will also outline additional ways to prove correctness of certain aspects of the program. We also intend to combine these with our approach to software tailoring in order to reduce the overall software stack to the smallest possible size.

7.5 Tailoring

The goal of the tailoring process is to keep as little code as possible in the final program. Apart from improved security, we also expect benefits in performance and reliability, as we eliminate code paths that are not required for the use case at hand, but may contain faults that might be triggered in certain circumstances. Adapting the entire software stack for a particular use case and IaaS cloud platform provides plenty opportunity for specialization. In the following sections we will give a short overview of subcomponents that impact the feature set of Tailored Memcached.

7.5.1 Cloud provider environments

As a service should be able to run on as many IaaS cloud platforms as possible, we have to support the virtual machine environments they provide. These often differ based on the virtualization solution they employ. Fortunately, there is only a limited set of products that are available and also seen on actual clouds, namely QEMU (in its KVM and HVM-Xen variant), HyperV,

VMware, paravirtualized Xen and VirtualBox. While each of these, except for the paravirtualized Xen, provides an environment that behaves very much like a standard x86 machine, they still need special drivers for accessing virtual network and storage devices. If we know what virtualization layer the IaaS cloud uses, we do not need any infrastructure to handle detection at runtime and can just link against the proper variant when compiling the tailored instance. A paravirtualized Xen environment is also quite different compared to the standard x86 model, especially in the area of memory management. Fully virtualized environments provides the same interface as the corresponding hardware unit for setting up memory mappings, while the actual memory allocation on the physical machine is managed by the virtual machine monitor and invisible to the guest system. With paravirtualized Xen, these mappings are still visible to the guest and it has to request changes by going through a special Hypercall-API instead of being able to directly reconfigure the memory management unit. In order to also support this platform it is not sufficient to have configurable network and storage drivers, but also to keep the memory management subsystem replaceable.

7.5.2 Application requirements

Apart from differences in the platform from the cloud provider, we have to anticipate the needs of the hosted service application. As we consider mainly simple, cooperating network services we need to keep the system much more configurable than what, for example, a standard Linux can provide. For simple services like Memcached [Fit04] it is not even necessary to have support for persistent storage or a filesystem. Regarding the network stack, many services require either TCP or UDP, but it is very seldom that both protocol are needed. Support for auxiliary internet protocols, like DNS name resolution is also a function often not required in internal backend services. The implementation of each of these features typically requires in the order of ten thousands lines of code each, which can not be disabled individually on commodity operating systems.

7.5.3 Implementation strategies

Such fine-grained variability needs systematic organization to stay manageable. On the implementation side we have to find a strategy to write maintainable code in the Haskell programming language that is extremely modular. On a higher level we need a feature model for the application developer, so that he can easily select which features are actually needed for his use case to compile a kernel image. As there are certain dependencies between the individual features, like UDP requiring IP support, the model needs to be able to encode such restrictions in a flexible way and ensure that the final configuration is valid. Fortunately, the required tool support is readily available from free as well as commercial offerings, so we will not go into further details regarding this aspect. One recent example for such a tool is the *FeatureIDE* framework [KTS+09].

This leaves us with the question how to keep the actual program code reconfigurable. On a module level it is very simple to just remove the files implementing the corresponding module from the body of compiled code. For changes within files, most Haskell compilers also offer to use the preprocessor of the C language to add macro support and conditional inclusion of code. While this may be acceptable for a low number of variation points, further testing has shown [LST+06, TSD+12] that it has serious shortcomings regarding scalability in more complex scenarios. The C preprocessor also only works on the text representation of the source code and does not allow any interaction with the actual structure of the program. This can lead

