TClouds

SEVENTH FRAMEWORK PROGRAMME

D2.1.2
Preliminary Description of Mechanisms 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 / D2.1.2 / 1.0
Activity and Work package contributing to deliverable:	Activity 2 / WP 2.1
Due date:	September 2012 – M24
Actual submission date:	28th September 2012

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

Abstract:	cf. Executive Summary
Keywords:	Insider Attacks, Remote Attestation, Trusted Virtual Domains, Mobile Devices, Fault Tolerance, Trusted Computing

Editor

Norbert Schirmer (SRX)

Contributors

Tobias Distler, Simon Kuhnle, Wolfgang Schröder-Preikschat (FAU)

Sören Bleikertz, Christian Cachin (IBM)

Imad M. Abbadi, Cornelius Namiluko and Andrew Martin (OXFD)

Rüdiger Kapitza, Johannes Behl, Klaus Stengel (TUBS)

Seyed Vahid Mohammadi (KTH Royal Institute of Technology)

Sven Bugiel, Stefan Nürnberger, Hugo Ideler, Ahmad-Reza Sadeghi (TUDA)

Mina Deng (PHI)

Emanuele Cesena, Antonio Lioy, Gianluca Ramunno, Roberto Sassu, Davide Vernizzi (POL)

Alexander Kasper (SRX)

Disclaimer

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

The information in this document is provided as is, and no warranty is given or implied that the information is fit for any particular purpose.

The user thereof uses the information at its sole risk and liability. The opinions expressed in this deliverable are those of the authors. They do not necessarily represent the views of all TClouds partners.

Executive Summary

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. Moreover as all cloud customers use the same resources, the infrastructure is shared among multiple clients, usually called tenants in this context, which may be competitors. Hence proper isolation of cloud customers becomes of crucial importance for acceptance of cloud offerings.

In this deliverable, we consolidate and analyze the requirements for building a trusted infrastructure cloud. The research and developments presented can be categorized into the following areas representing the key challenges for building a trusted infrastructure cloud:

Trust: As the customer's resources are hosted by the cloud provider, trust between the cloud provider and the customer has to be established. The threat of insider attacks performed by the cloud providers' employees have to be carefully analysed and security measures have to be established to mitigate the risks. Remote attestation is one technical means to establish trust relationships between the customer and the provider's sites. We develop deployment and key management schemes which reduce the level of trust needed required from cloud provider.

Confidentiality / Integrity: The customer's data is stored and processed at the providers' site. Confidentiality and integrity of the data has to be ensured throughout the complete life cycle of the data and seamlessly from the customers end-points into the cloud. In this context the concept of trusted virtual domains (TVD) is employed and we study how to integrate mobile devices as the customers end-points into the infrastructure to complete the picture.

Resilience: As resources are no longer under control of the customer and more and more business critical applications move into the cloud, availability and fault-tolerance of the cloud infrastructure becomes a crucial prerequisite for the operational business of the customers. We present a scheme for fault tolerance which reduces the costs by reducing the number of replicas needed.

Audit: As incidents may occur in the cloud infrastructure, proper means for audit and forensics have to be brought into place. A secure, tamper proof logging mechanism appears to be a crucial core ingredient for legal compliance, which we have designed in this deliverable.

Contents

1 Introduction **1**
1.1 TClouds — Trustworthy Clouds . . . 1
1.2 Activity 2 — Trustworthy Internet-scale Computing Platform . . . 1
1.3 Workpackage 2.1 — Trustworthy Cloud Infrastructure . . . 2
1.4 Deliverable 2.1.2 — Preliminary Description of Mechanisms and Components for Single Trusted Clouds . . . 3

2 Insider Attack Analysis **6**
2.1 Introduction . . . 6
2.2 Insiders . . . 7
2.3 Conceptual Models . . . 7
2.3.1 Organisational View . . . 8
2.3.2 Assets and Clients . . . 8
2.3.3 Infrastructure Model . . . 9
2.3.4 Procedure for Identifying Potential Insiders and Insiders . . . 9
2.4 Insiders Analysis for Home Healthcare . . . 10
2.4.1 Scenario . . . 10
2.4.2 Model Instance . . . 11
2.4.3 Identifying Potential Insiders and Insiders . . . 11
2.4.4 Insider Threat Analysis . . . 15
2.5 Related Work . . . 17
2.6 Conclusion . . . 18

3 Cryptography-as-a-Service **19**
3.1 Introduction . . . 19
3.2 Model and Requirements . . . 21
3.2.1 Trust and Adversary Model . . . 22
3.2.2 Objectives and Requirements . . . 23
3.3 Design and Implementation . . . 24
3.3.1 Client-controlled CryptoDomain DomC . . . 24
3.3.2 Security Extensions to the Xen Hypervisor . . . 26
3.3.3 Evaluation . . . 32
3.3.4 Scalability Challenges and Possible Solutions . . . 34
3.4 Security . . . 35
3.5 On the Implementation on Amazon EC2 . . . 37
3.6 Medical Use-Case . . . 38
3.6.1 Personal Healthcare Service . . . 38
3.6.2 Security Requirements and *CaaS* Benefits . . . 39
3.7 Related Work . . . 39
3.8 Conclusion and Future Work . . . 42

4 Remote Attestation **43**
4.1 Introduction . . . 43
4.2 On Scalability . . . 43
4.2.1 Our contribution . . . 44
4.3 On Code-Diversity . . . 44
4.3.1 Experimental methodology . . . 45
4.3.2 Client-side setup . . . 45
4.3.3 Reference database: internals and data . . . 47
4.3.4 Reference database: verifying a platform . . . 49
4.3.5 Remote attestation: experimental results . . . 51
4.3.6 Remote attestation: the costs . . . 51
4.4 On Configuration . . . 53
4.5 Conclusions and future work . . . 55

5 Mobile Device Clouds **56**
5.1 Motivation . . . 56
5.1.1 Predominant Web-Service Delivery Paradigms . . . 56
5.1.2 Extended Security Perimeter of Cloud-Based Virtual Infrastructures . . . 57
5.1.3 The Need for Mobile Platform Security . . . 58
5.2 Practical and Lightweight Domain Isolation on Android . . . 58
5.2.1 Introduction . . . 58
5.2.2 Android . . . 61
5.2.3 Problem Description and Model . . . 63
5.2.4 Design of TrustDroid . . . 65
5.2.5 Implementation and Evaluation . . . 70
5.2.6 Discussion . . . 73
5.2.7 Related work . . . 75
5.2.8 Conclusion . . . 77

6 Initialization and Update Mechanisms for TrustedServer **78**
6.1 Introduction to Initilization . . . 78
6.2 Initilization . . . 78
6.3 HDD Layout . . . 80
6.3.1 Bootloader partition . . . 80
6.3.2 Init partition . . . 80
6.4 HDD Encryption (FDE) . . . 81
6.5 TPM initialization . . . 82
6.6 Public Key Infrastructure (PKI) . . . 82
6.7 Platform Configuration Certificates . . . 82
6.8 Key sealing . . . 84
6.9 Updatekey and Updates . . . 84
6.10 Attestation and Identity Key . . . 85

7 Cheap BFT **88**
7.1 Introduction . . . 88
7.2 Preventing Equivocation . . . 90
7.2.1 From $3f + 1$ Replicas to $2f + 1$ Replicas . . . 90
7.2.2 The CASH Subsystem . . . 91

7.3 CheapBFT . . . 95
7.3.1 System Model . . . 95
7.3.2 Resource-efficient Replication . . . 95
7.3.3 Fault Handling . . . 96
7.4 Normal-case Protocol: CheapTiny . . . 97
7.4.1 Client . . . 97
7.4.2 Replica . . . 97
7.5 Transition Protocol: CheapSwitch . . . 100
7.5.1 Initiating a Protocol Switch . . . 100
7.5.2 Creating an Abort History . . . 101
7.5.3 Validating an Abort History . . . 101
7.5.4 Processing an Abort History . . . 102
7.5.5 Handling Faults . . . 103
7.6 Fall-back Protocol: MinBFT . . . 104
7.6.1 Protocol . . . 104
7.6.2 Protocol Switch . . . 104
7.7 Evaluation . . . 105
7.7.1 Normal-case Operation . . . 105
7.7.2 Protocol Switch . . . 106
7.7.3 ZooKeeper Use Case . . . 106
7.8 Discussion . . . 108
7.9 Related Work . . . 110
7.10 Conclusion . . . 111

8 Tailored Memcached **112**
8.1 Introduction . . . 112
8.2 Memcached Feature Sets . . . 112
8.2.1 Feature description . . . 113
8.2.2 Feature groups . . . 113
8.3 Implementing Variability . . . 114
8.3.1 The Haskell programming language . . . 115
8.3.2 Aspect-Oriented Programming in Haskell . . . 115
8.4 Conclusion . . . 116

9 Log Service **117**
9.1 Background . . . 117
9.2 Design . . . 118
9.2.1 Building blocks . . . 119
9.2.2 Data exchange . . . 120
9.3 Implementation . . . 121
9.3.1 Core library . . . 121
9.3.2 Bindings . . . 122
9.3.3 Building blocks implementation . . . 122
9.3.4 Integration in OpenStack “Essex” . . . 123

Bibliography **124**

List of Figures

1.1 Graphical structure of WP2.1 and relations to other workpackages. 3
1.2 Deliverable dependencies for D2.1.2 5

2.1 High-Level Organisational View of the Cloud 8
2.2 A view around the Client 9
2.3 A Breakdown of a Cloud Resource 9
2.4 Potential Insiders and Insiders Identification Process 10
2.5 Model Instance for the Home Healthcare Scenario 11
2.6 Physical Infrastructure 12
2.7 VMM Component Breakdown 13
2.8 VM access 13
2.9 Application Access 14

3.1 Typical IaaS cloud model including our adversary and trust model. 21
3.2 Basic idea of *CaaS*: Establishment of a separate, coupled security-domain, denoted as `DomC`, for critical cryptographic operations. 25
3.3 `DomC` usage modes: `DomU` can use `DomC` either as *Virtual Security Module* (e.g., HSM) or to transparently encrypt its storage or network data as a *Secure Virtual Device*. 25
3.4 Additional memory access control in Xen hypervisor on inter-domain (shared) memory access. 27
3.5 Workflow for `DomU` and `DomC` image provisioning and instantiation. 28
3.6 Trust establishment and VM instantiation 29
3.7 Booting `DomU` and coupling with corresponding DomC 31
3.8 Comparing the signing performance of a software-based HSM residing in `DomU` vs. `DomC`. 34
3.9 Cloud based personal healthcare service for depressed patients. 39

4.1 */etc/sysconfig/i18n* 54
4.2 */etc/udev/rules.d/70-persistent-cd.rules* 55

5.1 Classical star topology: Data flow from central cloud to mobile devices. 57
5.2 Distributed data: Flow within *Mobile Device Cloud*. 57
5.3 Android architecture 61
5.4 Communication channels and respective access control mechanisms in Android. 63
5.5 Approaches to isolation: (a) *TrustDroid*; (b) OS-level virtualization; (c) Hypervisor/VMM 65
5.6 *TrustDroid* architecture with isolation of different colors **(A)** in the middleware, **(B)** at the file system/default Linux IPC level, and **(C)** at the network level. . . . 66
5.7 Coloring of data (1) and isolation of data from different colors in the (2) System Content Providers and (3) System Service. 68

5.8 Control flow for the installation of a new application in case the installation package contains a RIM certificate (solid lines). If no RIM certificate is included in the package, this flow deviates (dashed lines). . . . 71

6.1 High-Level disk layout of a TrustedServer . . . 79
6.2 CoData to be included in the client certificate for Key-Based TLS . . . 83
6.3 Two example PCR chains . . . 83
6.4 Configuration example 1 for sealing . . . 84
6.5 Configuration example 2 for sealing . . . 85
6.6 Data to be included in the client certificate for Key-Based TLS . . . 86
6.7 Sequence diagram for a successful key-based authentication . . . 87

7.1 Implementation of CASH's trusted counter. . . . 92
7.2 Creation of a message certificate mc for a message m using the FPGA-based trusted CASH subsystem. . . . 93
7.3 CheapBFT architecture with two active replicas and a passive replica ($f = 1$) for normal-case operation. . . . 96
7.4 CheapTiny protocol messages exchanged between a client, two active replicas, and a passive replica ($f = 1$). . . . 97
7.5 CheapTiny agreement protocol for active replicas. . . . 98
7.6 CheapTiny execution-stage protocol run by active replicas to execute requests and distribute state updates. . . . 99
7.7 CheapTiny execution-stage protocol run by passive replicas to process updates provided by active replicas. . . . 99
7.8 CheapSwitch protocol messages exchanged between clients and replicas during protocol switch ($f = 1$). . . . 101
7.9 Dependencies of UPDATE (UPD$_*$) and COMMIT (COM$_*$) messages contained in a correct CheapTiny abort history for four requests a, b, c, and d ($f = 1$). . . . 103
7.10 Performance and resource-usage results for a micro benchmark with empty requests and empty replies. . . . 106
7.11 Performance and resource-usage results for a micro benchmark with empty requests and 4 kilobyte replies. . . . 107
7.12 Performance and resource-usage results for a micro benchmark with 4 kilobyte requests and empty replies. . . . 107
7.13 Response time development of CheapBFT during a protocol switch from CheapTiny to MinBFT. . . . 108
7.14 Performance and resource-usage results for different BFT variants of our ZooKeeper service for workloads comprising different mixes of read and write operations. . 109

9.1 Schneier and Kelsey's log entry creation scheme. . . . 118
9.2 Log Service high level view. . . . 119
9.3 Log Storage high level view. . . . 119
9.4 Cloud Component with additional Log Service module. . . . 120
9.5 Schneier and Kelsey's modified log entry creation scheme. . . . 122
9.6 Log Core low level architecture. . . . 123
9.7 Integration of Log Service in OpenStack. . . . 123

List of Tables

3.1 Using the fio disk benchmark tool . . . 33
3.2 Using the generic dd command . . . 33

4.1 Number of measurements and verify time for all IMA policies and installation flavors. . . . 52
4.2 Bootstrap time in seconds. . . . 53

6.1 Options for sealing configuration . . . 84

7.1 Overhead (in microseconds) for creating and verifying a message certificate in different subsystems. . . . 94
7.2 Size comparison of the trusted computing bases of different subsystems in thousands of lines of code. . . . 95

8.1 List of configurable features . . . 114

9.1 securelog configuration flags. . . . 124

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.

During the second year the focus was the design of the components and building prototypes.

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.

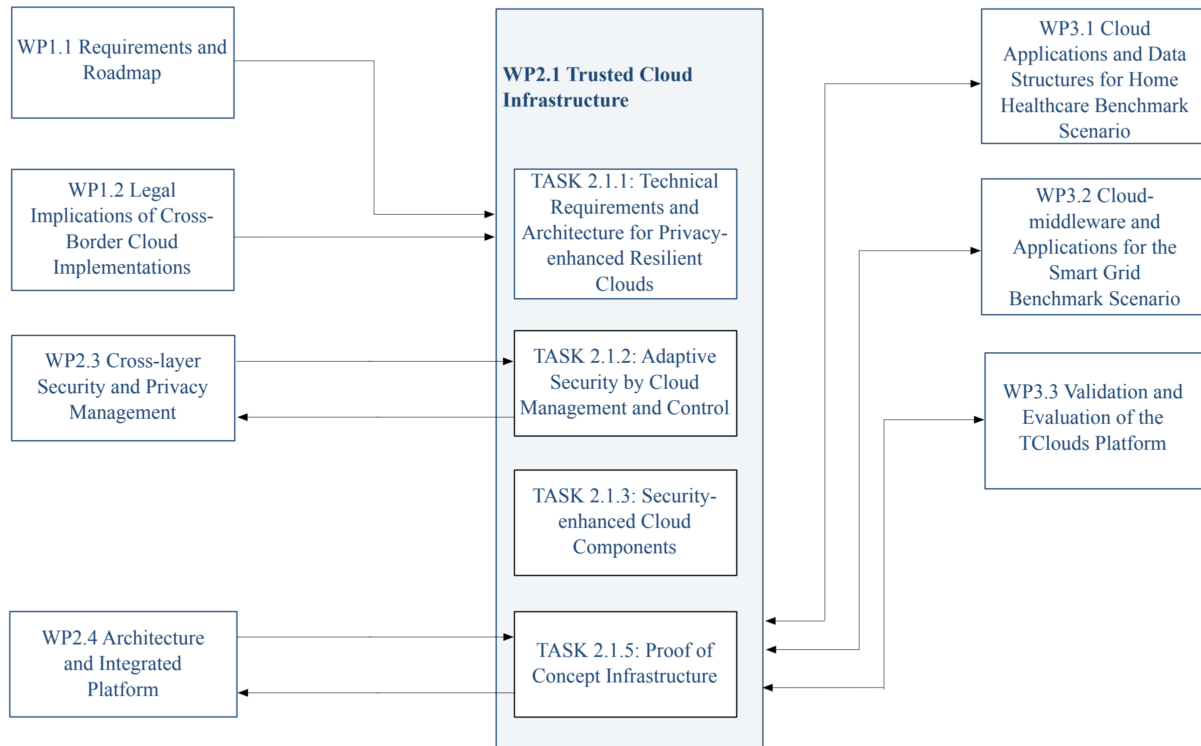


Figure 1.1: Graphical structure of WP2.1 and relations to other workpackages.

1.4 Deliverable 2.1.2 — Preliminary Description of Mechanisms and Components for Single Trusted Clouds

Overview. 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. Moreover, as all cloud customers use the same resources, the infrastructure is shared among multiple clients (usually called tenants in this context) which may be competitors. Hence proper isolation of cloud customers becomes of crucial importance for acceptance of cloud offerings. In this deliverable, we consolidate and analyze the requirements for building a trusted infrastructure cloud. The research and developments presented can be categorized into the following areas representing the key challenges for building a trusted infrastructure cloud:

Trust: As the customers' resources are hosted by the cloud provider, trust between the cloud provider and the customer has to be established. The threat of insider attacks performed by the cloud providers employees have to be carefully analysed and security measures have to be established to mitigate the risks. Remote attestation is one promising technical means to establish trust relationships between the customer and the providers sites. We develop deployment and key management schemes which reduce the level of trust needed required from cloud provider.

Confidentiality / Integrity: The customers' data is stored and processed at the providers' site. Confidentiality and integrity of the data has to be ensured throughout the complete life cycle of the data and seamlessly from the customers end-points into the cloud. In this

context the concept of trusted virtual domains (TVD) is employed and we study how to integrate mobile devices as the customers end-points into the infrastructure to complete the picture.

Resilience: As resources are no longer under control of the customer and more and more business critical applications move into the cloud, availability and fault-tolerance of the cloud infrastructure becomes a crucial prerequisite for the operational business of the customers.

Audit: As incidents may occur in the cloud infrastructure proper means for audit and forensics have to be brought into place. At the core a secure, tamper proof logging mechanism appears to be a crucial core ingredient for legal compliance.

Structure. Chapter 2 analyses the risk of insider attacks in cloud computing especially motivated by the requirements from the medical use case scenario of WP 3.1. The content of this chapter was published in [ANM11]. In Chapter 3 a scheme for 'Cryptography-as-a-service' is proposed which provides the technical means for secure key management within the cloud infrastructure minimizing the trust assumptions on cloud insiders.

Chapters 4, 5, 6 consider different aspects of trusted computing and trusted virtual domains. Remote attestation is discussed in Chapter 4 enabling to build trust in cloud computing nodes, which was published in [CRS+11]. Chapter 5 introduces the integration of mobile end-user devices into trusted virtual domains spanning cloud resources. Chapter 6 gives details on the initialisation and update mechanisms of the TrustedServer (cf. Deliverable D2.4.1 chapter 7.5).

The following two Chapters are concerned with improving resilience within the cloud. Chapter 7 discusses the further developments and results of the resource efficient Byzantine Fault Tolerance (BFT) framework, which was published in [KBC+12]. Chapter 8 elaborates on the concept of tailored services aiming to minimize the computing base of the service to minimizte the runtime overhead and to improve security.

Finally in Chapter 9 the design of a logging service is described.

Deviation from Workplan. This deliverable aligns with the DoW/Annex I, Version 2.

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 dependencies of deliverable D2.1.2 are depicted in Figure 1.2. The present deliverable D2.1.2 is directly related to the year 1 deliverable D2.1.2. Whereas D2.1.1 focused on requirements, a gap analysis and first ideas of a trusted cloud computing environment, this deliverable continues the work by describing concrete designs and a much more mature discussion of the components.

D2.1.2 contributes to the components and the architecture for cloud-computing to the development of the two application scenarios in WP3.1 and WP3.2. Furthermore, the components are validated in the context of WP3.3 ("Validation and Evaluation of the TClouds Platform").

Moreover D2.1.2 has a strong connection to D2.4.2. D2.1.2 focuses on the research aspects and the overall design of the components, which are then integrated into the TClouds proof-of-concept prototypes in D2.4.2. Therefor D2.1.2 reflects the research aspects whereas D2.4.2

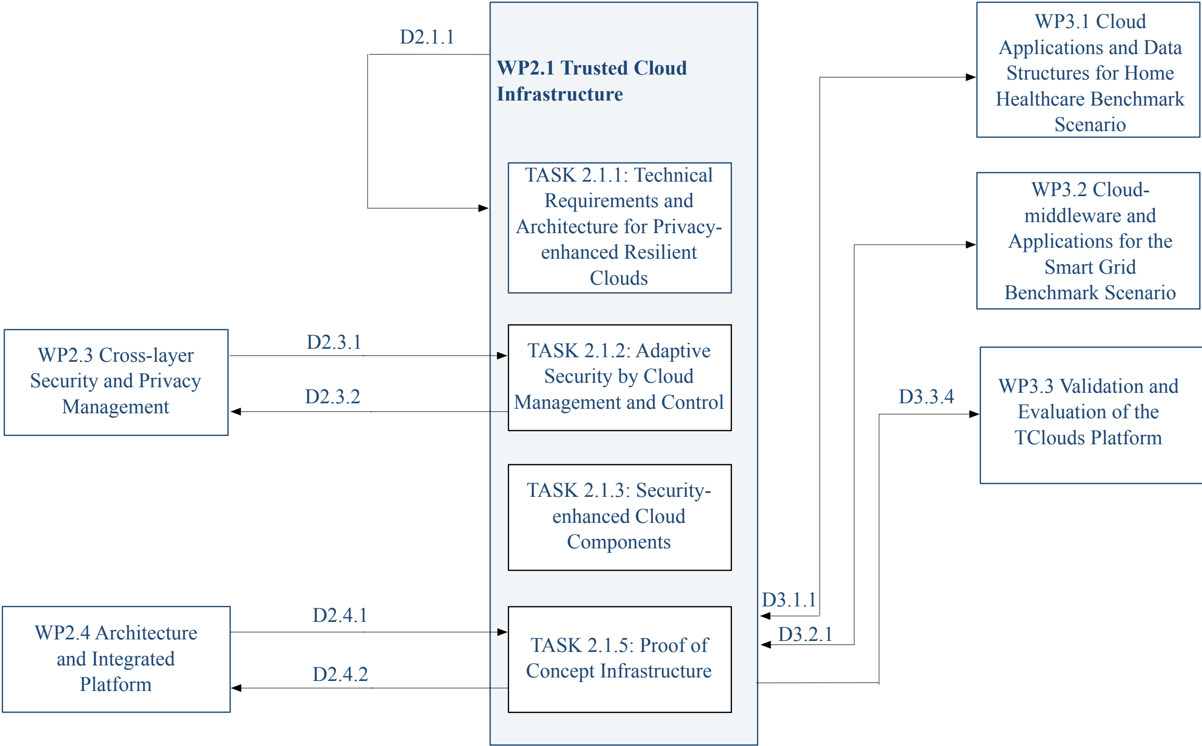


Figure 1.2: Deliverable dependencies for D2.1.2

is more concerned with the engineering and integration aspects. The mapping of legal and application requirements from Activity 1 and Activity 3 to component requirements is described in D2.4.2 as well as D3.3.4.

Chapter 2

Insider Attack Analysis

Chapter Authors:
Imad M. Abbadi, Cornelius Namiluko and Andrew Martin (OXFD)

2.1 Introduction

A cloud is a new buzzword in computing terms, which is defined as *'a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.'*[MG]. Cloud supports three main deployment types: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS) [MG]. IaaS provides the most flexible type for cloud users who prefer to have the greatest control over their resources, while SaaS provides the most restrictive type for cloud users where cloud providers have full control over the virtual resources. In other words, cloud computing provides a full outsourcing support for the SaaS, a partial outsourcing support for PaaS (more specifically it provides the virtual environment and software tools for users helping them to develop and deploy their applications), and a minimal outsourcing support for IaaS (more specifically cloud provider mainly manages the infrastructure components running the virtual machines). In this chapter our analysis mainly focus on IaaS cloud type in which cloud users would typically be organizations.

Insider problem is cited as the most serious security problem and the most difficult problem to deal with [BGHP08, Ric07]. As discussed by Alawneh et. al. [AA11] the insider problem in organizations is mainly caused by the holders of authorized credentials who are typically the internal and authorized employees. Such employees should successfully pass several security checks before being employed by the organization. Also, such employees have a direct contract with the organization and the organization trusts them to a certain level (e.g. based on prior experience). In cloud computing context the problem is of much more worries and has greater impact at organizations for following reasons: (a.) insiders' domain has expanded from organization internal employees and contractors to organization internal employees and contractors, cloud internal employees and contractors, cloud customers, and cloud third party suppliers; (b.) the organization does not have a direct relation with cloud employees and cannot anticipate their trust level; (c.) other cloud customers, which could be a competitor organization might share the same physical server as the organization (i.e. problems of multi-tenant architecture [RTSS09]); and (d.) cloud-of-clouds in which a cloud provider might host part of his customers data at another cloud provider (e.g. in case of major failure, increase in demand, etc) results in expanding cloud customer insiders to include the new cloud provider insiders. These increase the

exposed threats on organizations sensitive assets. Thus, the risk of insider threats when moving to cloud infrastructure is greater than the risk of insiders in the organization.

In this chapter we provide a systematic method for identifying insiders, which we use to identify insiders in home healthcare system. This chapter is organized as follows. Section 2.2 discusses insider definitions. Section 2.3 provides a set of models illustrating the relationship between actors, credentials and infrastructure in cloud computing context focusing on IaaS. It then provides a method for identifying insiders. Section 2.4 provides a home healthcare scenario, and uses the models in section 2.3 to identify insiders in home healthcare system. It then provides a threat analysis for the identified insiders in home healthcare system. Section 2.5 discusses related work, and section 2.6 concludes the chapter.

2.2 Insiders

In this chapter we build on Alawneh et. al. [AA11] work which provides a detailed analysis of insiders in organizations. In their work the authors distinguish between insiders and potential insiders and define them as follows.

> **A potential insider** is a user who is granted a credential in an authorized way to access sensitive corporate information for a specific purpose defined by the organization (does not cause harm), or a user who obtains a credential in an unauthorized way but does not use it to cause harm.

> **An insider** is an internal or external user who "uses credentials", obtained by either authorized or unauthorized means, to access sensitive corporate information that results in harm to the organization. Such a misuse could be either accidental or deliberate.

Based on the definition we conclude the following set of mandatory rules to identify an insider.

R1: The insider could be either a potential insider, as defined above, or someone who managed to obtain the potential insider credentials in some way; and

R2: Uses the credential to access a resource for a different purpose than the one which the credentials were originally granted for; and

R3: This misuse results in harm to the resource owner/manager.

In the remaining part of this chapter we use these rules to identify insiders in the cloud.

2.3 Conceptual Models

A cloud computing based system typically involves a number of actors, from different organizations, which interact with the system. In order to identify which actors are potential insiders and the threats emanating from their activities we first require identifying all actors within such a system. We then need to understand the relationships among the actors as well as their level of access to resources and assets that are part of the cloud. To this end, we build various conceptual models that illustrate the explicit relationships among the various entities and expose any implied relationships as well as interactions between the actors and the system.

2.3.1 Organisational View

A cloud computing based system for IaaS type may be designed to serve the needs of different communities including: i) users within a single organization or collaborating organizations (e.g. a private cloud within an enterprise); ii) users within a research community comprising, for example, a virtual organization; or iii) public community (e.g. a mixture of enterprise and individuals). In most cases, several organizations will be involved in a cloud based system. We develop an organizational model of such a system as shown in Figure 2.1 in which we identify a generic entity *Organisation* as the parent of any organizational entity within a cloud-based system.

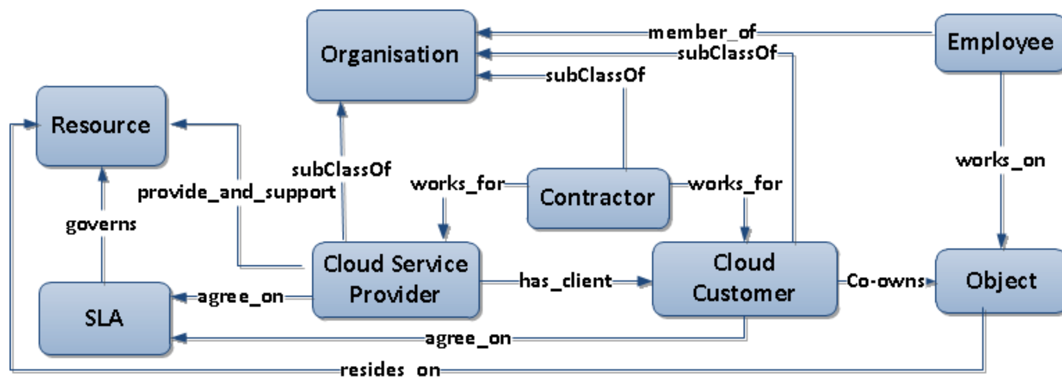


Figure 2.1: High-Level Organisational View of the Cloud

At a minimum, a cloud-based system comprises a *Cloud Service Provider*, a *Cloud Customer*, and quite often a *Contractor*, such as cleaning company or hardware suppliers, that may *work_for* either a *Cloud Service Provider* or a *Cloud Customer*. A *Cloud Customer has* some *Object*, such as computation or data, that they wish to take to the Cloud. To do this, the *Cloud Service Provider* and the *Cloud Customer agree_on* some service level agreements (*SLA*) which defines the *Resource* provided to the *Cloud Customer* and the conditions, such as performance, availability and liability, under which the resources are provided. After which the *Object* is transferred to the cloud and *reside_on* the Resource. Furthermore, an *Organisation has* one or more *Employee* who *work_on* the *Object* owned by the Cloud Customer.

2.3.2 Assets and Clients

One of the main advantages of a cloud-based system is that it can increase the availability of the *Object* to a wider audience via the Internet. It therefore becomes necessary to define the entities that may have access to an Object on the cloud, as illustrated in Figure 2.2. From the organization perspective, an *Object* can either be an *Asset* (with value to the organization) or not an asset (less valuable). These objects, especially the valuable ones, will require an *Authorisation Policy* to define a type of *Credential*, i.e. an *Authorised Credential*, that enables access to the Objects.

In some cases, the *Object* may be co-owned by the cloud customer and the cloud customers' *Client*. In such cases, the *Client* may have to *decide_on* all or part of the *Authorisation Policy* to define which other clients, and sometimes Employees of the cloud customers, have access to the *Authorised Credential*. We clarify these in context of an example when discussing the home healthcare system.

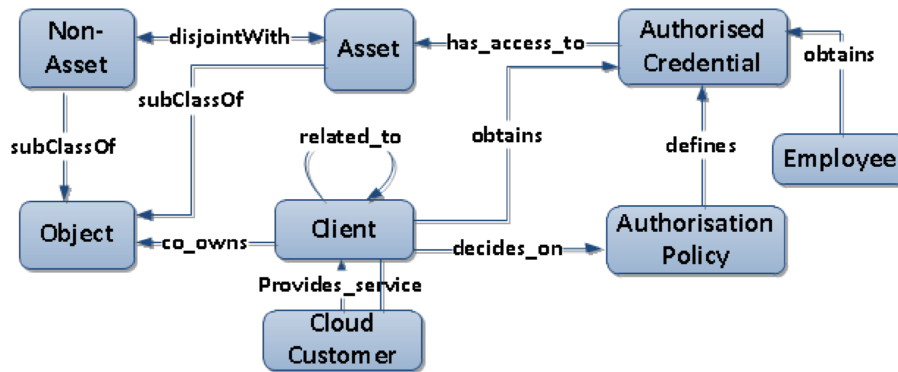


Figure 2.2: A view around the Client

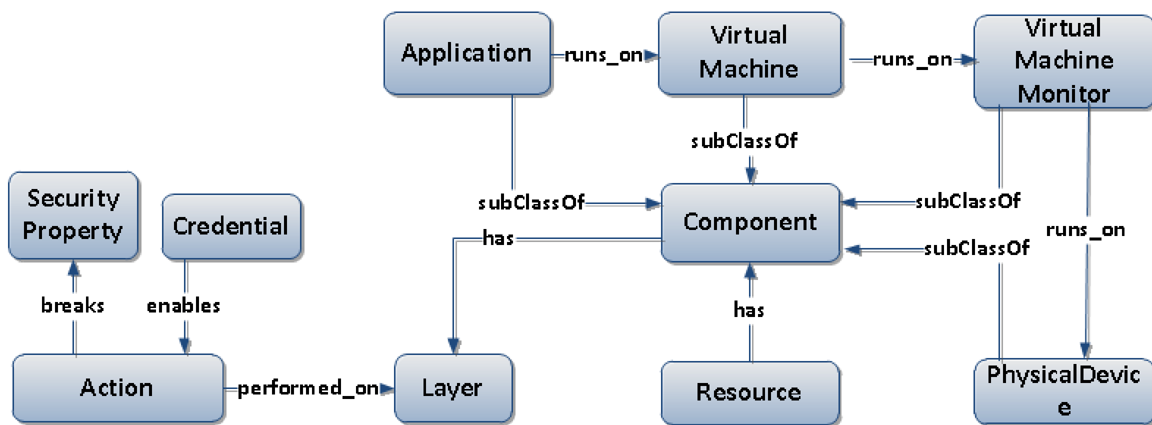


Figure 2.3: A Breakdown of a Cloud Resource

2.3.3 Infrastructure Model

We now develop a conceptual model of a *Resource*, as shown in Figure 2.3, within a cloud-based system to aid with identifying the interactions. In this model, a *Resource* is a composition of components including *Physical Device*, *Virtual Machine Monitor* (VMM), *Virtual Machines* (VMs), and *Applications*. A component may further be divided into layers[1] which indicate the parts of a component that interact to provide the functionality of the component. VMM runs on top of the *Physical Device* to enable one or more VMs to run on the *Physical Device* (based on the *Physical Device*'s layer, as explained in section 2.4.3). *Applications* are configured to run in VMs.

We define an *Action* as an event performed by the user of the resource. An *Action* is *performed_on* a layer, and may *break* zero or more *Security Property* and require some form of *Credential* in order to be performed.

2.3.4 Procedure for Identifying Potential Insiders and Insiders

The conceptual models defined above are used for identifying potential insiders and insiders that may exist within a given context. This is achieved by instantiating the models given above with the actual descriptions of entities that exist within the context. More specifically, one has to provide: i) *layers of each of the components*; ii) *actions that may be performed at each layer* (these should be limited to those that may have effects on one or more security properties);

[1] see [Abb11] for detailed discussion about cloud taxonomy and layering concept.

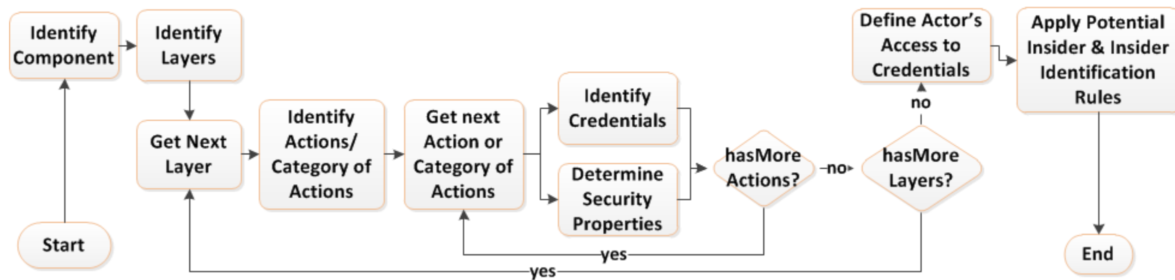


Figure 2.4: Potential Insiders and Insiders Identification Process

iii) *credentials that may be used within the system*, and iv) *actors that may have access to the identified credentials*.

The identification process, illustrated in Figure 2.4, involves mapping actions or categories of actions to layers on which a particular action or category of actions can be performed. Then for each identified action or category of actions, determine the security properties that it may break and the credentials required to enable the action. With the credentials identified, identify actors that may have access to each of the identified credential. Then potential insiders and insiders are those actors that satisfy the criteria as defined in Section 2.2. This process enables us to make it explicit the means through which potential insiders and insiders are defined. We use this method to identify potential potential insiders and insiders in home healthcare system.

2.4 Insiders Analysis for Home Healthcare

2.4.1 Scenario

In this section we describe a scenario for using home healthcare system in cloud computing. This scenario is the base of our insider discussion in the remaining part of this chapter. In this scenario, a hospital provides home services to clients, which are accessible through web portals that are provided through the hospital's website. These services are hosted on a cloud infrastructure and using Infrastructure as a Service (IaaS) model. Users should not need to be aware about the existence of clouds, as all technicalities must be transparent to them. Users might include patients, a patient family member (a care giver), hospital staff (e.g. general practitioner, medical consultant, psychiatrist), and other collaborating organizations with the hospital (e.g. research center).

The system administrators at cloud infrastructure provider allocate virtual resources and manage them based on a pre-agreed Service Level Agreement (SLA) with the hospital. This includes allocating VMs, virtual storage, networking and managing them. The hospital, on the other hand, is in charge of installing and maintaining the operating system and all software packages, which are needed to run the hospital application. For example, the hospital is in charge of maintaining the operating system, database management system, application servers, and developing and deploying the hospital application. The hospital can outsource this service to a professional IT services company, or can have its own IT staff to maintain the infrastructure provided by the cloud provider. Once the hospital application is deployed on the cloud, the hospital services are then made available to clients through the web. Clients should not notice the existence of the cloud, as they accessing the application by connecting to a URL provided by the hospital. The clients will then use the credentials provided by the hospital to login and access the allocated services.

The cloud service provider can have SLAs with other third party service providers (e.g. hardware suppliers and operating system vendors) to act as an escalation point for critical failures and to provide additional support for services that are not in house.

One of the cloud characteristics is supporting a multi-tenant architecture. By this the cloud infrastructure is shared by all cloud customers (i.e. organizations for IaaS type). For example, the virtual resources used by an organization might share the same physical servers as a competing organization. Also, cloud providers themselves could collaborate to provide better services to their clients. For example, hospital data can be replicated across several jurisdictions, which provide higher resilience, faster access, and load balancing.

2.4.2 Model Instance

Based on the scenario description above, we can instantiate the model as shown in Figure 2.5. The *Hospital* is an organization that needs cloud computing resources and therefore will be the *Cloud Customer*. A Hospital has a number of employees including: a *Researcher*, *Hospital System Administrator*, and a *Psychiatrist* who *work_on PatientRecord* (a type of asset for the Hospital).

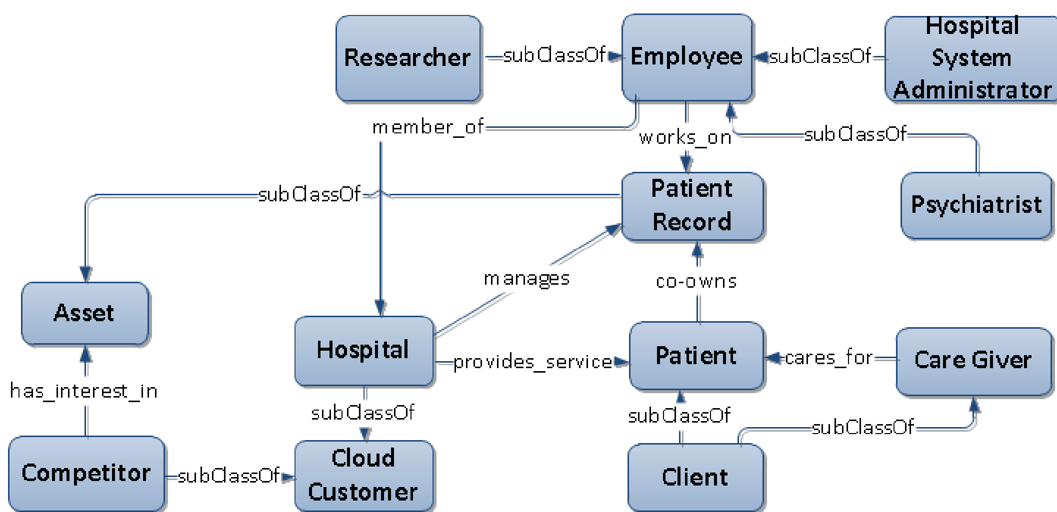


Figure 2.5: Model Instance for the Home Healthcare Scenario

The Hospital provides healthcare services to its Patients, (i.e Clients) who co-own the Patient record and are cared_for (a sub-relation of *Client related_to Client* in Figure 2.2) by their Care Givers. Hospitals may co-exist in the cloud with other organizations (which we label as Competitor) that may be interested in the Hospital's Assets.

2.4.3 Identifying Potential Insiders and Insiders

In this subsection we identify potential insiders and insiders using the process outlined in Section 2.3.4. The process starts from the identified system components given in Section 2.3.3; i.e. Physical Device, VMM, VM, and Application. In this section we identify potential insiders and insiders in each of these components, as follows.