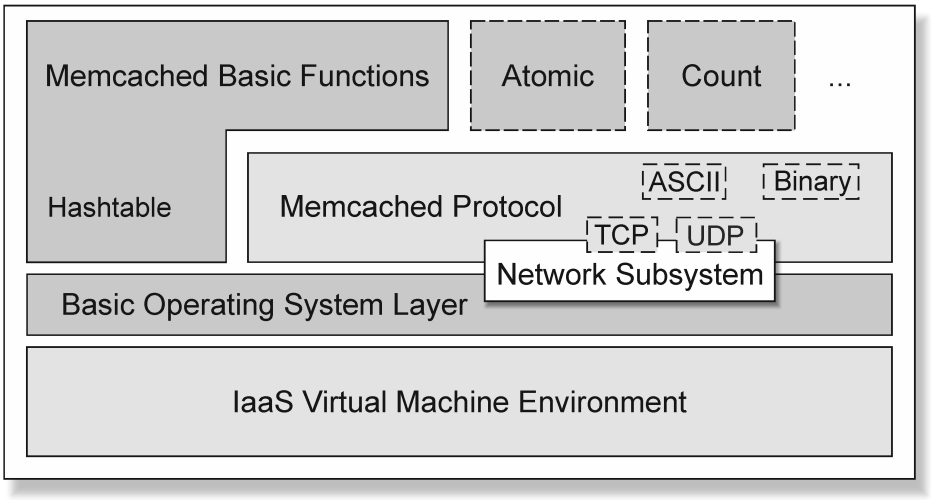


Figure 7.1: Software layers of HsMemcached prototype architecture

to program code in certain configurations that is still syntactically correct, but does no longer make any sense in the application context. In our use case of critical network services, such misbehaving program parts may create information leaks and other security relevant issues.

This is why we actually need better guarantees, which can only be obtained by tackling the reconfiguration problem on the language level and leveraging the type system. Therefore we take some of the general technique described in EffectiveAdvice [OSC10] in Tailored Memcached as a basis to explicitly describe whether features may interact with each other and also restrict the ways how they can do so. Using this approach, program features are expressed as functions that can be combined using special operators to form higher-level operations. We want to extend this notion to enforce certain interaction patterns through phantom types that exist only during compile time. This allows us to express restrictions like, for example, data that may only be passed over a network connection if it belongs to the corresponding authenticated user. If one tries to pass data without prior authentication, the types will not be compatible and thus prevent the compilation of a defective program. When we have such collection of features with additional guarantees expressed in the Haskell types, we gain further confidence in the correctness of the program.

7.6 Towards Software Verification

Building a completely verified system from the application via the compiler through to the hardware level is not feasible today, especially with the complexity involved in a cloud platform. Instead, our approach allows easier verification of certain aspects of the service. This is an interesting property to have, as the security requirements of a service are usually limited to certain aspects of the whole system. As an example, it may be considered a much more serious problem if one person can access another persons' tax data than if just the same person's tax estimates are calculated wrong. While both cases are correctness issues, only the first is actually relevant for security. Due to the isolation properties we gain from exploiting the Haskell type system as explained in Section 7.5.3, we can reduce the amount of code that has to be verified to these critical aspects.

	Hotspot Java VM	Linux	HaLVM + Libraries	Memcached	Haskell Memcached	Linux-based Memcached	Tailored Memcached on HaLVM
Haskell	0	0	83,279	0	2,449	0	85,728
Java	100,246	0	0	0	0	0	0
C/C++/C--	444,603	859,901	69,028	8,947	0	868,848	69,028
Assembler	858	17,172	1,691	0	0	17,172	1,691
Total	545,707	877,073	153,998	8,947	2,449	886,020	156,447

Table 7.1: Comparison of Source Lines of Code (SLoC) of different software components

7.7 System Architecture

In this section we give an overview about the basic system architecture we aim for in Tailored Memcached.

7.7.1 Overview

As described in Section 7.6, we take advantage of Haskell's type system as much as possible. In order to do so, we need also to have as much code of the platform written in Haskell, without any large operating system between the language runtime and the VM environment. Building blocks that are typically running in a kernel context, like network protocols and filesystem drivers, can be written in type-safe Haskell code this way.

Figure 7.1 shows a typical resulting software stack for our implementation of Memcached we present in Section 7.7.3. A thin operating system layer between the virtual machine environment and the lower-level application components houses the necessary runtime system for the Haskell language. The TCP/IP engine in the network subsystem builds the bridge between the packet-based interface on the operating system level and the application-level protocol. The main application communicates both with the operating system directly as well as its application-specific protocol handler. Optional features, like different protocol variants and specific service functions are depicted with dashed lines.