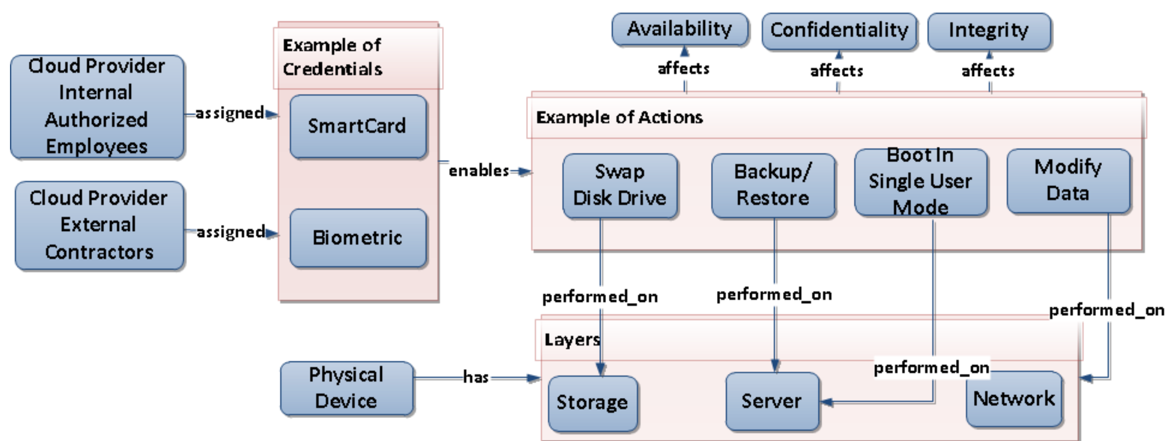


Figure 2.6: Physical Infrastructure

Physical Device

A Physical Device can belong to three layers: Storage, Network and Server, as illustrated in Figure 2.6. At the storage layer, the Physical Device type would be a storage device, which is vulnerable to different types of threats, for example: (a.) it can be swapped with a corrupt device; (b.) it can be taken away and mounted in another system, and (c.) the device's content can be copied or altered. Based on the attack scenario these threats could have an impact on content confidentiality, integrity and/or availability. For example, content may lose integrity through backup/restore operations, content may lose availability through removing device's content, and content could be leaked by copying it to a USB memory stick.

At the Server layer, the physical device type would be a physical server, which is vulnerable to different types of threats, for example: (a.) The physical server is vulnerable to all possible hardware threats, (a.) the physical server can be started in a different configuration from that expected, for example by booting in single user mode. These could affect content availability, integrity, and confidentiality based on the attack scenario. For example, booting the server in a single user mode enables attackers to access the superuser account without the need to possess authorization credentials.

At the network layer, the physical device type would be a network component. Data can be modified as it is transmitted to/from the device affecting availability, confidentiality as well as integrity.

Physical devices will normally be stored in data centers that have access restrictions to few individuals. This access is typically enforced using credentials such as Biometrics and Smartcards. We note such type of credential will be assigned to authorized employees from cloud service providers (e.g. system administrators) and employees from organizations contracted by the cloud service provider.

Based on our insider and potential insider definition cloud authorized employees and contractors are potential insiders as they are provided with credentials that can access Physical devices. Once the potential insiders use the credentials and cause harm, then they are insiders. Also, anyone who has access to these credentials (by stealing it or the system administrator himself shares it with unauthorised person) is considered an insider once he uses them and caused harm.

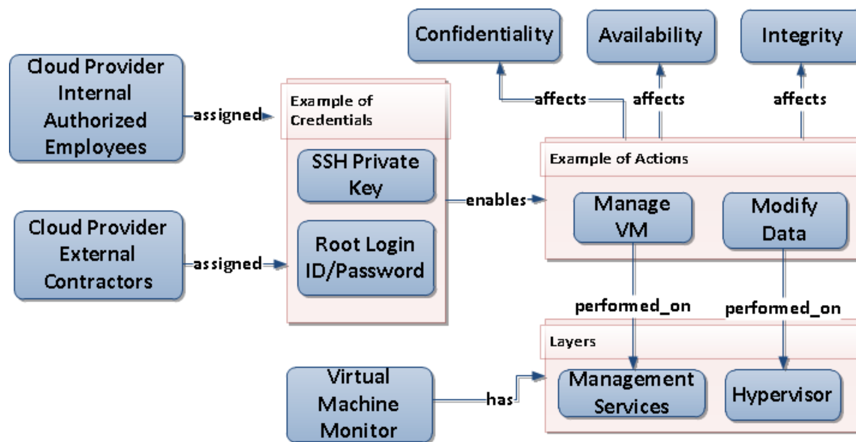


Figure 2.7: VMM Component Breakdown

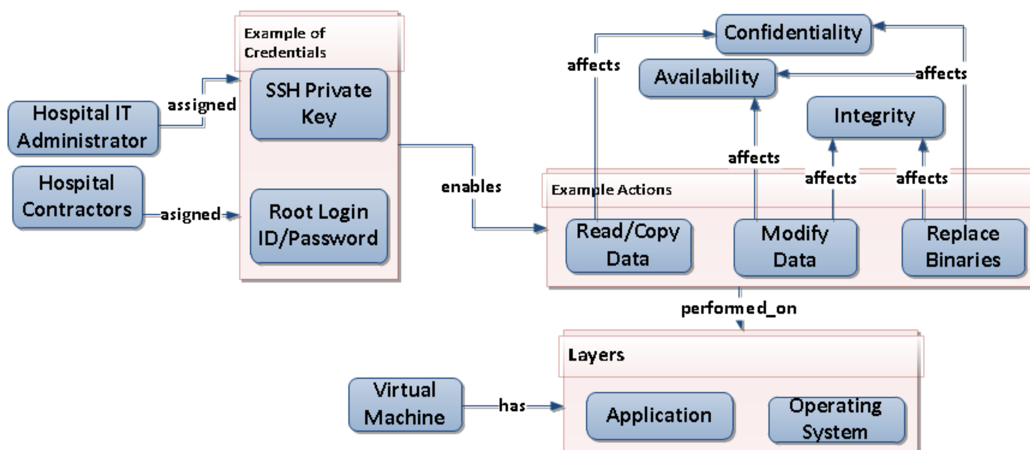


Figure 2.8: VM access

Virtual Machine Monitor

A VMM controls the VMs running on the physical device. It comprises of a hypervisor (a thin layer kernel) and management services, as illustrated in Figure 2.7. The management services enable VMs' management actions such as start, stop and migrate, to be performed. Because network traffic to and from the VMs is mediated by the VMM, data for the VMs can also be modified through the VMM. All these actions may impact availability, integrity and confidentiality of the services offered by the Hospital. The typical credentials that enable accessing the VMM to perform such actions include root login credentials and SSH private keys. *Cloud Povider authorized employees (e.g. system administrators) and contractors* are the main actors that are expected to be assigned these credentials.

Based on our insider and potential insider definition cloud authorized employees and contractors are potential insiders as they are provided with credentials that can access VMM. Once the potential insiders use the credentials and cause harm, then they are insiders. Also, anyone who has access to these credentials (by stealing it or the system administrator himself share it with unauthorised person) is considered an insider once he has used them and caused harm.

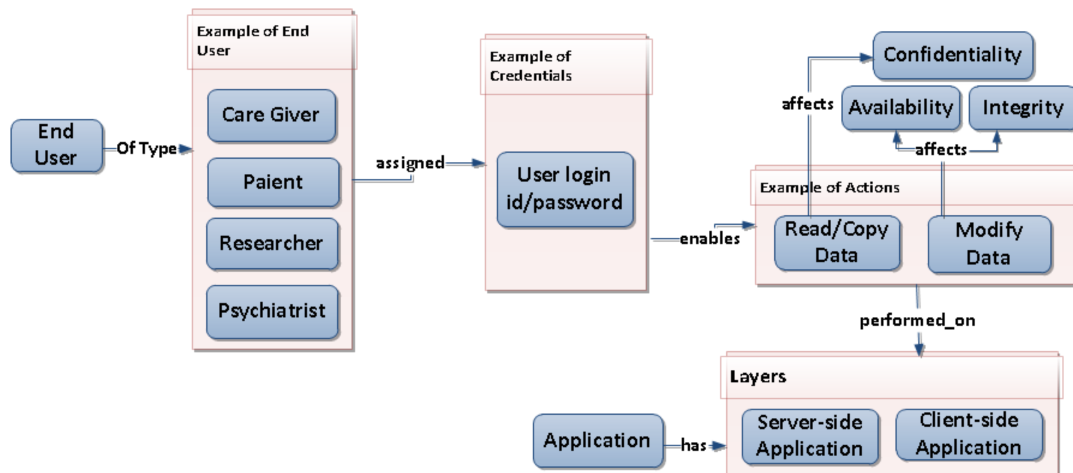


Figure 2.9: Application Access

Virtual Machine

VMs are containers that comprise an operating system and applications. These are stored together with configuration information in a disk image. Figure 2.8 shows examples of actions that may be performed on any of the layers, which could affect all three security properties (i.e. availability, confidentiality and integrity). For example, updating binaries can be performed on the operating system and application, which can affect the three security properties. The entire disk can also be copied affecting confidentiality of data stored in it or the data may be modified affecting availability and integrity.

Two types of credentials would typically be needed to perform the identified actions: the SSH private key and the root login id/password. These would enable all the actions identified at all layers. *Hospital internal system administrators and contractors* working on behalf of the hospital would be the main actors expected to be assigned root login and ssh private keys. However, system administrators from cloud service providers for IaaS type should not normally get root access to the VMs.

Based on our insider definition hospital cloud internal system administrators and contractors could be potential insiders as (a.) they are provided with credentials that can access the main patient information repository from server-side application. Also, anyone who has access to a system administrator's authorised authentication credential is considered a potential insider (by stealing it or the system administrator himself shares it with an unauthorised person).

Based on our insider and potential insider definition hospital internal system administrators and contractors are potential insiders as they are provided with credentials that can access VMs. Once the potential insiders use the credentials and cause harm, then they are insiders. Also, anyone who has access to these credentials (by stealing it or the system administrator himself share it with unauthorised person) is considered an insider once he has used them and caused harm.

Application

Applications run on VMs and can be either client-side application or server-side application, as illustrated in Figure 2.9. These are stored and run on VMs. Figure 2.8 shows examples of actions that may be performed on any of the layers, which could affect all three security properties (i.e. availability, confidentiality and integrity). For example, modifying data can be performed from

client-side or server-side application, and it can also affect the three security properties. Content stored in a server-side application can be copied affecting data confidentiality, altered affecting data integrity, or removed affecting data availability.

End Users would be assigned user logins allowing them to perform actions enabled by this credential. Example of such users include patients, care givers, and hospital employees (e.g. researchers and psychiatrists).

Based on our definition end-users could be potential insiders as (a.) they can access data using authorised credentials. Also, anyone who has access to an authorised authentication credential (by stealing it or the authorised user himself share it with unauthorised person) is considered a potential insider.

Based on our insider and potential insider definition end-users are potential insiders as they are provided with credentials that can access the application's content. Once the potential insiders use the credentials and cause harm, then they are insiders. Also, anyone who has access to these credentials (by stealing it or the end-user himself shares it with unauthorised person) is considered an insider once he uses them and caused harm.

2.4.4 Insider Threat Analysis

Insiders' actions could affect information/service availability, integrity, and confidentiality. Schemes that are proposed to address insider threats are for organization and to the best of our knowledge insiders in cloud computing environment has not been tackled before. Also, as we discuss in section 2.5, insider schemes proposed for organizations mainly focus on mitigating insider threats for content confidentiality. In our opinion this is due to two main reasons: (a.) the lack of solid cases discussing the insider threats, and (b) due to the nature of the problem which is not easy to address as insiders are authorised to update/remove records. We now discuss the possible threats that can be raised by the identified insiders in section 2.4.3.

1. *End Users* — access the hospital application services via a provided authentication credential. Each user is assigned a credential with access rights for accessing the provided services, which enables a user to create new records, update patient records, and delete patient records. Such rights should not provide the user the ability to access the system from backend (i.e. from operating system level or database management system level), and they should not provide users with the ability to have a global effect on services (e.g. stop a service or remove the whole data repository).

   End users' insider threats are restricted to the granted access rights that are provided to the end user credential. For example, if access rights allow the user to only read a patient record, then the insider threat affects content confidentiality. If access rights allow the user to update and delete patient record, then the insider threat affects content integrity and availability, and so on.

2. *Hospital internal system administrators* — access the hospital application and backend virtual resources via a provided authentication credential(s). A system administrator is in charge of maintaining the application and backend services (e.g. operating system and database management system). System administrators are assigned access rights for performing their job, which could enable them to do critical actions on the system (e.g. suspending a VM, backup/restore operations, migrating VMs, and stopping/restarting middle tier application servers). Such rights enable its holder to have a global effect on

provided hospital services (e.g. stop a service, remove the whole data repository, and leaking the data repository for patient records).

Insider risks in the above case would be based not only to the access rights that are provided to the account that is used by the insider but it also would be based on the used security best practices (e.g. separation of duty and least privilege concepts). For example, an organization might reduce the impact of data integrity by introducing database/application backup role, which is separate from system administrator role. Also, an organization can introduce an application maintenance role that is separate from database management role. Application of security best practice does not necessarily prevent insider's threats but it will lessen their effects. We now list the main insider threats for system administrator role.

(a) *Availability* — an insider can affect system availability. For example, application management role can stop/delete middle tier application services, database management role can stop/delete the database, and operating system role can stop the virtual resources. All these are examples of how an insider can cause a global effect on service availability.

(b) *Integrity* — an insider can affect system integrity. For example, application management role can create an authorized user account for a non existing general practitioner, update patient records, and then delete the account. A backup role can invalidate the backup. A database management role can update patient records directly from the database.

(c) *Confidentiality* — an insider can leak sensitive content to unauthorised parties. For example, a backup role grantee can copy the backup to a memory stick, restore it at home and then leak the content to others. A database management role can also copy the database to a memory stick or even searches and then extract selected patients records to a USB stick or leak them via email.

3. *Hospital contractors* — are provided with appropriate credentials enabling them to maintain part of the hospital provided services (e.g. application support, operating system, and database management system). Contractors should be assigned the minimal access rights that are sufficient enough to do the job. Such rights could enable them to do critical actions on the system, exactly as the one described for the system administrator role. Insider threats caused by external contractors cloud have the same severity level as the one caused by internal system administrators. Identifying these would be based on the roles granted to the contractor.

4. *Cloud provider internal employees* — those have full access to the physical hardware resources (servers, storage and network devices) and the operating system (hypervisor), which serves the provided virtual resources. In addition, they have full access to the cloud infrastructure management software packages. These are used to maintain and monitor the virtual resources, e.g. stop, start, suspend, resume, migrate and backup a VM, and allocate/revoke computational resources to/from a VM.

   The insider threats caused by the insiders of a cloud provider could have greater effect than the hospital insiders. This is because insiders could have even more authoritative access to the underlying infrastructure. Also, they are the party who manages the hospital allocated virtual resources. Following we briefly outline these threats.

(a) Insider threats that affect content availability. An insider who is granted a virtual resource management role can deprive some of the computational resources that are granted to VM, which cause the machine to be non-responsive, for example, in peak periods.

(b) Insider threats that affect content integrity. An insider who is granted access to the hypervisor layer as a superuser can access the VM running on the hypervisor enabling the insider to update the VM content. Also, an insider can restore VM storage from an old/hacked backup.

(c) Insider threats that affect content confidentiality. An insider with proper access privileges can copy a VM image or a backup from the storage server and restore these at home, which enable the insider to leak the hospital patient information to unauthorised parties.

5. *Cloud Provider external contractors* — are provided with appropriate credentials enabling them to maintain part of the cloud infrastructure (e.g. hardware suppliers and software application support). Contractors should be assigned the minimal access rights that are sufficient enough to do the job. Such rights could enable them to do critical actions on the system, exactly as the one described for the cloud internal employees. For example, a contractor that maintains the storage can perform backup of the storage and restore it at home, which enables him to leak sensitive content. Insider threats caused by external contractors cloud have the same severity level as the one caused by cloud internal employees. Identifying these would be based on the roles granted to the contractor.

6. *cloud-of-clouds internal employees* — as discussed before if two cloud providers collaborate, one cloud internal employee could access another cloud data that migrates across to their internal infrastructure. In this case the destination cloud provider's system internal employees can cause the same level of threats "on the migrated data" as the source cloud provider internal employees, as discussed in the previous point.

7. *Cloud provider customers* — in a multitenant architecture [RTSS09] organizations share the same hardware resources. In this all employees of an organization who are authorized to access their organizational resources in the cloud might be an insider for other organizations sharing the same hardware resources. For example, an attacker can learn sensitive information about other organizations (e.g. by exploiting covert channels [LBOR09, OBRL09]).

2.5 Related Work

Up to our knowledge our research is the first to analyse the insider threats in a cloud computing environment. In this section we discuss related research work in insider threats in other domains.

Dynamic domain schemes [AA08b, AA08c, AA08a] mitigate insider threats on content confidentiality for organizations. This is by using the domain concept; mainly, it moves the fundamental access control assumption that "authorized users are trusted (or should be trusted — not necessarily trustworthy and cannot measure their trustworthiness" to "the need to trust authorized devices whose trustworthiness can be measured and attested". In Enterprise Rights Management (ERM) schemes (see, for example, [MC05, Ora08]) content can be created, stored and exchanged between client devices. ERM does not address the insider threats. For example

content leakage can be realized in ERM by authorized users sharing their credentials. In a typical enterprise organization, users have a degree of freedom. Users may choose to exploit their access privileges, for example by revealing content or sharing credentials used to access content. If users exploit their access privileges then the threat of content leakage can be realized.

2.6 Conclusion

In this chapter we present a set of conceptual models, which help in identifying insiders and potential insiders in a cloud computing environment. We use the insiders and potential insiders definition in [AA11] and identified a set of rules for distinguishing insiders and potential insiders. We then use the rules and the conceptual models to identify insiders for home healthcare system in a cloud computing environment.

Chapter 3

Cryptography-as-a-Service

Chapter Authors:
Sven Bugiel, Stefan Nürnberger, Hugo Ideler, Ahmad-Reza Sadeghi (TUDA)
Sören Bleikertz (IBM)
Mina Deng (PHI)

3.1 Introduction

Cloud computing offers IT resources, including storage, networking, and computing platforms, on an on-demand and pay-as-you-go basis. This promise of operational and monetary benefits has already encouraged various kinds of organizations to shift from a *"classical"* data-center to a (public) cloud-based service deployment of their workloads. In particular, web-services profit economically from a cloud-based deployment [CS11].

However, moving to the cloud also means to relinquish physical control over the own data and computations. Instead, clients confide them to the cloud service provider of their choice. To protect their data and computations from attackers, clients can deploy cryptographic security mechanisms, which usually require the deployment of high value cryptographic credentials such as secret keys. For instance, clients may deploy a web-service that uses SSL/TLS to secure the communication with its end-users requiring an asymmetric key-pair or encrypt data written to (cloud) storage using an encrypted file-system with a symmetric encryption key. However, as a consequence of the loss of physical control, the client faces limitations concerning the (often debated) security in cloud computing [Clo10] and in particular regarding the protection of deployed high-value cryptographic credentials. In the classical deployment model of dedicated data-centers, clients had the option to incorporate their own hardware security devices like Hardware Security Modules (HSMs) or SmartCards into their infrastructure in order to protect their cryptographic credentials and operations from attackers compromising their (virtual) servers. In contrast, today this option is no longer available in the cloud since cloud providers strictly prohibit physical customizations and access to their facilities, thus making the deployment of additional external hardware in the cloud infeasible. Hence, it is desirable to have security mechanisms in the cloud that protect cryptographic primitives and credentials very much like in the traditional model, in a practical and reasonable attacker model.

In addition, in virtualized environments, such as clouds, the management domain of the virtualization infrastructure has to be trusted. This trust comes twofold: First, past attacks ([CVEa, CVEb, CVEc]) have shown that outsider attackers may compromise this management domain and gain elevated privileges that allow them to violate the clients' security and privacy, e.g., by extracting high-value information and credentials [RC11]. Second, particular to cloud

infrastructures, the management domain is under the control of the administrative staff of the cloud provider and thus potential insider attacks on the provider's side can have the same devastating consequences as a compromised management domain.

Finally, another concern, specific to public clouds, are new usage models, such as cloud appliance stores (e.g., The CloudMarket [the]), a popular way of sharing VM images with other clients. This kind of sharing on top of the cloud infrastructure bears severe security and privacy risks. A recent analysis of the Amazon cloud appliance market yielded that 5.2% of the publicly shared images contained cryptographic keys or credentials [BNP+11] – which compromise the security of both the image consumer and image publisher.

In presence of these shortcomings and threats, in this chapter we focus on a security architecture that allows for establishing *secret-less* client VMs and separating client's cryptographic primitives and credentials into a *client-controlled* and *protected* CryptoDomain (or short `DomC`), using available technology. This enables the client to securely provision and use her own cryptographic primitives, such as virtual Hardware Security Modules (vHSM) or virtual Full Disk Encryption, and thus establishes Cryptography-as-a-Service (*CaaS*). Simultaneously, segregating cryptographic operations and keys from the vulnerable client VM and encapsulating them in an isolated domain prevents attackers from accessing the VM state (e.g., end-users compromising the VM) and extracting cryptographic keys from this state. Moreover, a trusted hypervisor can efficiently protect a separate CryptoDomain against a compromised or malicious management domain.

As we elaborate on related work in detail (Section 3.7), different proposed solutions make the case for disaggregation of the privileged (potentially malicious or compromised) management domain [MMH08, SK10], deprivileging the management domain [ZCCZ11, SJV+05], or enabling more flexible and self-managed services in the cloud [WJW12, BLCSG12], where the Self-Service cloud model [BLCSG12] is the closest related work to our architecture. In our *CaaS*, we specifically aim at providing client-controlled cryptographic operations and credentials in the cloud, secure against different insider and outsider attackers. This requires novel security extensions to the VM life cycle management to protect the CryptoDomain during storage, transit, or instantiation, and to tightly couple the CryptoDomain to corresponding client VM.

Contribution In this chapter, we present the design and implementation of Cryptography-as-a-Service (*CaaS*) based on well-established and widely available technology. Our contributions are as follows:

- *CaaS* empowers the cloud client to be in control of the cryptographic operations and keys that she deploys in the cloud, independently of the cloud provider. We introduce a dedicated, client-specific domain `DomC` for the client's cryptographic primitives and credentials.

- Clients can leverage their CryptoDomain `DomC` in two different usage-modes: a) *Virtual Security Module* and b) *Secure Virtual Device*. In case a), `DomC` emulates a virtual hardware security device, like an HSM, attached to the client VM. In case b), `DomC` forms a transparent layer between the client VM and peripheral devices (storage disk or network card), which encrypts all I/O data streams to/from those devices similar to full-disk encryption or Virtual Private Networks (VPN).

- Based on our security extensions to the hypervisor and well-established Trusted Computing technology, `DomC` can be protected from malicious insiders and outsiders in a

reasonable adversary model during its entire life-time including instantiation, migration, and suspension.

- We present the reference implementation of *CaaS* based on the Xen hypervisor and evaluate its performance for full disk encryption of attached storage and for a software-based HSM. Moreover, we present a partial setup of *CaaS* on the AWS EC2 cloud.

- We evaluate the benefits of *CaaS* at the example of the Activity 3 life medical use-case for a cloud-based home healthcare service by the consortium partners PHI and FSR.

Outline The remainder of this chapter is structured as follows. In Section 3.2 we define our system model and security requirements. We present the design and implementation of our *CaaS* in Section 3.3 and evaluate its security in Section 3.4. We present a partial setup of *CaaS* on AWS EC2 in Section 3.5 and demonstrate the benefits of our security architecture for the medical use-case in Section 3.6. We discuss related work in Section 3.7 and conclude the chapter in Section 3.8.

3.2 Model and Requirements

Our model is based on the typical *Infrastructure-as-a-Service* (IaaS) compute cloud as depicted in Figure 3.1. We focus on the popular Xen hypervisor [BDF+03].[1] Thus, in the remainder of this chapter we will stick to the Xen terminology, as we explain in the following.

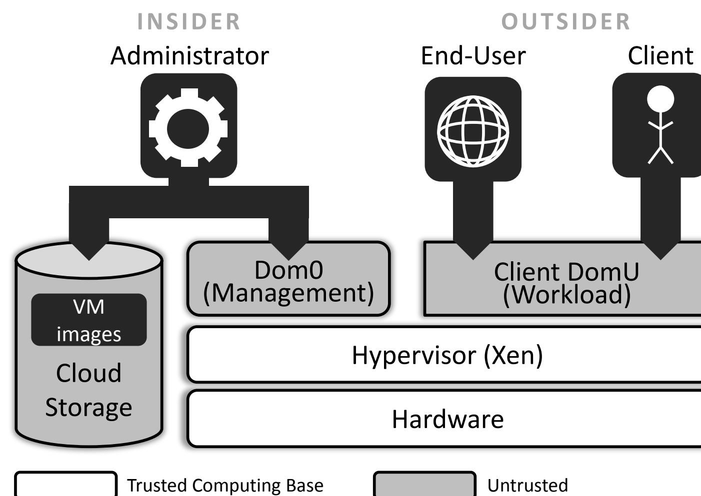


Figure 3.1: Typical IaaS cloud model including our adversary and trust model.

In IaaS clouds, *Clients* rent virtual resources from the provider and send their workloads to the cloud in form of virtual machines (VMs). Workloads can be private processes, but more commonly are public services such as web services offered to *End-Users* on the Internet. The clients' workload VM is denoted (in Xen terminology) as `DomU`, meaning *unprivileged* domains that are guests on the hypervisor and have no higher privileges such as direct hardware access.

[1] Technically the IaaS model could also be instantiated on various other virtualization solutions such as KVM [Qum06], VMWare [VMw09], or research approaches like Nova [SK10], and we stress that our architecture presented in this chapter applies to those as well.

While there can be many `DomU` executing in parallel on top of one Xen hypervisor and their number usually varies over time, there exists only one persistent *privileged* management domain, denoted `Dom0`. This domain is usually not exposed to outsiders. In contrast to other hypervisors, Xen does not configure or manage guest VMs nor does it emulate devices, but defers these tasks to the privileged domain `Dom0`, that holds the necessary rights for accessing hardware resources (e.g., memory). Thus, `Dom0` is naturally the place for the cloud infrastructure management and cloud (compute) *Administrators* to operate in.

Besides computation, IaaS clouds normally also provide *Cloud Storage*. While clients leverage this storage for their workload data, this storage is also used to save the *VM images*, i.e., binary representations of VM states, from which `DomU` are instantiated. In newer cloud usage models like cloud app stores [BNP+11], clients are also able to publicly provide their VM images and share them with other clients. Cloud storage is also maintained by cloud (storage) administrators.

3.2.1 Trust and Adversary Model

From the perspective of clients, one of the most debated issues in cloud computing security is the trust in the cloud provider. In order to build a reasonable and practical trust model we do not assume a fully untrusted provider, but rather consider the specific involved *cloud internal employees* (cf. subsection 2.4.4 and possible attacker types on the provider's side. Other security issues concern the clients offering cloud-based services and the users of these services. We categorize these cloud internal employees in different administrators:

Compute Administrator On a commodity hypervisor, administrators have read/write access to the memory of a running VM, e.g., for VM introspection. Hence, they are able to write data and therefore inject arbitrary code in the client's domain or extract sensitive information from the state of a running VM [RC11]. We only consider attacks from administrators with *logical* access to the physical servers, e.g., by operating in the privileged management domain `Dom0`, and *not* attackers with physical access. This attacker model stems from practical scenarios, where datacenters are operated by a small team of trusted administrators with physical access and a large number of administrators with logical access, often outsourced and provided by third parties with limited trust.[2]

Storage Administrator For administrators of storage resources, we consider both passive and active adversaries. A passive attacker aims at learning cryptographic keys stored in a VM image whereas an active attacker aims at modifying the VM image, e.g., by injecting malicious code into the image that will extract cryptographic keys of that instance at run-time. For storage administrators we allow physical access to hardware.

Network Administrator We model the network administrators (omitted in Figure 3.1) according to the Dolev-Yao [DY83] attacker, i.e., the attacker has full control of the network and can eavesdrop on and tamper with network traffic in between the nodes in the cloud and between the cloud and its clients.

[2] Note that purely cryptographic approaches such as secure multiparty computation ([BDNP08, BLW08]) or Fully Homomorphic Encryption [Gen09] allow operation on encrypted data and can be deployed in fully untrusted settings. However, they are still in their infancy and impractical due to their enormous complexity overhead. Besides the cryptographic solutions are not sufficient in a multi-tenant and large scale computing environment like clouds (see also [VDJ10]).

Clients Clients have in general the incentive to protect security sensitive information of their workloads against a malicious or compromised Dom0 and other (malicious) clients. Moreover, if their workload is a public service, an additional protection against malicious outsiders using the service is required.

Malicious clients, on the other hand, may extract security critical information, such as cryptographic keys, stored (and, for instance, forgotten) in public, shared VM images of other clients [BNP+11]. Based on these information, a variety of privacy and security compromising attacks against the publisher of a VM image or other consumers of an image is feasible. Moreover, they may also exploit side-channel attacks to extract cryptographic keys from VMs running on the same physical resources (cf. [RTSS09]). Side-channels are crucial and very challenging to defeat. We will discuss possible solutions against this type of adversary in Section 3.4.

Users We consider malicious end-users, who consume services provided by the VMs of cloud clients. This attacker aims at compromising a VM, for instance due to some vulnerabilities in the provided services, in order to extract security critical information stored or processed in that VM.

Hypervisor We exclude run-time attacks on the hypervisor compromising its integrity during operation as this is an open research problem and out of scope of this work. Under this assumption, we consider a trustworthy hypervisor in the sense that the client can deploy mechanisms to verify the trustworthiness of the code a hypervisor is running, i.e., whether it complies with a certain trust policy of the client. This is accomplished using standardized trusted computing mechanisms such as *authenticated boot* and *remote attestation* [Tru08], as we discuss in Section 3.3.

Denial-of-Service Attack We exclude Denial-of-Service attacks from our model. This is motivated by the fact that the privileged domain Dom0, although not trusted, cannot be completely excluded from all operational and management tasks, and thus is always able to block correct operation.

3.2.2 Objectives and Requirements

Our main security objective is the protection of the client's cryptographic keys and operations in the cloud abstractly much like software versions of Hardware Security Modules (HSMs) or SmartCards. Our goal is to address the following questions:

- How can cloud clients efficiently separate and protect their security sensitive cryptographic operations from their workload VMs in the cloud? And how can those be additionally protected from malicious management domains and cloud administrators?
- How can the corresponding solution be applied to real-life public clouds using existing technologies? And, how can legacy compliance with today's client VMs be preserved?

We consider the following main security requirements to ensure the secure storage and usage of cryptographic operations and credentials in the client's virtual machine in our adversary model:

- Protection of long-term secrets of client VMs at runtime, i.e., an attacker who compromised the workload VM DomU or a malicious/compromised management domain Dom0 cannot extract this information from the DomU VM state.

- Protection of long-term secrets of the client `DomU` VM and its integrity at rest, i.e., the client's `DomU` VM image must be protected such that an attacker can neither extract credentials from it nor tamper with it.

- Secure VM management operations, i.e., suspension and migration of the client `DomU` VM must preserve the integrity and confidentiality of `DomU`'s state on the source and target platform as well as during transit/storage.

3.3 Design and Implementation

In this section we introduce the architecture and design decisions of our Cryptography-as-a-Service (*CaaS*). We first explain in Section 3.3.1 how our architecture provides a client-controlled CryptoDomain `DomC` in the cloud to protect the client's credentials from an external attacker. Afterwards, we elaborate in Section 3.3.2 in more detail on our security extensions to the hypervisor to also protect those credentials against the cloud administrative domain `Dom0` (see Section 3.2.1).

We assume the availability of a hardware trust anchor on the cloud nodes, which can be used to securely attest the node's platform state. The most widespread trust anchor of this kind is the Trusted Platform Module (TPM) [Tru08].

For brevity, the following descriptions involve only one cloud client, however, we stress that the presented solutions can be easily applied to multiple client scenarios as well. Moreover, we misuse the term *encryption* to abstractly describe a cryptographic mechanism for both confidentiality *and* integrity protection, i.e., authenticated encryption.

3.3.1 Client-controlled CryptoDomain DomC

Idea and Entities

Figure 3.2 illustrates the *CaaS* architecture in the default IaaS cloud model (cf. Section 3.2) for Xen-based virtualization. The idea of *CaaS* is to separate client's security sensitive operations and data (like cryptographic computations and keys) from the client's workload VM `DomU` and move them into a *client-controlled* secure environment denoted CryptoDomain or short `DomC` (in accordance to Xen's terminology). Thus, achieving secret-less `DomU` in the cloud.

Our `DomC` is a separate `DomU` and based on the concept of *Stub Domains* [Thi10], which are `DomU` templates with a minimal code base, implementing only the necessary (small) software stack to operate on top of the Xen hypervisor (*MiniOS*). We extend this stack to provide the `DomC` functionality, e.g., a library of cryptographic functions exposed via an interface to its workload VM. We base our design on the decision that the client should be able to provision her own image for her `DomC`. Alternatively, the functionality of `DomC` could be static (e.g., pre-installed by the cloud provider) and the client only securely injects her key material into her designated `DomC`.[3] The `DomC` image is provided in protected form by the client to the cloud, i.e., encrypted and authenticated. While this decision affects our protocols and technical solution only minimally (cf. Section 3.3.2), it greatly improves the client's trust into `DomC`, since she is in control over the `DomC` code base. Further, in contrast to the workload VM `DomU`, neither the client nor compute administrators can *directly* influence the life-cycle of `DomC` (e.g., start or

[3] This approach is useful, when additional considering mechanisms to mitigate side-channel attacks (see Section 3.4).

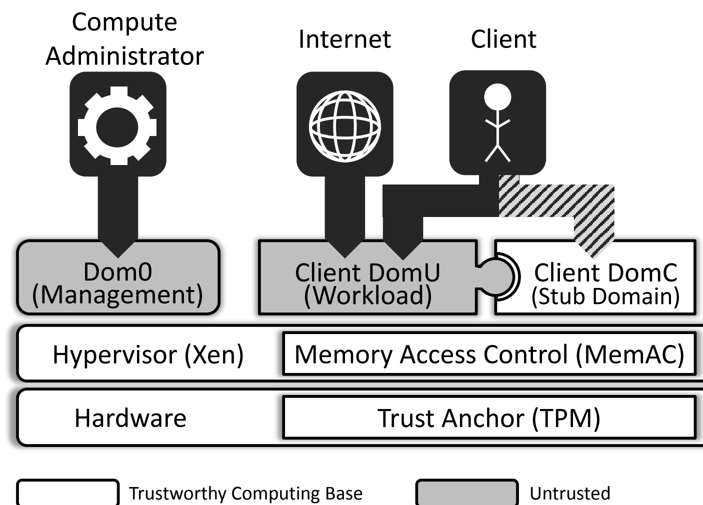


Figure 3.2: Basic idea of *CaaS*: Establishment of a separate, coupled security-domain, denoted as DomC, for critical cryptographic operations.

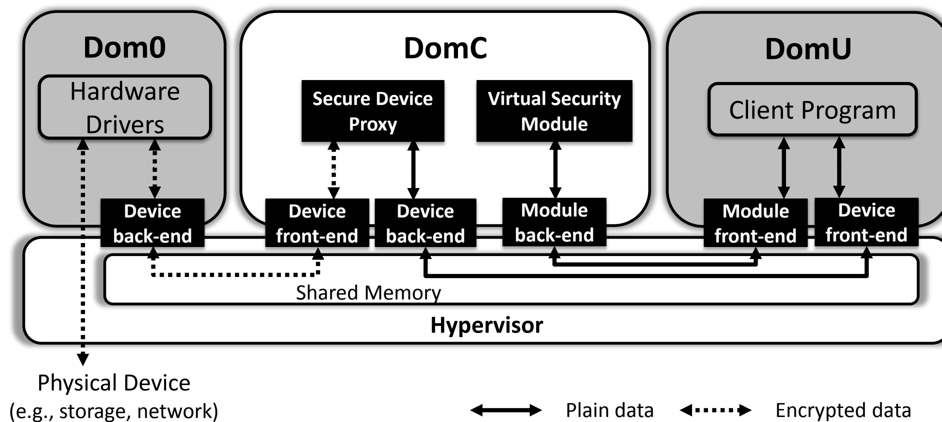


Figure 3.3: DomC usage modes: DomU can use DomC either as *Virtual Security Module* (e.g., HSM) or to transparently encrypt its storage or network data as a *Secure Virtual Device*.

stop). Instead, we leverage our extensions to the Xen hypervisor to establish a tight coupling between DomC and its DomU life-cycle and to simultaneously guarantee the confidentiality and integrity of the DomC VM. We achieve this by a new hypercall and memory access control (MemAC), as explained in detail in Section 3.3.2.

Usage Modes of DomC

Xen uses a *split driver* model for device drivers. It allows composing device drivers from a front-end and a back-end component/module (cf. Figure 3.3). The latter controls the actual physical device and is usually located in Dom0, while the former provides a virtualized representation of the physical device within guest domains. These two components are connected over shared memory and establish an inter-domain communication channel between Dom0 and the guest domain. Although the driver back-end component is usually (for convenience reasons) located in Dom0, the involvement of Dom0 is actually not mandatory for inter-domain communication.[4]

[4]A good example are Xen *driver domains*, where hardware access occurs from DomU over a driver domain, without Dom0 involvement.

Instead, the back-end component could be located in any other guest domain, which hence directly receives the data from the front-end component.

In *CaaS*, we leverage this split-driver mechanism to connect `DomC` as a Xen virtual device to `DomU`. Figure 3.3 shows the two modi operandi of `DomC` that we describe below: *Virtual Security Module* and *Secure Device Proxy*.

Virtual Security Module In this mode of operation, `DomC` resembles a (passive) security module such as a Hardware Security Module (HSM), Trusted Platform Module (TPM), or a SmartCard. In this mode, `DomU` has to be aware of the `DomC` so that it can use its interface for outsourcing traditional crypto operations like an SSL/TLS wrapper for a web service running in the VM, decrypting external storage the VM uses or to authenticate data. For instance, in our prototypical implementation `DomC` emulates a HSM or SmartCard and provides at its front-end a standardized PKCS#11-compliant interface, which makes it compatible to legacy applications running within `DomU`.

Secure Device Proxy In this mode, `DomC` acts as a transparent layer between `DomU` and external devices, such as attached storage medium or network card. This layer can be used, for instance, to transparently encrypt/decrypt the I/O data streams between `DomU` and these resources. We use this layer as a convenient building block for advanced applications such as booting fully encrypted VM images (cf. Section 3.3.2). A further application is, for instance, Trusted Virtual Domains [GJP+05] in clouds.

In the *Secure Device Proxy* mode, we chain two front-end-back-end communication channels. The first channel exists between `DomC` and `Dom0` where `DomC` connects to a device offered by `Dom0` (e.g., storage or network). The second channel exists between `DomC` and `DomU`, where `DomC` provides an identical device interface to `DomU`. These channels are connected by forwarding all I/O streams from `DomU` to `Dom0`, thus positioning `DomC` as a proxy for the offered physical device. However, `DomC` encrypts and decrypts on-the-fly all data in this stream. Although it is technically feasible that `DomC` writes directly to the physical device, routing encrypted I/O streams through `Dom0` avoids implementing (redundantly) device drivers in each `DomC`.

These modes are not mutually exclusive. A transparent encryption layer can be used while `DomU` is yet aware of the `DomC` and additionally uses it for explicit cryptographic operations.

3.3.2 Security Extensions to the Xen Hypervisor

While the usage modes of `DomC` described in Section 3.3.1 technically do not require any changes to the Xen hypervisor and thus can be instantiated on default deployments such as public clouds (cf. Section 3.5), they do not prevent a potentially malicious `Dom0` from compromising `DomC`.

In this section, we elaborate on our security extensions to the Xen hypervisor to protect `DomC` from malicious cloud administrators, i.e., compute (with logical access), network, and storage (see Section 3.2.1). In particular, our extensions comprise

- an additional Mandatory Access Control on low-level resources such as memory
- modifications to the VM launch process including a new hypercall INSTANTIATE_DOMC.

The former extension achieves a logical isolation of the client's `DomC` from any other domain including `Dom0`, while the latter one is required to protect `DomC` and `DomU` during domain life-cycle operations such as start, stop, suspension, or migration.

Using standardized Trusted Computing technology based on the TPM, we establish the necessary trust between the client and our extended hypervisor.

Logical Isolation of DomC

Figure 3.4 illustrates Xen's mechanisms to establish inter-domain memory access and our modifications to enforce domain isolation in *CaaS*. By default, Xen supports two different mechanisms to establish shared memory between two different domains (including `Dom0`): *Grant Tables* (**A**) and *Privileged Domains* (**B**). Additionally, *Direct Memory Access* (DMA) enabled devices are able to read memory of different domains (**C**).

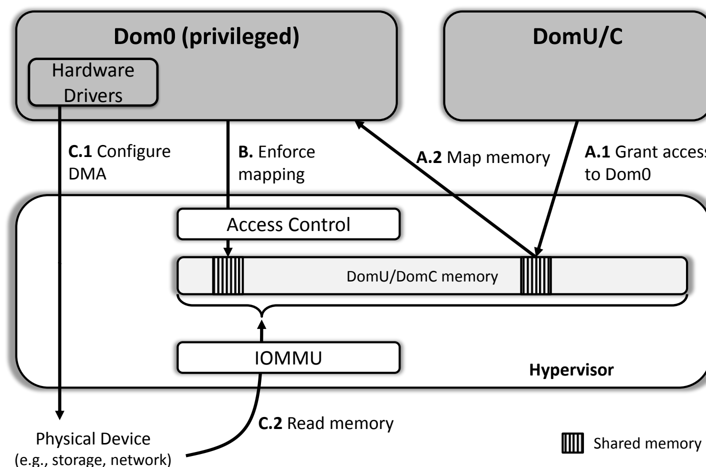


Figure 3.4: Additional memory access control in Xen hypervisor on inter-domain (shared) memory access.