7.7.2 Runtime implementation

A good starting point for a thin layer to execute the Haskell runtime without a fully featured operating system already exists in form of the *Haskell Lightweight Virtual Machine (HaLVM)* [Gal13a]. It currently allows direct execution of Haskell programs on the Xen Hypervisor [BDF+03] and also has some companion projects implementing higher-level operating system functionality, like a TCP/IP stack with *HaNS* [Gal13b]. As pointed out in Section 7.5.1, our goal with Tailored Memcached is to provide a platform for simple network services in IaaS clouds, so support for a wider range of virtualization environments needs to be added. This involves rewriting all calls to the Xen Hypervisor in HaLVM to perform equivalent tasks on fully virtualized Intel x86 hardware and providing an abstract driver interface for different kinds of virtual network devices. Additionally, the current HaLVM and HaNS subsystems are neither reconfigurable with an explicit feature model nor do they support any aspect-oriented features as described in Section 7.5.3.

7.7.3 Current Prototype Work

We have a preliminary Haskell reimplementation of Memcached, dubbed HsMemcached, as a typical cloud-based network service that serves as the primary test case. It basically provides a key/value store with some extensions for direct data manipulation and atomic operations. In

contrast to the original implementation, most functions can individually be enabled or disabled at compile time and it also supports different in-memory data structures as storage backends. The reconfiguration is realized using the EffectiveAdvice [OSC10] programming technique already discussed in Section 7.5.3. Memcached was chosen because of its manageable complexity while still presenting plenty configuration options, as it supports several network protocol variants (UDP/TCP, text/binary) and command sets. An in-memory storage service also represents the worst case scenario for the Haskell programming language: Since the language is pure, extra work is needed in the program code to explicitly keep track of global state and side effects, and a Memcached implementation almost entirely consists of such operations.

In order to compare the TCB of different components typically used on IaaS clouds and our own prototype, we measured the Source Lines of Code (SLoC) using *sloccount* [Whe01] of various software packages. The results are shown in Table 7.1 and lists the following items we examined from left to right: The Hotspot Java Virtual Machine from OpenJDK 7u6 (without class libraries), the Linux kernel in version 3.2.48, the HaLVM runtime including network support and Haskell libraries, the C reference implementation of Memcached and finally own reconfigurable HsMemcached. Test cases, benchmarks and also any platform specific parts not relevant for the Intel x86 platform were stripped from the source code before measurement in all cases. We also removed many subsystems from the Linux kernel (i.e. audio support, esoteric networking features, most device drivers) that probably can not be activated in a IaaS cloud setting anyways to get a better estimate of what might actually be reachable code once the system is running. The last two columns show the actual TCB required to run a standard Memcached on Linux, compared to our implementation in context of Tailored Memcached. The overall code that is needed in the VM environment to provide the same service is almost an order of magnitude smaller than using the reference implementation. For our tailored HsMemcached we included all possible optional features, the entire HaLVM and HaNS layers, and all the libraries the system depends upon, even if many of the modules in these libraries will never end up in the produced binary.

More importantly, the majority of the code present on the platform is written in strictly typed Haskell, as opposed to error-prone C with the traditional approach.

7.8 Conclusion

The goal of Tailored Memcached is to leverage advanced type systems and functional programming to provide an example for tailored services with an emphasis on security and reliability. The overall trusted computing base of a deployed service can be made one magnitude smaller than with today's common cloud platforms using tailored software. We achieve this small size using a specialized runtime and fine-grained software tailoring techniques, while still being able to support a wide range of applications. A first prototype implementation of a tailored Memcached shows promising results.

Chapter 8

S3 Confidentiality Proxy

Chapter Authors: Alexander Bürger

We present the S3 confidentiality proxy, which ensures confidentiality of data stored in untrusted commodity cloud storage. The encryption keys are not stored in the storage cloud but in the proxy itself. Hence the cloud provider or any other third party has no means to decrypt the data. The immense value of such an component became very prominent this year in the light of the PRISM scandal. The S3 proxy was implemented in two variants. The first is a standalone S3 Proxy Appliance which can be put into an office and provides a standard network fileshare to the user. The data stored there is transparently encrypted and put into the commodity cloud storage. The second variant is the integration into the TrustedServer. The security kernel was extended with the S3 proxy functionality and the encryption is coupled with the TVD encryption. This setup allows the secure integration of legacy cloud storage into the Trusted Infrastructure Cloud. The principle functionality of both integrations is depicted in Figure 8.1.

8.1 S3 Proxy Appliance

This section describes the S3 Proxy Appliance, the standalone solution for storing encrypted data within commodity clouds.

8.1.1 Overview

The S3 Proxy Appliance conduces as a standalone transparent encryption component for all connected clients on the internal network side. All traffic to/from the attached S3-storage is transparently encrypted and decrypted respecetively, with a user's chosen passphrase.

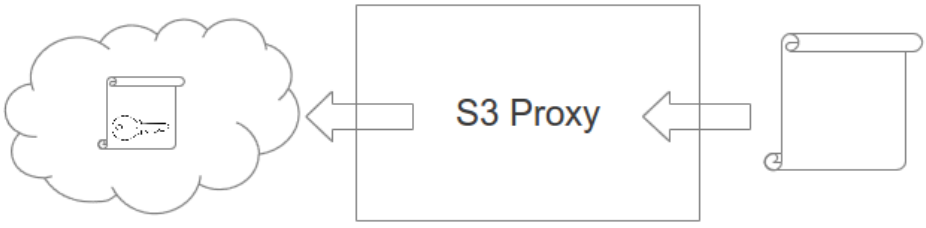


Figure 8.1: Principal functionality of the S3 confidentiality proxy

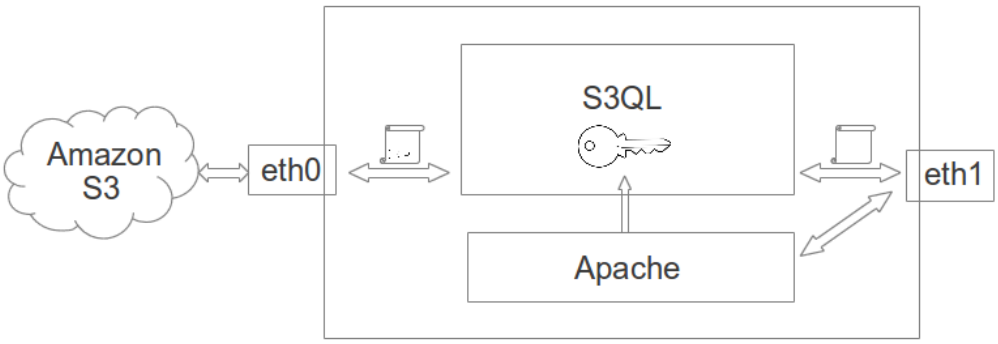


Figure 8.2: Internal functionality of the S3 Proxy Appliance

8.1.2 Technical implementation

The S3 Proxy Appliance runs with an operating system with a stripped down linux kernel, adapted just to support the given hardware at first. On top of it, only software components are installed, that are essential for the rudimentary service provisioning, namely DHCPD, SMB/CIFS, S3QL and an APACHE webserver. Software components, not facilitating the inteded use are not installed at all and cannot be reinstalled after the appliance is produced.
The production mechanism, described in TC-D2.1.2 chapter 6 ensures, that the machine's software state is not circumvented intentionally or accidentialy by a user, protecting the appliance's integrity. In addition, a full-disk encryption and an integrity measurement, based on TPM-capabilities protect the machine's integrity even if the appliance is powered down. No dedicated user account exists so that all login attempts fail.

Once, the machine is configured for its intended use, there is no direct interaction to the machine required anymore, in order to fullfill the envisaged task. To sum up, the special features of the hardware are two physical network connection possibilities and the TPM. One network port (referred to as eth1) delivers DHCP-adresses in the Class-C network range to attached components. See figure Figure 8.2 The other network port (referred to as eth0) provides a connection to the outside world, namely the internet. The direct communication between these two networks on the appliance itself is strictly prohibited by iptables-rules, preventing plaintext data leakage.

The S3 storage applicance offers an automounting mechanism for commodity cloud storage via the external (eth0), and a fileserver service (smbfs) on the internal network interface (eth1). As the central component, a FUSE[1] file system (S3QL[2]) is used, which also includes a cipher-module responsible for encrypting data, before it is being transferred to the online data storage.