Grant tables Grant (memory) tables are the default mechanism to establish shared memory pages between different domains, e.g., to realize the split driver concept. It relies on a discretionary access control approach, where the domain owning a memory page can explicitly grant access to other domains access to this page by creating an entry in Xen's grant tables. By default, domains do not hold the permission to access any page of another domain until a grant table entry allows it. Grant tables provide a sufficiently fine-grained access control to grant memory access to particular domains (step **A.1**), which are then able to map this shared page into their own memory space (step **A.2**).

In *CaaS*, no additional access control on Grant Tables is required, as `DomU` and `DomC` are in control of their own pages and thus can by default deny any access from other domains.

Privileged Domains Exceptions to the access control realized by Grant Tables are privileged domains, such as `Dom0`. In default Xen, a privileged domain is always able to map the memory pages of other domains (step **B**). This is motivated by the fact that during domain building, the domain building VM (usually `Dom0`) has to perform modifications to the new domain's memory for a successful building. Since the domain owning this memory is not yet executing and hence cannot use Grant Tables, the concept of privileged domains was introduced to provide the privileged domain builder the necessary memory access.

In our architecture, we extend this binary access control decision with a *privileged-per-domain access control*. That means the privileged memory access of privileged domains (e.g.,

Dom0) can be granted and revoked on a per-DomU basis. Thus, the capability of Dom0 to enforce access to DomU's and DomC's memory can be revoked at runtime. This access control is enforced in the logic of the Xen hypervisor for mapping foreign memory pages into a domain's memory range. Thus, when Dom0 tries to acquire access to DomU's or DomC's memory and this privilege has been revoked, our new memory access control will return an error code. Similarly, if Dom0 has already mapped foreign memory pages of DomU when its privileged access to DomU is revoked, the memory access control enforces an unmapping of the corresponding pages from Dom0's memory range.

Additionally, we introduce so called *memory authorities*, which are conceptually privileged domains that are excepted from the access control. However, their privilege is bound to one specific DomU determined by the hypervisor. A memory authority is always able to map the memory pages of its designated DomU and is further in control of granting/revoking the privileged access by Dom0 to its DomU.

Direct Memory Access Another exceptional role concerns physical devices with Direct Memory Access (DMA) capability. These devices have by default access to the entire memory range including any guest domain (DomU and DomC). DMA is configured by the domain that is in control of the physical hardware (by default Dom0; step **C.1**). In order to prevent such an extensive memory access and provide isolation of DomU and DomC against DMA, we require hardware support in form of an IOMMU (Input/Ouput Memory Management Unit; step **C.2**). Such support was introduced, for instance, on Intel platforms with Intel's VT-d technology and on AMD based platforms with AMD-Vi. Using IOMMU, the hypervisor can control which memory regions are visible to devices and thus protect DomU's and DomC's memory regions from DMA devices misused by a malicious Dom0.

Trust Establishment and DomU Life-cycle

In order to create a VM in the default IaaS model, cloud clients can choose via a public cloud interface from a number of preconfigured images provided by the cloud service provider and other clients [BNP+11, aws]. Alternatively, clients can even upload their own images to the cloud (over the same interface). Images are by default stored on cloud storage. From there, the selected image is retrieved and deployed on a physical node in the cloud infrastructure, determined by a scheduling process, and instantiated as VM executing on top of the hypervisor on that node.

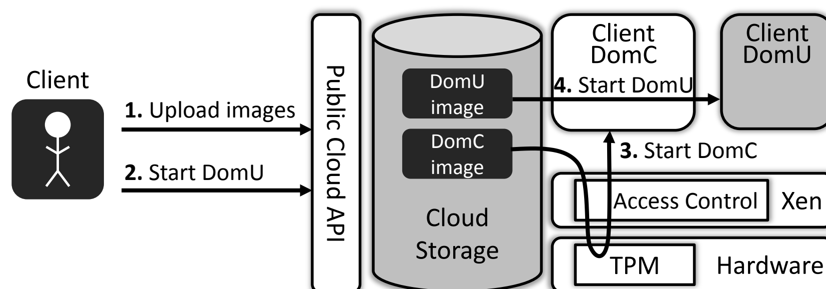


Figure 3.5: Workflow for DomU and DomC image provisioning and instantiation.

Figure 3.5 illustrates an overview of the default workflow (steps 1 and 2) and includes our extensions to this process (steps 3 and 4) in order to protect the client's keys during life-cycle management like starting or migration/suspension of DomU.

Figure 3.6 depicts this process in more detail and in the following we explain the two phases of our extensions: *Setup of Images* and *VM Instantiation*.

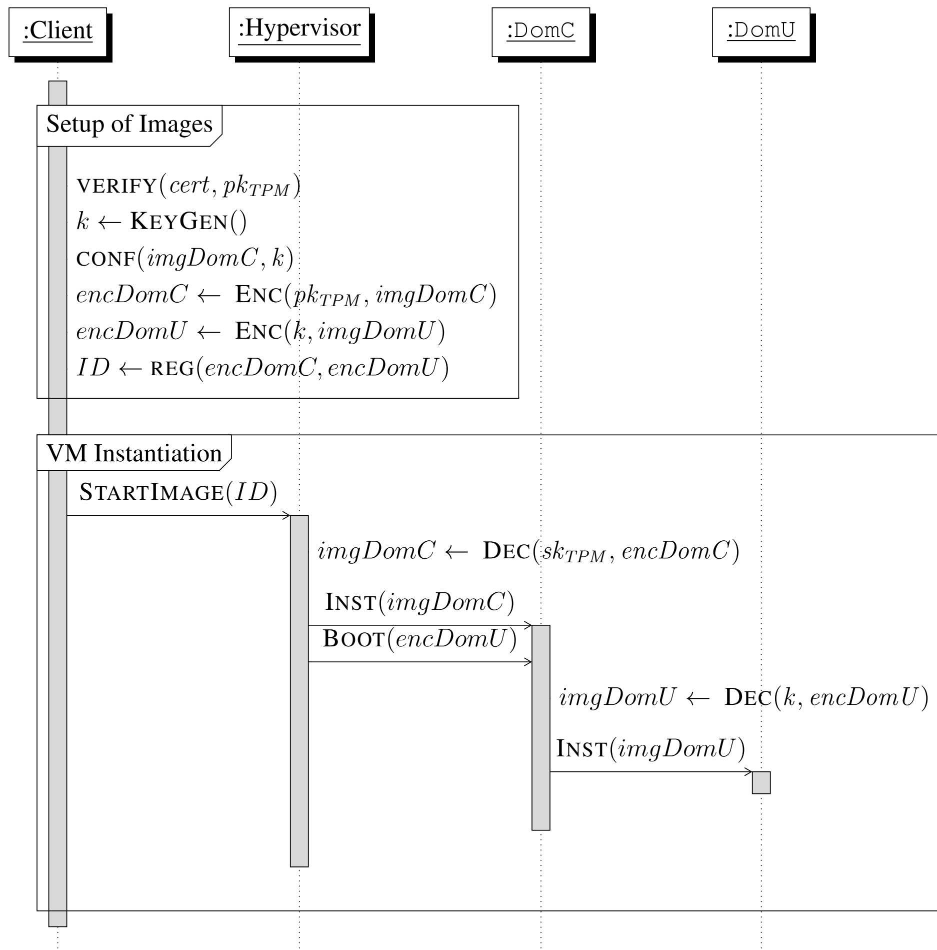


Figure 3.6: Trust establishment and VM instantiation

Setup of Images In the first phase, the client has to deploy the images of her `DomC` and her workload VM in the cloud. To ensure that the client can entrust her secrets and images to the cloud, we leverage standard TCG Trusted Computing protocols based on the Trusted Platform Module (TPM) [Tru08]. This technology provides the means to establish a trusted end-to-end channel between the client and the hypervisor in the cloud by binding all data sent over this channel to the state S of the hypervisor. Hence, the client can encrypt data such that only a platform in certain trusted state S (i.e., running our modified version of Xen) is able to decrypt this data. Technically, this is realized using a TPM key-pair (sk_{TPM}, pk_{TPM}) denoted as *certified binding key*. The usage of the private key sk_{TPM} is bound to the platform state S. A certificate *cert* proves that the key-pair was created by a genuine TPM and that this binding property holds. To make the same key available on all cloud nodes, we make this key *migratable*, i.e., its usage is bound to a platform state (or a set of trustworthy platform states) but not a particular platform. For brevity, we omit the setup of this TPM key from our protocol and refer to related work [CDE+10].

To measure the platform state, we rely on an *authenticated boot* [Tru07], which measures the platform state during boot. Moreover, besides the platform state S, the usage of sk_{TPM}

is bound to a parameter called *locality* L, which defines the layer in the platform's software stack that is able to use this key. This layer can be, for instance, the bootloader, hypervisor, VM, or application layer. We leverage this property of TPM keys to ensure that only the trusted hypervisor, but not any other software component like `Dom0`, is able to use the certified binding key sk_{TPM}. The locality of TPM commands such as using a private key sk_{TPM} is determined by the hypervisor.[5] Thus, other software layers such as a `Dom0` can still use the TPM, however, not at the locality reserved for the hypervisor.

The client verifies the TPM certified binding key pk_{TPM} using certificate *cert*. Afterwards, she generates a new secret *k* and configures her `DomC` with this *k*. Configuration means that she creates an image of her `DomC` and deploys the necessary code base (e.g., crypto libraries) and secrets for the intended mode of `DomC`, for instance, configuration as SmartCard (*Virtual Security Module* mode) or for transparent cryptographic protection (e.g., encryption) of an attached block storage (*Secure Device Proxy* mode).

Further, the client deploys her key *k* in *imgDomC*, because *k* is used to encrypt the client's workload VM image *imgDomU* and is later required by `DomC` to decrypt *imgDomU* during launch. The configured `DomC` image is then encrypted under pk_{TPM}. This forms implicitly a trusted channel between the client and the hypervisor to securely deploy *imgDomC* and thus *k*, i.e., only a trustworthy hypervisor in state S is able to decrypt *encDomC*. Both encrypted images are then uploaded and registered in the cloud under a certain *ID*. Using *ID*, the client can manage her images, e.g., issue a command to launch an instance from her `DomU` image.

In the above described workflow, `DomC` is configured, encrypted and uploaded by the client. Thus, the client has knowledge of the code executing within her `DomC` and hence trusts her `DomC` entirely. Alternatively, the client could only provision her key *k*, encrypted under pk_{TPM}, and transitively trust the hypervisor to reveal *k* only to a known-good `DomC` image. Moreover, if the client does not require a full-disk encryption of *imgDomU*, she can opt out by registering an unencrypted image and only use her `DomC` as a crypto service provider for her secret key *k*.

Instantiation of DomC and DomU The instantiation of the uploaded workload image consists of two steps as shown in the phase *VM Instantiation* in Figure 3.6: First, `DomC` is instantiated and then the workload VM `DomU` with the help of `DomC`. This process is technically more involved, since it requires the involvement of the privileged domain `Dom0` (for scheduling purposes). Thus, it requires modifications to the default `DomU` launch procedure and the hypervisor in order to protect (the confidentiality and integrity of) `DomC` during the launch. This is achieved based on our memory access control as described in Section 3.3.2. The particular steps to bootstrap the client VM (`DomU`) are depicted in Figure 3.7.

`Dom0` first allocates the memory region for the new `DomU`, assigns an identifier to it, and loads the encrypted `DomU` image into this region[6] (step 1). It similarly puts the `DomC` domain in place in memory, which is still encrypted under the key pk_{TPM} (step 2). It afterwards issues a newly introduced hypercall instantiate_DomC to the hypervisor (step 3), which takes as parameters the pointers to the memory regions of the just loaded images. The hypervisor in turn enables the memory isolation of `DomC` and `DomU` from `Dom0` (step 4). It decrypts the loaded `DomC` image with the help of the platform's TPM (step 5). Before returning control to `Dom0`, the hypervisor assigns `DomC` to be the *memory authority* for `DomU`'s memory region (step 6). This

[5] Locality is actually determined based on the memory address for communication with the TPM. However, the hypervisor is in charge of memory access control and thus controls which domain, including `Dom0`, is able to access which locality.

[6] Technically, it loads the encrypted kernel, which in turn loads the rest of the image from storage into memory during boot.

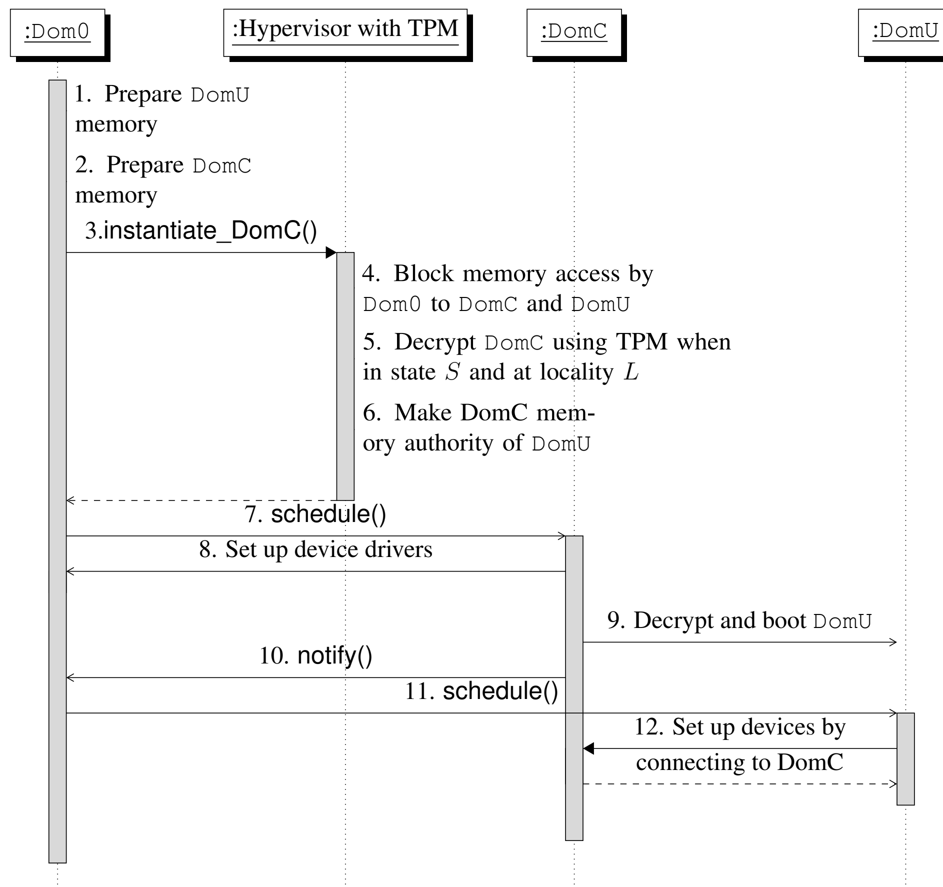


Figure 3.7: Booting DomU and coupling with corresponding DomC

means, only DomC is able to access DomU's memory directly (cf. Section 3.3.2) and to decide when to share it with Dom0 (e.g., during suspension, cf. Section 3.3.2). Once the hypervisor finished this setup and returned from the hypercall, Dom0 schedules DomC for execution (step 7).

The new DomC first negotiates with Dom0 device drivers in a regular way (step 8), connecting to the block storage device eventually intended for DomU. Afterwards, DomC bootstraps DomU by performing the steps that Dom0 normally performs for a domain launch (step 9). To exclude Dom0 from the domain building process, we include the necessary building code in DomC. The concept of disaggregating this code from Dom0 was introduced in [MMH08]. However, in contrast to [MMH08], we include this code in every client DomC instead of a central, trusted "domain building domain", thus empowering the user to keep control of this process and tightly coupling it with the cryptographic operations of DomC. Further, DomC sets up shared memory pages with DomU so that DomU is able to discover DomC and thus DomC is positioned as a proxy in between DomU and Dom0 for any I/O operation required during boot of DomU.

DomC notifies Dom0 about the readiness of DomU for execution (step 10), which in turn schedules the new DomU (step 11). During the subsequent execution of DomU, it connects to the devices offered by DomC (step 12) and it can continue as usually where DomC decrypts/encrypts DomU's I/O streams transparently. At this point, DomC can unmap DomU's memory range from its own memory since it can use ordinary grant tables for communication (cf. Section 3.3.1).

Suspension and Live Migration

Suspension and Live Migration are fairly similar in the sense that suspension saves the current execution state of a VM to storage and can restore it later at any time, whereas live migration transfers the execution state from one physical machine to another while minimizing the downtime. We only cover migration here, as this technically implies suspension.

In order to support live migration, the usual migration protocol ([CFH+05, SCP+02]) needs to be wrapped, but in essence works unaffectedly from the perspective of the client and `DomU`. Instead of migrating plaintext VM memory from one Node to another, the memory must be encrypted, since migration requires the involvement of `Dom0` in order to distribute the saved state to the target platform. Thus, before granting `Dom0` the required access to `DomU`'s memory, `DomC` encrypts this memory in-place. Then, `DomU`'s state is transferred to the target node using protocols from traditional live migration.

To restore the transferred state on the target platform, `DomC` has to be migrated as well to decrypt the migrated `DomU` state on the target platform. Restoring a VM state requires platform-dependent modifications to the state, such as rebuilding the memory page-tables. `DomC`'s domain building code performs these modifications on `DomU` during `DomU`'s resumption. While it would be possible to delegate this task to our trusted hypervisor or a central trusted domain building VM ([MMH08, BLCSG12]), this has the drawback of bloating the Xen hypervisor code base and introducing unnecessary complexity at that level or removing the client's full control over the `DomU` domain building process. However, before `DomC` can perform this task, its own state has to be restored. We opted in our design for `DomC` *program* state migration instead of VM state migration. We therefore transfer only the state of the cryptographic programs in `DomC` and do not migrate the `DomC` VM state (e.g., CPU state). To migrate the program state, the target platform instantiates a new `DomC` and updates its state with the `DomC` state of the source platform. Afterwards the new `DomC` is able to decrypt and resume the `DomU` state on the target platform and the old `DomC` on the source platform can be discarded. To achieve the protection of the transferred `DomC` state, this state is encrypted under the TPM key pk_{TPM}. Thus, only a target node running our trustworthy hypervisor is able to decrypt and resume the `DomC` state. All cloud nodes running our hypervisor form a *trusted network*, in which the client transitively trusts all nodes to securely distribute her `DomU` and `DomC`, after she has successfully verified the node on which she instantiated her `DomU`.

In case of suspension, the protocol works identical, except that the "target platform" is cloud storage to which the protected `DomC` and `DomU` states are saved by `Dom0`.

3.3.3 Evaluation

Our goal is to evaluate the performance overhead induced by offloading cryptographic operations to `DomC` for both the Secure Device Proxy and Virtual Security Module modes. Our test machine is a Dell Optiplex 980 with an Intel Core i5 3.2GHz CPU, 8GB RAM, and a Corsair 120GB SSD hard-drive connected via SATA2.

Secure Device Proxy

This setup consists of the Xen v4.1.2 hypervisor with our extensions, an Arch Linux `Dom0` (kernel 3.2.13), a Debian `DomU` (kernel 3.2.0) and a MiniOS based `DomC`. All domains and the hypervisor execute in 64-bit and each guest domain has been assigned one physical core and 256MB of RAM. `DomC` mounts a virtual block storage device provided by `Dom0` and implements

MODE	UNENCRYPTED (MB/s)		ENCRYPTED (MB/s)	
	READ	WRITE	READ	WRITE
D0	189±32	196±37	129±29	108±34
D0 ↔ DU	171±34	179±9	124±28	83 ±6
D0 ↔ DC ↔ DU	170±33	178±10	120±28	102±9

Table 3.1: Using the `fio` disk benchmark tool

MODE	UNENCRYPTED (MB/s)	ENCRYPTED (MB/s)
	WRITE	WRITE
D0	233	125
D0 ↔ DU	207	118
D0 ↔ DC ↔ DU	206	130

Table 3.2: Using the generic `dd` command

the corresponding back-end driver which `DomU` mounts via its front-end driver as a peripheral hard-disk. All I/O data streams from `DomU` to the virtual block storage is passing through `DomC` and is transparently encrypted using AES-128 in CBC-ESSIV mode based on code ported to MiniOS from the disk-encryption subsystem *dm-crypt* of the default Linux kernel.

To evaluate the performance overhead of our *CaaS* in this setup, we measure the disk I/O throughput in plain and encrypted form in a) only `Dom0` mode, b) `DomU` without `DomC` mode (this forms the baseline measurement), and c) `DomU` with `DomC` mode (which is the default mode in *CaaS*). Encryption in case a) and b) is realized with LUKS and dm-crypt in the Linux kernel of `Dom0` and `DomU`, respectively, using the same parameters as in `DomC`.

Table 3.1 summarizes our measurement results using the *fio* Linux disk I/O benchmark tool, averaged over a 20 minute disk I/O stress-test. Table 3.2 shows the averaged results for writing ten times a 10GB test file with the Linux *dd* command.