S3QL is a standard conforming, full featured UNIX file system that is conceptually indistinguishable from any local file system. Furthermore, S3QL has additional features like compression, encryption, data de-duplication, immutable trees and snapshotting which makes it especially suitable for online backup and archival.
When the file system is created initially, mkfs.s3ql generates a 256 bit master key by reading from /dev/random. The master key is encrypted with the passphrase that is entered by the user, and then stored with the rest of the file system data. Since the passphrase is only used to access the master key (which is used to encrypt the actual file system data), the passphrase can easily be changed. Data is encrypted with a new session key for each object and each upload. The session key is generated by appending a nonce to the master key and then calculating the SHA256 hash. The nonce is generated by concatenating the object id and the current UTC time as a 32 bit float. Once the session key has been calculated, a SHA256 HMAC is calculated over the data that is to be uploaded. Afterwards, the data is compressed with the LZMA Bz2 algorithm or LZ and the HMAC inserted at the beginning. Both HMAC and compressed data are then encrypted using 256 bit AES in CTR mode. Finally, the nonce is inserted in front of the encrypted data and HMAC, and the packet is send to the backend as a new S3 object.

Data flowing from the internally attached clients via eth1 to the S3 Proxy appliance will be encrypted with a passphrase and transferred to the online data storage afterwards via eth0, which is strictly isolated from eth1. See figure Figure 8.2. Vice versa, a pull-request of online stored data will lead to a transfer of the encrypted data from the cloud provider to the S3 proxy appliance. Not till then it will be decrypted and allocated to the plaintext smb-share for delivery to the attached clients, via the internal network interface only. Since the encryption key resides only within the appliance itself, at no time the cloud provider does have access to it.

As well as the AWS credentials, the passphrase to en-/decrypt the online stored data is kept on the appliance itself. Therefore all user-related data, to login to Amazon (in the presented scenario) and the users' chosen passphrase is protected by the full-disk-encryption of the appliance, even when it's powered off. The access clearances to the decrypted data can be steered by fine-grained access-profiles of the samba-server. Thus the S3 Proxy Appliance can be used in a private, as well as in a business environment, considering different roles of the appliance's administrator.

Configuration

For configuration purposes of the S3 proxy apppliance, an apache webserver provides a web-interface, where the user is able to choose between different commodity cloud-storage providers (such as Amazon, Rackspace, SkyDrive, GoogleDrive, etc.) to be used. In addition, the user is forced to set a passphrase for the en- and decryption process of data to be saved to the chosen online data storage provider.
This configuration form can only be reached from the internal network (via eth1), preventing a reconfiguration from outside the class-C network. As an example the principal steps to configure and use the appliance is described on the basis of Amazon's S3 storage, which can be used free

[1] http://fuse.sourceforge.net/
[2] http://code.google.com/p/s3ql/

Figure 8.3: Configuration interface of the S3 Proxy Appliance

of charge.
From the main form (see figure Figure 8.3), the user chooses a storage provider (here: Amazon). After that, the presented credential-fields have to be filled. The backend login and password for accessing S3 are not the user id and password used to login into Amazon Webpage, but the "AWS access key id" and "AWS secret access key" provided by the users AWS "MyAccountAccess Identifiers". The user enters the URL to the S3-bucket which has had to be created via "Amazon AWS Console" beforehand. In addition, a passphrase has to be chosen, in order to en-/decrypt data transferred to the commodity cloud. If no password is provided, the online data storage will be mounted in plaintext. Potential already existing encrypted files within the Amazon S3-bucket, will stay encrypted of course. Although they are visible, the files cannot be decrypted.
Clicking "OK" results in an attempt to mount the S3-bucket with the provided configuration to the S3 Proxy Appliance permanently. Finally, the applicance ensures the provisioning of the mountpoint to all attached clients on the internal network, so that the external storage is disposable immediately.

8.2 S3 Proxy functionality within TrustedServer

This section describes the port of the S3 Proxy Appliances' functionality to TrustedServer.