The results show, that the `Dom0`-only mode has naturally the highest throughput, since `Dom0` directly connects to the physical hardware, while the other two modes additionally require inter-domain communication. In plain mode, `DomC` induces a negligible overhead (between 0.5% and 0.6%) compared to the default direct `DomU` to `Dom0` communication, which can be attributed to the higher throughput of the shared memory based inter-domain communication in comparison to the actual disk I/O throughput in `Dom0`. Enabling encryption on disk read leads to an almost equal degradation of the performance in all modes and `DomC` imposes in this case only a minimal performance overhead (3.2%) compared to the `DomU`-local encryption. For writing encrypted data, the optimized and dedicated code base of `DomC` outperforms the `DomU` and `Dom0` based encryption (between 9.2% and 18.6%).

Virtual Security Module

Our setup consists of *SoftHSM*[7], a software-based implementation of a HSM that can be accessed via a PKCS#11 interface. We compare two scenarios: **a)** where *SoftHSM* is running in a Linux-based `DomC`, and **b)** where it is running inside a `DomU` that want to use *CaaS*. The PKCS#11 interface of *SoftHSM* is exposed to the client using *pkcs11-proxy*[8], a client-server architecture for providing PKCS#11 over a network. In scenario **a**, the server resides within `DomC` and the

[7] http://www.opendnssec.org/softhsm/
[8] http://floss.commonit.com/pkcs11-proxy.html

client in DomU, and the communication is realized through a shared memory based network communication between the two domains, which is of high-performance and does not involve Dom0. In scenario **b**, both server and client reside in DomU and the network loopback device is used.

We measure the performance of RSA signing using a HSM access via PKCS#11. This is a typical scenario found in practice, e.g., CAs signing TLS certificates or signing of domain names within the DNSSEC system. In particular we are focusing on the latter scenario and leverage the benchmark software ods-hsmspeed from the OpenDNSSEC project[9]. As parameters for ods-hsmspeed, we selected 8 threads requesting signatures from the HSM, RSA1024 as the signing algorithm, and varying number of total signatures requested ranging from 1 to 10000.

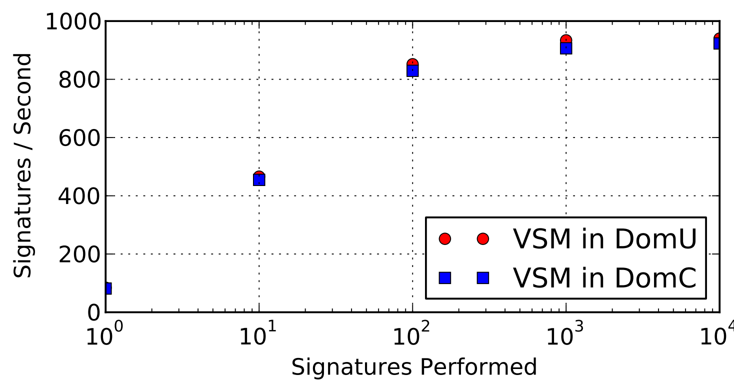


Figure 3.8: Comparing the signing performance of a software-based HSM residing in DomU vs. DomC.

Our results are illustrated in Figure 3.8. When requesting a low number of signatures, i.e., only 1 or 10, the costs for the connection and benchmark setup are significant. However in practical scenarios, we expect a large number of signatures that are requested. Comparing the performance in terms of signatures per second between a *SoftHSM* residing in DomU vs. DomC, we notice a less than 3% overhead when offloading the cryptographic operations to DomC. This is similar to the performance overhead measured for the Secure Device Proxy mode.

Further optimizations are possible when replacing the network and *pkcs11-proxy* stack by a split-driver architecture specific to PKCS#11 devices, which allows to efficiently exchange PKCS#11 commands via shared memory between domains.

3.3.4 Scalability Challenges and Possible Solutions

The *CaaS* design described so far relies on the fact that the client is ensured about the trustworthiness of the cloud node (i.e., our extended hypervisor) on which she deploys her DomU. This trust is established using TPM sealing functionality to the trusted state S (cf. Section 3.3.2). However, the standard TCG concepts have crucial drawbacks when applied in the context of *real-life* cloud computing, i.e., large-scale infrastructures:

1. The platform state S is reported using binary attestation and hence identified as a cryptographic hash (SHA1) of the software components comprising the state. Thus, minimal changes to the software stack (e.g., a security update) result in a different hash and hence the client has to be aware of all possible trustworthy hash values. Moreover, determining the trustworthiness of a hash value requires knowledge of the code base the hash is derived

[9] http://www.opendnssec.org/

from and thus the client gains full insight in the cloud provider's infrastructure. This is, on the one hand, infeasible for the provider, since his code base is his trade secret, and on the other hand unnecessary complexity for the client, who is rather interested in the software stack fulfilling certain security properties (e.g., protection of her secrets) instead of having full insight.

2. Leveraging a *certified* TPM binding key always requires at one point in the certification chain a TPM-dependent, i.e., platform-dependent, certification key[10] (independently from being a migratable or non-migratable key). Hence, the client must be able to verify that she is communicating with a genuine TPM, requiring a certification of the TPM's credentials by a certification authority, denoted in TCG terminology as *privacy CA*. However, cloud infrastructures can include hundreds of thousands of nodes on which the client's VMs can be deployed, requiring the client to verify hundreds of thousands of platforms.

3. Successfully verifying the attestation of a platform, only informs the client about the trustworthiness of the platform's software stack, but does not provide important meta-information such as the physical location of the platform. In our adversary model (cf. Section 3.2.1), the client has to be ensured that the attested platform is on the premises of the cloud provider and thus under physical control of a trusted staff of hardware administrators. Otherwise, a *logical* attacker can trick the client into deploying her secrets onto a platform running a trusted software stack, but deployed outside this trusted perimeter and thus easily prone to physical attacks by outsiders.

While providing solutions to the above mentioned scalability challenges of Trusted Computing is out of scope of this work, we want to briefly mention here our approaches towards resolving these issues. Problem 1 is a long standing problem of TCG proposed Trusted Computing and a possible solution is *property-based attestation* (see, e.g., [SSW08]), which abstracts binary measurements of software to their desired security properties. A possible solution for Problems 2 and 3 would be to establish the cloud provider as trusted Certification Authority, which provides on-demand the required certification for all its platforms within the trusted perimeter. Thus, if the provider certifies a platform, the client is assured that she is not tricked into revealing her secrets to an outside attacker. Considering the large number of platforms in a cloud (and its strong variances due to maintenance) and the consequent enormous complexity of managing this list, this approach is in practice infeasible. In *CaaS*, we conceptually solve this problem more efficiently, leveraging transitive trust (cf. migration in Section 3.3.2 and [SMV+10][SGR09]). The client initially instantiates her VM images only on a node belonging to a fixed subset of the cloud nodes (*"builder nodes"*), which can be publicly attested by the client. From these nodes, the instantiated VM is securely migrated to *"computation nodes"* for actual execution.

3.4 Security

In this section we discuss how our architecture protects the client's high-value cryptographic keys with regards to the requirements and adversary model defined in Section 3.2. We also discuss the corner cases that our architecture does not handle.

Compute Administrator Our solution protects the keys against a malicious *Compute Administrator*, because of the logical isolation of domains by the trusted hypervisor. `DomC` is

[10]This key is of type *Attestation Identity Key* (AIK) [Tru08].

not accessible by the management domain Dom0 and hence not by the compute administrators with logical access. Our extension to the domain building process of DomU based on DomC combined with the TPM based protocols (cf. Section 3.3.2) ensures that Dom0 cannot access DomC's or DomU's memory in plaintext. Any modifications Dom0 does on the encrypted images during launch will lead to integrity verification failures and abortion of the launch, and hence form a denial-of-service. The same holds for the saved, encrypted state of DomU and DomC during migration and suspension. As mentioned in our adversary model, we exclude compute administrators with physical access, since it seems there exists no practical solution against these attacks yet.

Storage Administrator Our solution protects against a malicious *Storage Administrator* by storing images only in encrypted and integrity protected form. Thus, this attacker cannot extract any sensitive information from the images and, similar to a malicious Dom0, any modifications to the images before loading them into memory result in a denial-of-service attack. Solutions against replay attacks of outdated images, which we do not consider in this work, can be based, for instance, on the TPM [vDRSD07, SWS+07].

Network Administrator Images and VM states are protected (encrypted and integrity checked) during provision to the cloud, transfer between cloud nodes and storage during migration and suspension, respectively. Thus, a malicious *Network Administrator* cannot extract the client's keys from intercepted network data. However, dropping network traffic or tampering with it will lead to a denial-of-service attack. Freshness of network communications to protect against replay attacks or injection of non-authentic data is established by using message Nonces or by establishing session keys.

Cloud Clients and Users Since keys are neither stored nor processed within a customer VM, our solution protects against *Other Cloud Client*, who consume shared VM images that are created from these VMs, and against malicious *Users*, who may compromise the clients' VMs.

On side-channel attacks In practice our solution could be vulnerable to side-channel attacks, where a malicious co-located domain can extract information from another logically isolated domain like DomC [RTSS09]. Our current approach towards tackling this problem is as follows. We restrict DomC to contain only static code and the client-control is limited to provisioning her secret keys to her DomC (see Section 3.3.2). We leverage the segregation of security sensitive operations from the workload VM DomU to mitigate side-channel attacks. When considering side-channels through any kind of shared hardware resource, DomC and malicious DomU must be physically isolated. In *CaaS*, the cloud infrastructure could be partitioned into nodes executing only DomC VMs and nodes executing only DomU VMs. Thus, no malicious code is executing on the same physical node as DomC and hence side-channels are mitigated. It is a reasonable assumption, that cloud providers can offer such separation as an additional (charged) service to his clients. However, this approach entails the additional cost that all communication between DomC and DomU is now network based instead of shared memory. In the particular case of CPU cache based side-channels [RTSS09], the same partitioning can be applied at a per-node basis by scheduling DomC VMs on different CPU packages[11] than DomU, thus avoiding shared L1 and

[11]On most multi-core CPU architectures, each CPU core has a separate L1 cache. Cores are organized into packages, where cores in the same package share the L2 cache. Usually there is a L3 cache, which is shared among all cores.

L2 caches between DomC and (malicious) DomU and only the "very noisy" L3 cache remains shared.

Oracle attacks Furthermore, if a customer VM is compromised, the attacker can misuse DomC as an oracle, e.g., to sign arbitrary messages in the client's name. This problem applies also to the default non-cloud scenario when using HSMs or SmartCards and potential solutions could be an auditing mechanism within DomC or enhanced access control within DomU.

The general problem of user-controlled approaches Like any other *user-controlled* security technique/enhancement (e.g., asymmetric cryptography or anonymity mechanisms), our *CaaS* bears the risk to be misused by malicious clients to hide malevolent/criminal activities. For instance, a malicious (or compromised) VM providing illegal content, like pirated software, cannot be inspected anymore by the cloud provider unless the client allows it. Possible solutions can be based on, e.g., establishing a mutually trusted observer for the client VMs activities, which simultaneously preserves the client VM's privacy [BLCSG12], but such discussion is out of scope of this work.

3.5 On the Implementation on Amazon EC2

Cloud clients of existing public infrastructure clouds can already use our security architecture to protect their high-value credentials against end-users compromising their workload VMs.

However, this retrofitting of our architecture will lose certain protection features, namely preventing malicious cloud administrators from reading VM memory, because the necessary Xen hypervisor modifications are not deployed in existing public clouds. Nevertheless, we believe such an implementation on existing public infrastructure clouds could be the first step for a wider adoption of the concept of client VM disaggregation and the usage of Cryptography-as-a-Service.

We describe how our implementation can be adapted for the Amazon EC2 cloud, which we chose due to its usage of Xen and its popularity. The following requirements have to be achieved for our implementation:

- Co-location of DomC and DomU.
- Establishing driver back-ends without Dom0 support for Secure Device Proxy mode.
- Establishing shared memory and control channel for optimized Virtual Security Module mode.

Co-location can be achieved through highly expensive services of the cloud provider (e.g., AWS Virtual Private Cloud and Dedicated Instances[12]) or cheaper through "brute-forcing", i.e., starting a set of virtual machines and checking for co-location among them. To our surprise, we learned that the co-location check based on the first hop in the network route as proposed in [RTSS09] is not reliable anymore. In addition to the first hop check, we propose a new way of verifying co-location among client's VMs using the *XenStore* local storage space between VMs. Essentially, for each domain we write a value to a domain-local area of *XenStore* and share this value with all potentially co-located domains. If a VM can read that value from the storage area of another VM, we verified the co-location of these two VMs.

[12] http://aws.amazon.com/dedicated-instances/

However, ideally the provider would introduce a new flag for indicating co-location when starting a set of VMs, therefore making "brute-forcing" and co-location checks obsolete.

Once `DomC` and `DomU` are co-located, we have to establish either the split-driver architecture for block and network devices for the Secure Device Mode, or establish shared memory and control channels for the optimized Virtual Security Module mode. The setup of split-drivers is currently performed and managed in Dom0 and thus in the scenario of public infrastructure clouds, the cloud client is not able to establish such split-drivers due to lack of access to Dom0. Therefore, the setup mechanisms have to be incorporated in the `DomU` and the Xen drivers in the `DomU`'s kernel have to be modified. In case of shared memory and control channels, we can leverage existing Xen mechanisms such as grant tables (cf. Section 3.3.2) and *XenBus* as the control channel. In our initial experiments, we successfully established shared memory and a control channel between various co-located VMs. For easing the inter-domain communication, we can leverage existing inter-domain communication frameworks, such as *libvchan*[13] and *XenSocket* [ZMRG07].

Amazon EC2 also supports the use of custom kernels that allow constructing a `DomC` based on *MiniOS* [Thi10] as described in our *CaaS* design (cf. Section 3.3).

3.6 Medical Use-Case

To demonstrate the benefits of *CaaS* for a real life scenario, we briefly outline in this section how *CaaS* can be used in the real life medical use-case of personal healthcare services, being developed by the TClouds partners PHI and FSR in WorkPackage 3.1 of Activity 3. For brevity, we restrict the discussion here to the security aspects of that use-case that are relevant for our *CaaS*. We provide a high-level overview of the home health care application in Section 3.6.1 and briefly discuss how our *CaaS* is beneficial in Section 3.6.2. For more details on the use-case and a more comprehensive discussion of all security requirements, we refer the reader to the TClouds deliverables D3.1.1 *"Trust Model for Cloud Applications and First Application Architecture"* (in particular Chapter 5) and D3.1.2 *"Application API and First Specification on Application Side Trust Protocols"* (in particular Section 6.4).

3.6.1 Personal Healthcare Service

In this scenario, a personal healthcare web-service is established to monitor and professionally advise patients suffering under depression. To benefit from the merits of cloud computing this service is deployed in a cloud. Since from the perspective of the healthcare provider a private cloud is not economically sustainable, the deployment should be on a public (or semi-public community) cloud.

Figure 3.9 gives a high-level overview of this healthcare service. It consists of two parts: the *Home Monitoring Device* and the *Medical Service Web-Portal*. Patients are monitored with a mobile device, e.g., a wearable device (e.g., Philips-Respironics Actiwatch), to collect on a 24/7 basis information about their sleep and physical activity as well as ambient light information. When the patient connects the device to a computer or mobile device, the collected data is uploaded to the medical service in the cloud. This service stores the collected data in a personal health record (PHR) database on cloud storage and performs data analysis, e.g., to generate a graph-based representation of the patient's activities. The results are provided to healthcare professionals for personalized coaching and to the patient for self-managed services.

[13] http://lists.xen.org/archives/html/xen-devel/2011-08/msg00806.html

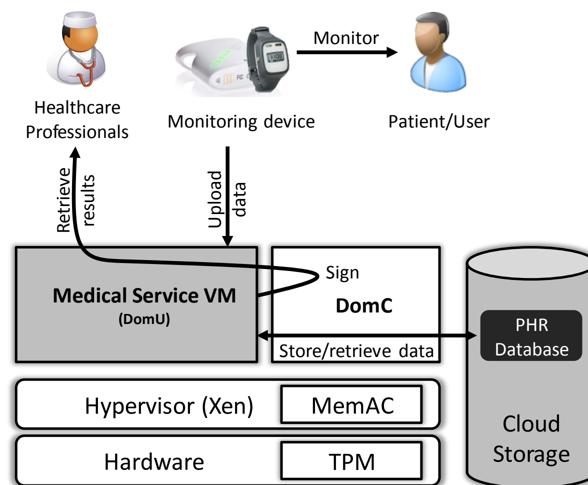


Figure 3.9: Cloud based personal healthcare service for depressed patients.

3.6.2 Security Requirements and *CaaS* Benefits

Due to their privacy sensitive nature and compliance to data protection regulations, patients' personal data stored in the cloud need to be protected. Besides requirements such as secure logging, access control, or emergency access (that are out of scope of this work), the confidentiality and integrity of the collected data must be protected against outside and inside attackers during transit and storage.[14] To ensure the protection during transit, clients connect to the web-portal over a SSL/TLS secured connection. To protect stored data, the web-portal stores the collected data on encrypted storage. Moreover, analysis results that are retrieved by clients are for legal reasons digitally signed to preserve their integrity and authenticity.

This service currently leverages *CaaS* in different ways: First, it uses `DomC` as a virtual security module to protect the TLS/SSL secret key and the document signing key. `DomC` provides a PKCS#11 interface, to which the SSL/TLS library and signing program easily and without modifications can plug in. For secure storage, `DomC` exposes a default block storage device to the medical service VM, where the service stores its files and database and that are hence transparently encrypted by `DomC`. Since the encryption is transparent, no extra configuration for secure storage has to be done in the service VM.[15]

3.7 Related Work

The area of cloud security is very active and touches various research areas. In this section, we compare our *CaaS* solution only to the closest related work.

Virtualization Security The *Terra* [GPC+03] architecture by Garfinkel et al. implements the concept of moving security management to the virtualization layer by providing two different execution security contexts for VMs on top of a trusted VMM. Our architecture differs from Terra: We provide client-controlled cryptographic primitives for multi-tenant virtualized environments (such as clouds) and thus have to tackle the challenges on how to securely provision and use those primitives in the presence of a malicious cloud management domain.

[14]Please refer to TClouds deliverable D3.1.2
[15]The protected block storage use-case has been referred to as *"Secure Block Storage"* (SBS) in past deliverables.

Today's virtualization solutions usually require a large, privileged management domain as part of the TCB. To address this issue and separate privileges, Murray et al. demonstrated on Xen the disaggregation of the management domain `Dom0` [MMH08], e.g., by extracting the domain builder functionality into a separate domain. Our *CaaS* leverages the concept of `Dom0` disaggregation in order to extract the domain building code from an untrusted `Dom0` into `DomC`. Kauer's *NOVA* [SK10] extends this disaggregation even further, by completely redesigning the virtualization software stack, based on a microkernel, and disaggregating the management domain and VMM into user-space processes.

Other related works leverage nested virtualization to advocate similar goals as *CaaS*. Williams et al. introduced the *Xen-Blanket* [WJW12], which adds an additional virtualization layer, empowering clients to avoid cloud provider lock-in. The *CloudVisor* [ZCCZ11] architecture by Zhang et al. adds a small hypervisor beneath the Xen hypervisor to protect client's `DomU` against an untrusted or compromised VMM or `Dom0` (including encrypted VM images). However, nested virtualization induces an unacceptable performance overhead by adding a second virtualization layer. In *CaaS*, we *avoid* nested virtualization and instead apply Murray's concept of `Dom0` disaggregation on top of the commodity Xen hypervisor, which is assumed trustworthy. We note, that hardening hypervisors against attacks is an active, orthogonal research area [WJ10] that benefits *CaaS*.

The closest related to our work, is the *Self-Service Cloud* (SSC) framework by Butt et al. [BLCSG12], which was developed independently and in parallel to our work. In SSC, clients are able to securely spawn their own meta-domain, including their own user `Dom0`, in which they are in control of deployed (security) services, such as `DomU` introspection, storage intrusion detection, or storage encryption. This meta-domain is isolated from an untrusted `Dom0` using a mandatory access control framework in the Xen hypervisor (XSM [SJV+05]). In contrast to SSC, we focus in *CaaS* on the specific use-case of client-controlled cryptographic operations and credentials. We tackle the challenge of how to protect and securely use our CryptoDomain `DomC`, running isolated but tightly coupled to its `DomU`. This requires modifications to the VM life cycle management, i.e., secure migration/suspension of `DomU` and instantiating fully encrypted `DomU` images.