8.2.1 Overview

In principle, the capabilities are almost the same as for the standalone version, with the exception that the whole functionality lies in the TrustedServer's computing base, and is therefore considered as a trusted component.
As the standalone version, the TrustedServer S3Proxy solution provides a transparent encryption for all internally connected virtual machines (compartments) running on the Trusted Server. In addition, the isolation of Trusted Virtual Domains (TVDs) i.e. the information flows between them, is taken into consideration by inserting a cryptographic stacked file system between s3ql and the samba file server.

8.2.2 Technical implementation

Technically, the S3QL file system serves as the basis of the solution (as depicted in figure Figure 8.4), but is just used to mount the external storage. The cryptographic capabilities of s3ql are not used here, since a more fine grained PKI is needed, in order to disinguish between data stored from different TVDs. This is realized by adding an ecryptfs-layer on top of the initially (plaintext) mounted s3ql filesystem, which is able to handle multiple keys and to report different mount-points to the samba file server. The used keys for en-/decryption on a per-file-basis, are derived from the TVDs, configured within the TrustedInfrastructure, allowing a TVD-wide sharing of the mountpoint. That means, that a the cloud storage endpoint can be shared between different TVDs on the client side without apprehending data leakageor loss. In that sense, the S3 Proxy Appliance serves as an exit-node for TrustedInfrastructure's TVD concept.

Configuration

The configuration is described in TC-D2.1.4-2.3.3 section 3.1.10.

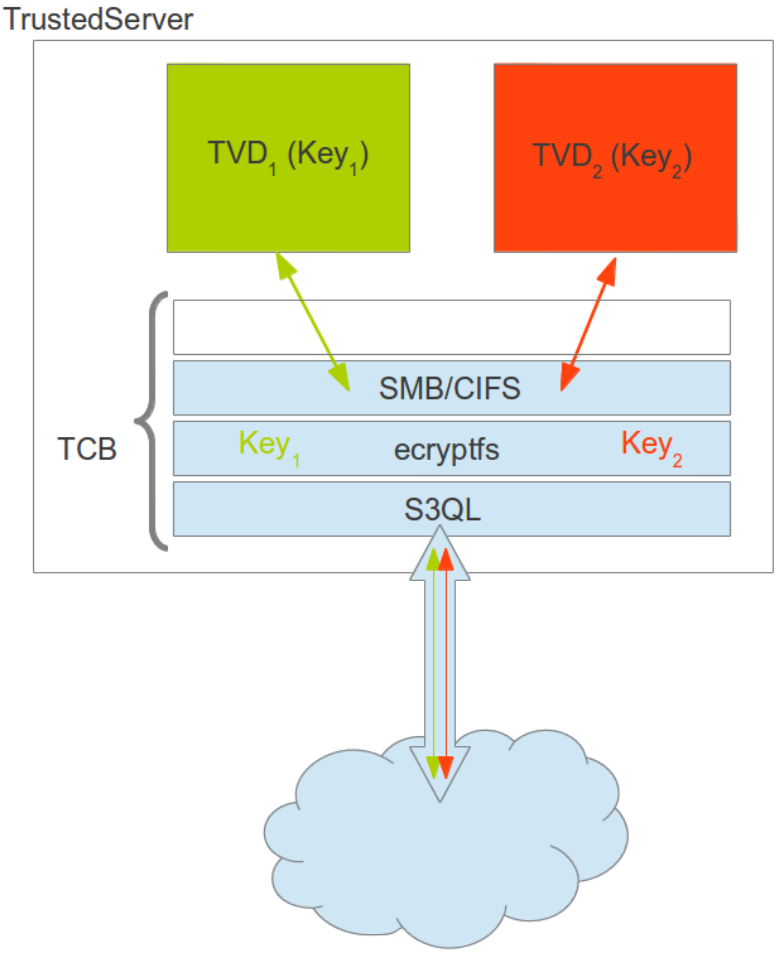


Figure 8.4: S3 Confidentiality Proxy within TrustedServer

Bibliography

[B+13a] Alysson Bessani et al. TClouds – D2.2.4 Adaptive Cloud-of-Clouds Architecture, Services and Protocols. Deliverable D2.2.4, TClouds Consortium, September 2013.

[B+13b] Sören Bleikertz et al. TClouds – D2.3.4 Automation and Evaluation of Security Configuration and Privacy Management. Deliverable D2.3.4, TClouds Consortium, September 2013.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[Bra11] Edwin C. Brady. IDRIS — Systems Programming Meets Full Dependent Types. In *Proceedings of the 5th ACM workshop on Programming languages meets program verification*, PLPV '11, pages 43–54, New York, NY, USA, 2011. ACM.

[BS+13] Sören Bleikertz, Norbert Schirmer, et al. TClouds – D2.1.4/D2.3.3 Proof of Concept Infrastructure / Implementation of Security Configuration and Policy Management. Deliverable D2.1.4/D2.3.3, TClouds Consortium, April 2013.

[CB04] Sébastien Carlier and Jérémy Bobbio. hOp. Announcement on Haskell-cafe mailing list. http://www.haskell.org/pipermail/haskell-cafe/2004-February/005839.html, February 2004.

[CL99] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, 1999.

[CLM+10] Luigi Catuogno, Hans Löhr, Mark Manulis, Ahmad-Reza Sadeghi, Christian Stüble, and Marcel Winandy. Trusted Virtual Domains: Color Your Network. *Datenschutz und Datensicherheit (DuD) 5/2010*, pages 289–294, 2010.

[CX05] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. *SIGPLAN Not.*, 40(9):66–77, September 2005.

[D+13] Mina Deng et al. TClouds – D3.1.5 Proof of concept for home healthcare. Deliverable D3.1.5, TClouds Consortium, October 2013.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating*

systems principles, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[DKSP+12] Tobias Distler, Simon Kuhnle, Wolfgang Schröder-Preikschat, Sören Bleikertz, Christian Cachin, Imad M. Abbadi, Cornelius Namiluko, Andrew Martin, Rüdiger Kapitza, Johannes Behl, Klaus Stengel, Seyed Vahid Mohammadi, Sven Bugiel, Stefan Nürnberger, Hugo Ideler, Ahmad-Reza Sadeghi, Mina Deng, Emanuele Cesena, Antonio Lioy, Gianluca Ramunno, Roberto Sassu, Davide Vernizzi, and Alexander Kasper. *D2.1.2 – Preliminary Description of Mechanisms and Components for Single Trusted Clouds*. TClouds Project Deliverable, September 2012.

[Fit04] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):5–, August 2004.

[Fou13a] Eclipse Foundation. AspectJ Programming Language. http://eclipse.org/aspectj/, 2013.

[Fou13b] Free Software Foundation. The GNU Compiler for the Java Programming Language. http://gcc.gnu.org/java/, 2013.

[Gal13a] Galois, Inc. HaLVM – Haskell Lightweight Virtual Machine. http://corp.galois.com/halvm, 2013.

[Gal13b] Galois, Inc. HaNS Network Stack. http://corp.galois.com/hans, 2013.

[GSVL09] A. N. Bessani G. S. Veronese, M. Correia and L. C. Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *Proceedings of the 28th International Symposium on Reliable Distributed Systems*, SRDS '09, pages 135–144, 2009.

[HJLT05] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in Haskell. In *Proceedings of the 10th ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 116–128, New York, NY, USA, 2005. ACM.

[HL07] Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review*, 41:37–49, 2007.

[int10] Using Intel AES New Instructions and PCLMULQDQ to Significantly Improve IPSec Performance on Linux. http://www.intel.de/content/dam/www/.../aes-ipsec-performance-linux-paper.pdf, 2010.

[KBC+12] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th European Conference on Computer Systems*, EuroSys '12, pages 295–308, 2012.

[KBS+12] Rüdiger Kapitza, Johannes Behl, Klaus Stengel, Tobias Distler, Simon Kuhnle, Wolfgang Schröder-Preikschat, Christian Cachin, and Seyed Vahid Mohammadi. *Cheap BFT*, pages 88–111. In [DKSP+12], September 2012.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.

[KTD+13] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium*, NDSS'13, Berkeley, CA, USA, 2013. USENIX Association.

[KTS+09] Christian Kastner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. FeatureIDE: A tool framework for feature-oriented software development. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 611–614, Washington, DC, USA, 2009. IEEE Computer Society.