Secure Execution Environment Instead of relying on the trustworthiness of the virtualization layer, `DomC` would ideally run in a Secure Execution Environment (SEE) that is available as a hardware security extension on modern CPUs, e.g., *Flicker* by McCune et al. [MPP+08]. However, invocations of SEE suffer from the critical drawback that they usually stop all other code executed on the underlying platform, and thus incur a significant performance penalty. Consequently, this makes them unsuitable for streaming operations such as encryption of data of arbitrary length. McCune et al. address this issue with their *TrustVisor* [MLQ+10] by leveraging hardware virtualization support of modern platforms, trusted computing technology, and a custom minimal hypervisor to establish a better performing SEE. Conceptually, TrustVisor is related to our *CaaS* from the perspective of isolating security sensitive code in an SEE, however, TrustVisor is designed to protect this code from an untrusted legacy OS while *CaaS* targets the specific scenario of cloud environments and thus faces more complex challenges: First, *CaaS* has to address an additional virtualization layer to multiplex multiple clients' VMs and their inherent scalability requirements. Second, our adversary model must consider a partially untrusted cloud provider and malicious co-located clients.

Trusted Computing Different trusted cloud computing architectures have been proposed that ensure protected execution of virtual machines. The Trusted Cloud Computing Platform (TCCP) [SGR09] by Santos et al. and the architecture proposed by Schiffman et al. [SMV+10] use TCG remote attestation to prove the trustworthiness of the cloud's compute nodes. Our approach also builds on Trusted Computing technology and concepts, but with the goal to protect cryptographic operations and credentials from external and internal attackers. Santos et al. [SRGS12] extended their TCCP architecture to address the problems of binary-based attestation and data sealing (cf. Section 3.3.4) with an approach very similar to property-based attestation [SS04], that they call policy-based sealing.

Security Modules The Xen Security Modules (XSM) introduced policy enforcement hooks at key locations within the Xen hypervisor (e.g., grant tables). XSM allows policy modules to be plugged in, which use those hooks for diverse policy enforcement, for instance, the ACM/sHype [SJV+05] module by IBM Research or the FLASK module by the NSA[16] for fine-grained mandatory access control on resources and inter-domain communication. Our architecture requires a memory access control in the hypervisor to isolate `DomU` and `DomC` from `Dom0`, however, a direct use of XSM is not the most appropriate for our *CaaS* architecture. First, this would require an adaption of the XSM to our hypervisor extensions (e.g., a new hypercall; cf. Section 3.3.2). Second, in our protocol for `DomU` launch the hypervisor decides on blocking access from `Dom0` to `DomC` and `DomU` (see step 4 in Figure 3.7). With XSM, this would require a complex communication between the hypervisor and the XSM module domain to implement this dynamic policy enforcement.

Key Management and Cryptographic Services In physical deployments, cryptographic services are typically provided by cryptographic tokens [ABCS06], hardware-security modules such as IBM's 4764 crypto processor [DLP+01], generic PKCS#11-compliant [RSA04] modules, e.g. smart cards, and the Trusted Platform Module (TPM) [Tru08]. In our approach, we study how such cryptographic services can also be securely provided in virtualized infrastructures and cloud deployments.

To provide TPM functionality to virtual machines, virtual TPMs have been proposed by Berger et al. [BCG+06, SSW08] and secure migration of VM-vTPM pairs by Danev et al. [DMKC11]. Our *CaaS* is conceptually a generalized form of such as a service, since `DomC` could also provide a vTPM daemon. However, in contrast to [BCG+06], our solution aims at protecting such a service in cloud environments and does not rely on a security service running within a potentially malicious `Dom0`.

Providing a cryptographic service over a network has been considered in large-scale networks, such as peer-to-peer or grid systems, by Xu and Sandhu [XS07]. Berson et al. propose a *Cryptography-as-a-Network-Service* [BDF+01] for performance benefits, by using a central service equipped with cryptographic hardware accelerators. Our *CaaS* targets specifically multi-tenant cloud environments and aims at tightly but securely coupling the client and her credentials within the cloud infrastructure to enable advanced applications such as transparent encryption of storage or network.

[16] See `http://wiki.xen.org/wiki/Xen_Security_Modules_:_XSM-FLASK`

3.8 Conclusion and Future Work

In this chapter we present the concept of secret-less virtual machines based on a client-controlled Cryptography-as-a-Service (*CaaS*) architecture for cloud infrastructures. Analogously to Hardware Security Modules in the physical world, our architecture segregates the management and storage of cloud clients' keys as well as all cryptographic operations into a secure crypto domain, denoted `DomC`, which is tightly coupled to the client's workloads (VMs). Extensions of the trusted virtualization layer enable clients to securely provision and use their keys and cryptographic primitives in the cloud. `DomC` can be used as virtual security module, e.g., vHSM, or as a transparent encryption layer between the client's VM and hardware resources, such as storage or network devices. Furthermore, these extensions protect `DomC` in a reasonable adversary model from any unauthorized access that tries to extract cryptographic material from the VM – either from a privileged management domain or from any guest VM. The flexible nature of `DomC` allows for building more advanced architectures, such as Trusted Virtual Domains [CDE+10], on top of our CaaS. Moreover, we discuss solutions that provide `DomC` over a network [BDF+01] in order to prevent side-channel attacks against `DomC` [RTSS09].

Evaluation of full disk encryption with our reference implementation showed that `DomC` imposes a minimal performance overhead. Moreover, we presented the partial setup of *CaaS* on the AWS EC2 cloud.

Future work aims methods to mitigate run-time attacks against `DomC`, which do not reveal but misuse the securely stored credentials. Another object of future work is to investigate how multi-core support and optimized scheduling of VMs (`DomU` and `DomC`) can help to improve the performance of client's cryptographic operations in the cloud.

Chapter 4

Remote Attestation

Chapter Authors:
Emanuele Cesena, Antonio Lioy, Gianluca Ramunno, Roberto Sassu, Davide Vernizzi (POL)

4.1 Introduction

Remote attestation is the process performed by a verifier to assess if a remote platform is trusted, i.e., according to the Trusted Computing Group (TCG), if it "will behave in a particular manner for a specific purpose" [Tru07]. The TCG specified the building blocks of a Trusted Platform providing the primitives for the so called *binary attestation*, which consists in identifying the running components (and their configuration) through their digests, to infer the platform behavior. Among the problems of binary remote attestation, scalability has often been mentioned in literature because a verifier must know all possible measurements considered acceptable. They include the measurements of the running components – problem that can be referred to as *code-diversity* – and of their configuration – that can be referred to as *system configuration*. To overcome these issues, other approaches have been proposed in the literature: property-based [SS04] and model-based [AZN+08] attestation. In this chapter, we present work-in-progress experiments showing that code diversity is a manageable issue to attest a complete Linux distribution. We also show that there is a path to address the configuration of components, whilst problems to be solved are present, like identifying the executed scripts with a low impact on the performance and other file types (keys and logs).

The chapter is organized as follows: in Section 4.2 we review the literature on scalability and preface our contribution. Section 4.3 is focused on code-diversity: we present our approach, methodology and tests results. Section 4.4 is about experiments on configuration. We conclude with Section 4.5.

4.2 On Scalability

In this section we review the definition of *scalability* in the context of remote attestation, by analyzing its occurrence in literature.

Scalability has often been mentioned as one of the main problems that limit the feasibility of binary attestation. For instance, "Binary attestation requires the verifier to know all potential hash-values of all (combinations of all) components of any machine that it may be required to verify. Knowing all acceptable configurations is hard to manage" [PSHW04] and "The remote attestation process requires for the verifier that it knows about all possible binary measurements

that are acceptable as secure software components for a given purpose. Any customization of security-critical components yields a further difficulty [...]" [KSS07].

In both contributions, we can identify two aspects of the scalability problem. According to England [Eng08], the first one is *code-diversity*, related to the identification of the processes running on a platform, while the second one is verifying the *configuration* of these processes.

This distinction was already pointed out by Chen et al.: "[...] The second problem is *complexity* since the number of different platform configurations grows exponentially with the number of patches, compiler options and software versions. [...] A further problem is the *scalability*, since update and patches lead to configuration changes" [CLL+06]. In this chapter, however, we shall follow England's terminology.

Whilst England provides figures that seem to question the viability of remote attestation (e.g., "a typical Windows installation loads 2,000 or more drivers from a known set of more than 4 million" – the latter growing at 4,000 per day), at least for the code-diversity these numbers are actually manageable by current database systems, as demonstrated by the existence of companies such as Bit9 or CoreTrace that actually base their business on *application whitelisting*. Bit9, for example, runs a Global Software Registry containing "over 5 billion records, [...] growing at a rate of up to 20 million files each day" [Bit] (this was already pointed out by Lyle et al. [LM09]).

More precise statements are given by Sailer et al. [SJZvD04]: "Client measurements grow linearly with the number of new software modules executed. [...] Verification time per measurement is constant (based on hash table retrieval), so the verification time is also linear in the number of measurements of the client. The verification space is linear with the size of the distribution".

4.2.1 Our contribution

Our long-term goal is to perform a remote attestation of a complete Linux distribution, intended as a real product running on commodity computers and not just as a prototype demonstrating that binary attestation is technically feasible. A first minimal outcome is to verify that only software "known to be good" is running. A more ambitious one would be determining the platform configuration and useful security properties. In this chapter we report the results of our experiments on the path towards our long-term goal and show, in line with [SJZvD04], that scalability is manageable. In detail we discuss the identification of the software running on a platform from the code-diversity perspective and we analyze the problem of attesting the configuration of each component. We also provide preliminary results on the performance of our approach.

Our experiments are based on freshly installed Linux distributions. Although this may be seen a strong constraint, results by St. Clair et al. [SCSJM07] suggest that this is a viable solution to define an initial trusted state.

4.3 On Code-Diversity

In this section we report the *experimental methodology*, detailed as *client-side setup* and *reference database* (*internals* and *verification procedure*), followed by *experimental results* and *costs*.

4.3.1 Experimental methodology

We chose Fedora Linux 14 as distribution for our experiments. The client platform to be attested is a commodity computer equipped with 2.6GHz Intel CPU, 2GB RAM and Infineon TPM v.1.2: we have tested on it different installations and configurations of Fedora (details are given later). On the verifier side we built a reference database containing all "known good values", i.e. the digests of all files belonging to Fedora 14 packages. For this purpose we use a dedicated commodity computer equipped with a 3GHz Intel CPU and 4GB RAM. We did not implement attestation protocols; this is out of the scope of this work and solutions can be found in literature.

4.3.2 Client-side setup

As subsystem to measure the files accessed for reading or execution, we chose Integrity Measurement Architecture (IMA) [SZJvD04] that does not require any modification to the Operating System as it is integrated in the kernel. It only needs to be enabled at the bootstrap time by adding the parameter `ima=on` to the kernel command line.

Once the kernel has been initialized, IMA starts measuring the accessed files according to the criteria specified in the policy, which can be automatically set through the kernel command line parameter `ima_tcb=1` (that identifies the *Standard* IMA policy) or provided in user space by writing all its statements to the special file `policy`, in the *securityfs* file system. Anytime, the list of measured files (with their digests) is available through another special file `ascii_runtime_measurements` from the same file system, encoded as ASCII text (or, also, in binary form, through the file `binary_runtime_measurements`).

In our tests we used three different policies: *Execution*, *Standard* and *User*. All of them share a common part, that instructs IMA to exclude special file systems from the measurements (see Listing 4.1).

```
# Don't measure files in the procfs filesystem
dont_measure fsmagic=0x9fa0
# Don't measure files in the sysfs filesystem
dont_measure fsmagic=0x62656572
# Don't measure files in the debugfs filesystem
dont_measure fsmagic=0x64626720
# Don't measure files in the tmpfs filesystem
dont_measure fsmagic=0x01021994
# Don't measure files in the securityfs filesystem
dont_measure fsmagic=0x73636673
# Don't measure files in the selinuxfs filesystem
dont_measure fsmagic=0xf97cff8c
```

Listing 4.1: IMA policy: common part.

The *Execution* policy sets IMA to measure executable code only: the main application's binary executed via `execve()` and the related shared libraries, loaded through the `mmap()` system call by either the linker-loader, after finding the required dependencies in the ELF header, or by the programs themselves using the *glibc* function `dlopen()`. All other non-executable files (e.g. for configuration) are not measured. This setting is obtained via the policy statements in Listing 4.2.

```
# Measure all files mapped in memory as executable
```

```
measure func=FILE_MMAP mask=MAY_EXEC
# Measure all files executed by the execve() syscall
measure func=BPRM_CHECK mask=MAY_EXEC
```

Listing 4.2: IMA policy: *Execution*.

The *Standard* policy is a superset of the *Execution* one since it adds all (non-executable) files read by the superuser `root` to the ones measured via *Execution* policy. In this case all system configuration files read during the bootstrap process are measured, as well as all files (configurations, plug-ins, etc.) accessed by the services – like Apache – before they change their user context from `root` to a less privileged one via `setuid()`. This setting is obtained via the policy statements in Listing 4.3 added to the *Execution* policy.

```
# Measure all files read by the root user
measure func=FILE_CHECK mask=MAY_READ uid=0
```

Listing 4.3: IMA policy: delta from *Execution* to *Standard*.

The *User* policy is a superset of the *Standard* one and we wrote it with two different specializations: *Webserver* and *Desktop*. Indeed, to the files measured via the *Standard* one, the first policy adds all files read by the Apache Web server after lowering its privileges through `setuid()`, while the second one includes all files read by the programs executed within the user context with `UID = 500` (standard non-privileged user). This setting is obtained via the policy statements in Listing 4.4 and 4.5, respectively for the *Webserver* and *Desktop* specializations, added to the *Standard* policy.

```
# Measure all files read by the apache user
measure func=FILE_CHECK mask=MAY_READ uid=48
```

Listing 4.4: IMA policy: delta from *Standard* to *Webserver*.

```
# Measure all files read by the user with uid 500
measure func=FILE_CHECK mask=MAY_READ uid=500
```

Listing 4.5: IMA policy: delta from *Standard* to *Desktop*.

Beside the variations of the IMA policies, we experimented with variations of the installed system. Indeed we tested three different installation flavors[1] of Fedora: *Minimal* that contains the essential packages required for a text login shell and to set up a network connection; *Webserver* with Apache set up to serve a PHP page and *Desktop*, a simple GNOME setup with some widely used applications: Mozilla Firefox, OpenOffice.org Writer and GIMP.

Each experiment has been performed on a fresh installation with only minimal tweaks required to each installation flavor, i.e. setting the network, updating the initial ramdisk to load the custom IMA policy and uploading the files for the workload simulation. The bootstrap time has been measured using the *bootchart*[2] tool which "... will run in background and collect process information, CPU statistics and disk usage statistics from the `/proc` file system".

[1] We call here flavors the three main installation options.
[2] http://www.bootchart.org/

Once the bootstrap process is completed, for the *Minimal* flavor we just collected the measurements by accessing the platform through a `root` shell and copying to another platform the *bootchart* (`/var/log/bootchart.tgz`) and the IMA measurements list (`ascii_runtime_measurements`).

For the other two installation flavors, before taking the measurements as described earlier, we simulated a workload. For *Webserver* by accessing a PHP test page (which simply displays the output of the `phpinfo()` function) from a remote platform and for *Desktop* by opening a new GNOME session, accessing the Fedora[3] Web site and finally opening sample files using in sequence the mentioned applications, one operation at a time.

We repeated the experiment for all combinations of installation flavor and IMA policy (except for the *User* policy with the *Minimal* installation, since we did not execute any application on behalf of regular users). At the end of this procedure, we checked each IMA measurements list against the reference database, to verify if the measured files were correctly identified as part of Fedora distribution. Test matrix and results are reported in Table 4.1 and discussed later.

Remark. Attesting that no (known) malicious binary is running on a platform is feasible, as shown later, but it does not mean that "no malicious software" is being executed. Beside binaries deemed up-to-date but holding flaws not discovered yet, script interpreters like `bash` or `python` pose a number of problems. Depending on how a script is invoked, directly as command (`./test_script.sh`) or as a parameter of the interpreter (`bash test_script.sh`), when IMA is set for the *Execution* policy, respectively measures the script or not. The *Standard* policy is then required to capture all scripts irrespective of the invocation method. Further, to make things worse, IMA does not report the operation performed on measured files and, thus, the related security impact, so that one measurement may refer to a script executed by the interpreter or to a configuration file read by the latter (e.g. `bash bash_history`).

4.3.3 Reference database: internals and data

We built the reference database with Apache Cassandra[4]. This is a highly-scalable NoSQL database, suitable to manage huge amount of data with a *key-value* structure: it is an actual candidate for offering a real service by adding new nodes for data partitions or replicas. One of the main problems of Cassandra is that in some conditions it does not guarantee immediate consistency of data in the case of concurrent write. However, this is not a problem in our case since there is only one entity entitled to update the database.

The Cassandra data model is based on the concept of *Column*, an elementary data structure with a name (or key), a value and a timestamp. It also supports a more complex data structure, called *SuperColumn*, whose value consists in a map of *Column*s, instead of plain data. The container that encloses them is called *ColumnFamily*, which in the first instance can be considered as a table in relational databases, but it appears more akin to an associative array, because data it stores do not have a fixed structure. A client can arrange data by giving the server only the *row*, i.e. the key of the associative array, to retrieve the data, or the *row* together with the *Column* or *SuperColumn* he wants to insert.

We found this model suitable for our purposes, since it provides the necessary flexibility for storing arbitrary data about files provided with the distribution, keyed by hash, as well as information about all packages identified by name.

[3] `http://fedoraproject.org`
[4] `http://cassandra.apache.org`

Our database is organized around two main *ColumnFamilies*: `FilesToPackages` and `PackagesHistory`.

`FilesToPackages` (see Listing 4.6) binds the digest of each file (the *row*) to its full path name and the packages (the *Columns*) in which it is contained. The latter are further grouped by the distribution name and the processor architecture (the *SuperColumn*) to speed up the data analysis.

For binary executable files, the *SuperColumn* also includes a *Column* named `executable`, which contains a list of the linked shared libraries. Since this list may include the name of links, instead of regular files, the *SuperColumn* also contains the *Column* `aliases`, only for shared libraries, to store the name of all their symbolic links.

```
FilesToPackages = {
   file_hash: {
      distro_name-pkg_arch: {
         rpm_file_1: 'pkg_name_1'
         rpm_file_2: 'pkg_name_2'
             ...
         fullpath: 'path_name_full'
         executable: 'linked_lib_1...linked_lib_m'
         aliases: 'lib_symlnk_1...lib_symlnk_o'
}}}
```

Listing 4.6: Structure of `FilesToPackages` *SuperColumn*.

`PackagesHistory` (see Listing 4.7) stores the history of packages. Packages are keyed by the concatenation of their name and distribution (the *row*)[5]. They contain information on the update type (the `updatetype` *Column*) for each version and release number (the *SuperColumn*) as delivered by the vendor.

The possible update types are: *newpackage*, which identifies new packages, *enhancement*, which means that the package contains new features, *bugfix*, which reports that non-security critical bugs have been corrected and, lastly *security*, which indicates that security vulnerabilities found in a older version have been solved.

```
PackagesHistory = {
   pkg_name-distro_name: {
      pkg_version-pkg_release: {
         name: 'pkg_name_full'
         updatetype: 'newpackage'|enhancement'|'bugfix'
                     |'security'
}}}
```

Listing 4.7: Structure of `PackagesHistory` *SuperColumn*.

Examples of data stored in the reference database and expressed with JSON can be found in Listing 4.8.

```
FilesToPackages = {
    1d8f2dc451e76ad88077566a49bf8f1be920d639: {
        Fedora14-x86_64 : {
```

[5]This was chosen to exactly identify the package history related to a specific distribution, as vendors use proprietary naming convention and release updates

```
            executable: 'libblkid.so.1,libuuid.so.1,'
                        'libselinux.so.1, libsepol.so.1,'
                        'libc.so.6,ld-linux-x86-64.so.2,'
                        'libdl.so.2',
            fullpath: '/bin/mount'
            util-linux-ng-2.18-4.8.fc14.x86_64.rpm:
                        'util-linux-ng-2.18-4.8.fc14'
}}}

PackagesHistory = {
    util-linux-ng-Fedora14: {
        2.18-4.3.fc14: {
            name: 'util-linux-ng-2.18-4.3.fc14',
            updatetype: 'newpackage'
        },
        2.18-4.8.fc14: {
            name: 'util-linux-ng-2.18-4.8.fc14',
            updatetype: 'security'
}}}
```

Listing 4.8: Example data in the reference database expressed with JSON.

The database is daily updated as described in the following. First, new packages and updates are downloaded from the official Fedora repository (with `rsync`).

Then, a `python` script unpacks the fetched packages, computes the `sha1` hash value of the files contained and eventually inserts the measurements and the full path name into the *ColumnFamily* named `FilesToPackages`. The script collects additional information about unpacked files: it determines whether the file is an ELF main executable (with `readelf`) and in this case it retrieves the list of linked shared library (with `ldd`); or it resolves the name of the regular file for links to shared libraries. This information is used to update data previously inserted into `FilesToPackages`.

Finally, the script populates `PackagesHistory` through `bodhi`[6], a service offered by Fedora which provides information about its packages, including the update type.