[LLC13] Excelsior LLC. Excelsior JET. http://www.excelsior-usa.com/jet.html, 2013.

[LSHSP12] Daniel Lohmann, Olaf Spinczyk, Wanja Hofer, and Wolfgang Schröder-Preikschat. The Aspect-Aware Design and Implementation of the CiAO Operating-System Family. *Transactions on Aspect-Oriented Software Development (TAOSD IX)*, IX:168–215, 2012.

[LST+06] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. *SIGOPS Oper. Syst. Rev.*, 40(4):191–204, April 2006.

[McB02] Conor McBride. Faking it – Simulating dependent types in Haskell. *Journal of functional programming*, 12(4-5):375–392, 2002.

[MMS+13] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsosand David Scott, Balraj Singh, Thomas Gazagnaireand Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the 18th international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'13, New York, NY, USA, 2013. ACM.

[OSC10] Bruno C. d. S. Oliveira, Tom Schrijvers, and William R. Cook. EffectiveAdvice: Disciplined Advice with Explicit Effects. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pages 109–120, New York, NY, USA, 2010. ACM.

[PBWH+11] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the Library OS from the Top Down. In *Proceedings of the 16th international conference on Architectural Support for Programming*

Languages and Operating Systems, ASPLOS'11, New York, NY, USA, 2011. ACM.

[Per13] Nuno Pereira. TClouds – D3.2.5 Smart Lighting System Final Report. Deliverable D3.2.5, TClouds Consortium, October 2013.

[R+13] Gianluca Ramunno et al. TClouds – D2.4.3 Final Reference Platform and Test Case Specification. Deliverable D2.4.3, TClouds Consortium, October 2013.

[RI11] Chris Rohlf and Yan Ivnitskiy. Attacking Clientside JIT Compilers. http://www.matasano.com/research/Attacking_Clientside_JIT_Compilers_Paper.pdf, 2011.

[S+12a] Roberto Sassu et al. TClouds – Initial Component Integration, Final API Specification, and First Reference Platform. Deliverable D2.4.2, TClouds Consortium, October 2012.

[S+12b] Norbert Schirmer et al. TClouds – Preliminary Description of Mechanisms and Components for Single Trusted Clouds. Deliverable D2.1.2, TClouds Consortium, September 2012.

[SAS+99] S. Doaitse Swierstra, Pablo R. Azero Alcocer, Joao Saraiva, Doaitse Swierstra, Pablo Azero, and JoÃčo Saraiva. Designing and Implementing Combinator Languages. In *Third Summer School on Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 150–206. Springer-Verlag, 1999.

[SGSP02] Olaf Spinczyk, Andreas Gal, and Wolfgang SchrÃűder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems*, CRPIT '02, pages 53–60. Australian Computer Society, Inc., February 2002.

[SK99] Bruce Schneier and John Kelsey. Secure audit logs to support computer forensics. *ACM Trans. Inf. Syst. Secur.*, 2(2):159–176, May 1999.

[SSWSP10] Isabella Stilkerich, Michael Stilkerich, Christian Wawersich, and Wolfgang Schröder-Preikschat. KESO: An Open-Source Multi-JVM for Deeply Embedded Systems. In Tomas Kalibera and Jan Vitek, editors, *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 109–119, New York, NY, USA, 2010.

[SU06] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science Inc., New York, NY, USA, 2006.

[SVCBL10] G Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Ebawa: Efficient byzantine agreement for wide-area networks. In *High-Assurance Systems Engineering (HASE), 2010 IEEE 12th International Symposium on*, pages 10–19. IEEE, 2010.

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

[TSD+12] Reinhard Tartler, Julio Sincero, Christian Dietrich, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Revealing and Repairing Configuration Inconsistencies in Large-Scale System Software. *International Journal on Software Tools for Technology Transfer (STTT)*, 14(225):531–551, 2012.

[VS13] Paulo Viegas and Paulo Santos. TClouds – D3.2.4 Smart Lighting System Final Prototype. Deliverable D3.2.4, TClouds Consortium, September 2013.

[Whe01] David A. Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/, 2001.

[WHVDV+13] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine: An Approachable Virtual Machine For, and In, Java. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):30:1–30:24, January 2013.