4.3.4 Reference database: verifying a platform

We created a second `python` script that can be used by a verifier (also via Web front-end[7]) to query the database and perform the analysis to validate the platform configuration. Our `python` scripts use a library called `pycassa`, which simplifies the interactions with Cassandra. The script that performs the verification of the platform integrity can connect remotely to the database. At this stage, our verification method consists in checking if digests of files, collected by IMA in the client platform, are recognized as part of Fedora. Anyway, a deeper analysis is necessary to assert that all components running on a platform are good. Therefore, the script verifies if a measured file is up-to-date or if a newer version has been released to fix security flaws, by using the version and the full history of the packages containing it. Further, by using the dependencies between main executables and libraries, the script can evaluate if a whole process is up-to-date or not.

The script exploits the Cassandra *multiget* query to batch requests. The verification process requires two queries: the first one to identify the software running on the platform; the second

[6] https://fedorahosted.org/bodhi
[7] Available at http://security.polito.it/tc/ra/verify

one, to assess the freshness of the components.

In the first query, the script determines which measurements correspond to software provided within the distribution. It queries the database by sending all the digests collected by IMA and receives the associated *SuperColumn*s of `FilesToPackages`. The measurements that are *not* returned indicate files not provided within the distribution.

Ideally, if the distribution was only updated using official packages, all of these measurements should be in the database. Together with the measurements, the server also specifies the packages in which the files are contained.

The result of this first query is the basis for the analysis that, depending on the verifier's requirements, can stop whenever a single unknown file exists, or can proceed trying to derive finer information about what is known or unknown.

Here we describe a graph-based analysis that determines the freshness of each process running, taking into account the dependency from shared libraries.

The script builds a graph whose nodes are files and packages and edges represent dependencies. Each node has an attribute describing its status. From data returned by the first query, the script: 1) creates a *file node* for each measurement; 2) sets the status for each file node; 3) indexes shared libraries by file name and all its aliases; 4) for each executable, connects its linked shared libraries to it; 5) creates a *package node* for each package found in the query result; 6) connects each package to the files that it contains.

In step 2, the status can be: *ok* or *not-found* if the hash value was respectively found or not in the database; *name-mismatch* if the name returned from the database for a given digest differs from that in the measurement list. In step 4, executables are recognized by the presence of the `executable` *Column* in the *SuperColumn*. In this case, the list of libraries (by file names or aliases) stored in the column allows to find which file nodes need to be connected. Libraries not present in the graph (because of measurements missing) are added anyway as file nodes with status *fake* and connected to the executable to make the algorithm correct.

After these steps, a second query to the database is performed to decide if the software running on the platform being attested is up-to-date. The script selects all packages returned by the first query, concatenates their name with that of client platform's distribution, and sends the obtained *row*s to the database. The database replies with the *SuperColumn*s of `PackagesHistory` that contain the version and release numbers of requested packages as delivered by the vendor of the given distribution.

For each package, the script: 1) finds all newer versions than the one from the first query; 2) checks the update type of each newer version and selects the most critical one; 3) updates the *status* attribute of the *package node*. Here, the status can be *ok* if the package is up-to-date, *bugfix* if a newer version fixes non-security critical bugs or *security* if discovered security flaws are solved in a more recent version.

Then, the script performs a breadth first propagation of the status in the graph: the status of packages is propagated to files and the status of libraries to binary executables linking them (*security* overwrites *bugfix* that overwrites *ok*). For instance, a status *security* from the package `openssl` is propagated to `libssl.so` and then to Apache `mod_ssl`.

At the end of the execution, the script returns to the verifier the result of the verification either in table form, with the information collected for each file and package, or in graphical form, with an image of the graph for each file measured by IMA, if it belongs to the distribution and the update status, also including all other metadata returned by the database.

4.3.5 Remote attestation: experimental results

The test matrix with the results of the measurements on the client side and the verifications are reported in Table 4.1: the data are obtained by submitting the stored IMA measurements to the Web front end. For each combination of installation flavor and IMA policy (the selected options are indicated by the text respectively of row and column headers), three figures are reported: (1) the total number of measurements done by IMA according to the selected policy; (2) the number of unknown measurements[8], i.e. digests not found in the reference database: the corresponding files are not recognized as part of Fedora; (3) the verification time expressed in seconds. Each figure, related to a single combination, is the average value of verification times measured during five tests: we used the system timer within the script to measure these times for the whole verification procedure, excluding all communications and the data rendering. We restarted Cassandra before each test to flush its cache.

The figures for the *Execution* policy show that all executable code (main application binaries and shared libraries) running on the platform, from the bootstrap to the time the list of IMA measurements is saved, is recognized as part of Fedora. This is a first positive result.

Moreover, our verification procedure also checks whether a file recognized as part of Fedora is up-to-date or not and, in the latter case, for which reason.

These results show that we reached our minimal goal, i.e. *verifying that only software "known to be good" is running*.

Besides, we found a low percentage of measurements (12%, in the worst case), in the lists obtained with the *Standard* and *User* policies, that were not recognized (i.e. the returned digests were not in the database).

A deeper look at those records, e.g. for the case *Desktop-User*, shows that the number of unrecognized files is lower. Indeed, out of 386 records, 151 are just violation reports (i.e. a reader and writer access a file at the same time), thus obtaining 235 records for unknown files. Further, records with the same file name appear often. Since the IMA version used in our tests is not able to report more information about measured files than their name, it is not possible to distinguish if two records with the same file name refer to a single *inode* or to two different ones. However, we verified that in our client installation no duplicate files exist, so in our test bed multiple records with the same file name represent multiple measurements of the same file. Therefore, taking only the records with unique file names and excluding the record `boot_aggregate` (that reports the digest of aggregated values from `PCR0` to `PCR7`) lead to identify just 186 unknown files (i.e. 5%).

After a further inspection to them, we noted that the unrecognized files can be classified in three main categories: configuration files, keys or logs. We refer the reader to Section 4.4 for a method that allows to reduce the number of unknown measurements for configuration files.

4.3.6 Remote attestation: the costs

The cost of the verification in terms of time appears to be reasonable. In fact, the chosen test bed and the experiments setup – i.e. a single node configuration on a standard commodity desktop, only one request at a time and the cache flushed at every test – were not intended to measure the performance of our reference database using an optimized, production-like, configuration of Cassandra. Our goal was only to check whether there is a reasonable upper bound for the verification time that makes our approach feasible or not: we think that for these preliminary tests, a worst case of 3.3s is acceptable (note that the TPM requires about a second to sign).

[8] The percentage is the ratio between the numbers of unknown and total measurements.

Installation Flavor	IMA Policies								
	Execution			Standard			User (i.e. Webserver/Desktop)		
	Measurements #		Verify	Measurements #		Verify	Measurements #		Verify
	Total	Unknown	time (s)	Total	Unknown	time (s)	Total	Unknown	time (s)
Minimal	201	0	0.36	408	48 (12%)	0.93	n/a	n/a	n/a
Webserver	494	0	0.59	1772	75 (4%)	1.86	1934	93 (5%)	2.09
Desktop	1077	0	1.03	2496	178 (7%)	2.55	3683	386 (10%)	3.30

Table 4.1: Number of measurements and verify time for all IMA policies and installation flavors.

In Table 4.2 we reported the measured time of the bootstrap with IMA disabled and we repeated the procedure for each of the three mentioned policies to show the overhead of measuring. The cost of having IMA enabled on the client platform is quite relevant, especially in the case of *Standard* and *User* policies, required to measure the system configuration and the scripts in addition to executable binaries.

Installation Flavor	IMA off	Execution Policy	Standard Policy	User Policy
Minimal	23	29 (+26%)	35 (+52%)	n/a
Webserver	30	40 (+33%)	67 (+123%)	70 (+123%)
Desktop	30	39 (+30%)	63 (+110%)	63 (+110%)

Table 4.2: Bootstrap time in seconds.

We also report the costs of the reference database. Fedora 14, daily updated within 8 months since the release, contains 45GB of packages that result in a database with 1.9GB of data (1GB just for the release version) describing more than 22,000 packages for a total of around 2.9 millions files. During this period we observed more than 14,000 updates with around 3,200 new digests per day, i.e. almost 110MB added to the database per month.

The cost of the reference database for 8 months updates is, in term of space, 4.2% of the size of one distribution version. Under the hypothesis of linear growth, the cost for two years updates would be 8.2%. However we can imagine that whenever a new major version is released, the growth rate for the previous one will be lowered until it will be discontinued. These considerations and the measured verification times show that, on the verifier side, code-diversity can be successfully managed, solely basing on the known good values provided by the Linux distributor.

4.4 On Configuration

Our experiments showed an unexpectedly high number of measured configuration files matching the original ones included in the packages – about 317 of 360 files residing in the directory `/etc` – with their digests, hence, present in the reference database.

Besides, some of these configuration files (even if not present in the database) can be re-generated on the verifier side from a template file and simple platform properties. To obtain the same content, the original files should be modified only through the related tools, when available, or validated before their use. Then, reconstructed files can be compared against the measured ones, thus reducing the number of unrecognized digests (in our first tests we identified 20 more files).

To pursue this goal, we took as an example *anaconda*, the Fedora installer, that generates some configuration files from a fixed structure (stored internally) filled using user selected preferences (stored in `anaconda-ks.cfg`, with other platform data useful for automated installations) and places the resulting file onto the target storage media. In some cases, *anaconda* performs this operation by invoking an external program.

We created two `python` scripts: a *collector*, that must be run on the client after the bootstrap and generates a file `platform_properties.list` (see an example in Listing 4.9) containing the platform properties, i.e. data from `anaconda-ks.cfg` and additional data that we

identified to reconstruct more configuration files; and a *generator*, integrated in our verification script, that takes as input `platform_properties.list` and deals with each configuration file through specific modules, each one managing a single property. To recognize additional configuration files, new properties, not present in `anaconda-ks.cfg`, must be defined (see, e.g., `cdrom` in the second example) and new modules must be developed.

```
# content of platform_properties.list for the examples
i18n en_US.UTF-8 latarcyrheb-sun16
cdrom cdrom|cdrw|dvd COMBO_SHC-48S7K
    pci-0000:00:1f.2-scsi-1:0:0:0
keyboard us
firewall --service=http
selinux --enforcing
timezone Europe/Rome
```

Listing 4.9: Example file `platform_properties.list`.

Figure 4.1 shows how our tool can reconstruct the configuration file `/etc/sysconfig/i18n` for the language of the Fedora distribution by reading the language identifier and the system font from the standard `i18n` property, and by formatting these data using the associated template.

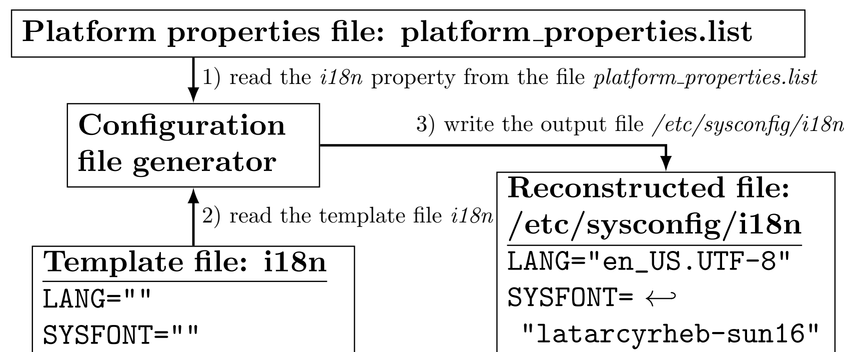


Figure 4.1: */etc/sysconfig/i18n*

Figure 4.2 shows another example, i.e. a rule for *udev* that defines how optical device files are created during the bootstrap (`/etc/udev/rules.d/70-persistent-cd.rules`) and that is more complex to be reconstructed. Our tool parses the `cdrom` property and sets the proper `bash` environment variables. Then, it launches `/lib/udev/write_cd_rules`, an external program part of the *udev* package, which generates the desired rules according to the variables previously set.

Further, we note that many configuration files have a well-known structure (e.g. property=value) useful to derive system properties helpful to assess the overall security, like verifying the authentication methods accepted by SSH server.

This observation and our experiments bring to the conclusion that a distributor could easily integrate a tool to automatically reconstruct configuration files and provide the application developers with the related support.

Despite that we successfully identified many configuration files, others were not recognized. Moreover, how to handle the measurements of other file types, such as cryptographic keys or logs, is not yet clear.

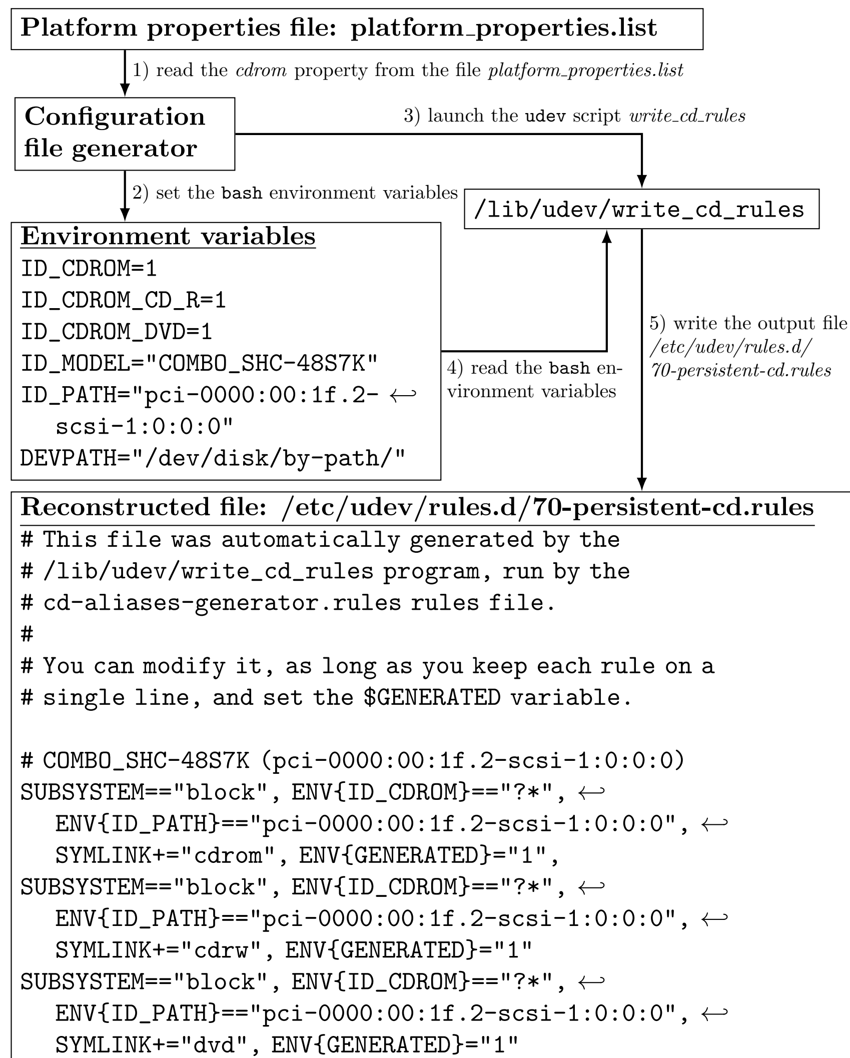


Figure 4.2: */etc/udev/rules.d/70-persistent-cd.rules*

Finally, we also ran our tests on a non fresh installation of the *Desktop* flavor: the number of unknown measurements increased from 10% to 20%.

4.5 Conclusions and future work

In conclusion, we showed that code-diversity can be managed when attesting a complete Linux distribution: our reference database could be directly maintained by the distributor at low cost. Open issues are identifying scripts with low impact on the performance and files likes keys and logs. For configuration files, instead, we identified a path to be further explored to increase the number of measurements that can be successfully recognized.

Chapter 5

Mobile Device Clouds

Chapter Authors:
Sven Bugiel (TUDA)

5.1 Motivation

In this section, we briefly discuss today's predominant service delivery paradigms *clouds* and *apps* (cf. Section 5.1.1), their interaction, and why designing security solutions for cloud-based infrastructures has to take a consolidated look on the interaction of clouds and mobile devices (cf. Section 5.1.2). In conclusion, we advocate that mobile devices extend the security perimeter of security domains within clouds (e.g., Trusted Virtual Domains; cf. Chapter 12 of TClouds deliverable D2.1.1) and hence necessitate mobile platform security architectures that assert the security policies of the cloud on the mobile platform (cf. Section 5.1.3). We present the design and implementation of an Android OS based architecture, tailored to the purpose to securely integrate mobile platforms in (virtual) security domains of large-scale (cloud-based) infrastructures (cf. Section 5.2).

5.1.1 Predominant Web-Service Delivery Paradigms

In the recent years, two predominant paradigms have evolved to deliver ubiquitous services to end-users and customers:

Clouds Clouds provide their clients virtual resources such as storage, network, and computation, on a highly flexible on-demand and pay-per-use basis. These resources are ubiquitously accessible and provide in particular for web-service providers operational and monetary benefits when deploying their service cloud-based [CS11].

Apps Apps are small programs for a specific purpose (e.g., calendar, social networking, etc.) and can be based on different platforms, for instance, browser based or desktop based. However, the most common form of apps today are mobile apps, i.e., apps running on smart devices such as smartphones or tablets. These apps provide end-users a very convenient way to customize their mobile smart devices and have become a desired companion in our every day lives. Although mobile apps technically deploy the service on the mobile device, new affordable data plans by mobile carriers allow for an (almost) permanent Internet connection of mobile devices and hence enable mobile apps to be backed up by Internet-based services (e.g., Google maps for for map-based services).

5.1.2 Extended Security Perimeter of Cloud-Based Virtual Infrastructures

Not surprisingly, clouds and (mobile) apps harmonize very well and service providers have quickly adapted to this combination. For instance, Amazon introduced the cloud Player[1], consisting of a small app that plays music streams retrieved from the Amazon cloud instead of first downloading the music file. While mobile apps provide convenient end-user front-ends on ultra-mobile devices and leverage the mobile platforms' hardware for better user-experience (e.g., powerful sensors like accelerometers and gyroscopes), clouds provide the ideal infrastructure for service back-ends (i.e., high-availability, ubiquitous access, elasticity, vast storage, high network bandwith).

When considered from network topology point of view, this interaction between the cloud and mobile devices exhibits two distinct patterns for data flows. First, the "classical" form of a star topology (cf. Figure 5.1), centered around the cloud. Second, data retrieved from the cloud is further distributed in a meshed network (or p2p network) between mobile devices (cf. Figure 5.2) with physical proximity.

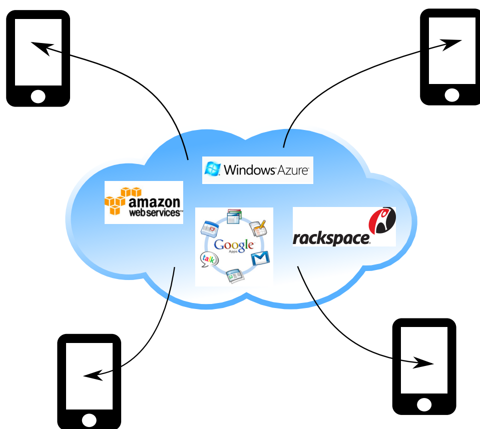


Figure 5.1: Classical star topology: Data flow from central cloud to mobile devices.

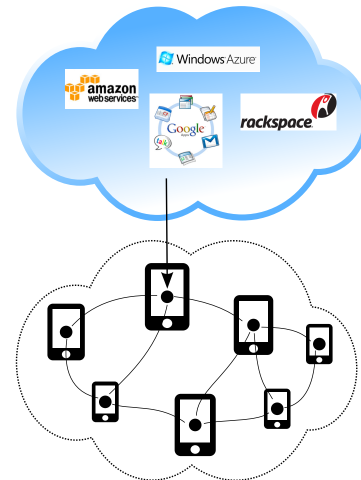


Figure 5.2: Distributed data: Flow within *Mobile Device Cloud*.

Star Topology The star topology stems from the default client-server relationship between the mobile devices and the cloud-based service, where mobile apps connect as described above to the cloud-based services. By default, all client devices do not interact directly with each other, but instead exchange data and synchronize with each other via the cloud.

Device Clouds Device clouds are formed when mobile platforms (ad-hoc) inter-connect in a wireless meshed network (WMN). This is enabled due to advances in wireless device-to-device communication, such as Bluetooth Low-Energy, WiFi Direct, Direct Link Setup (802.11z), Qualcomm FlashLinq and advances in ad-hoc routing (e.g., BATMAN or OLSR). In this topology, data retrieved by one device from the cloud service is distribute to proximal peer devices at local high-speed bandwiths (usually much higher than mobile Internet connections). This is in particular beneficial when considering locally-relevant data, e.g., traffic conditions, localized

[1] http://www.amazon.com/cloudplayer

social networking, or sharing photos. First apps using this new technology have been presented by academia (e.g., floating content [OHL+11]) or industry[2].

5.1.3 The Need for Mobile Platform Security

As described in the preceding sections, mobile devices extend the perimeter of cloud-based infrastructures, and thus it is crucial that mobile devices are equipped with a security architecture that ensure the enforcement of security policies of the virtual infrastructure on the mobile device. As an example, we briefly explain this problem in the particular context of Trusted Infrastructures (TClouds deliverable D2.1.1, chapter 12).

Trusted Virtual Domains (TVDs) establish logical isolated domains among Virtual Machines within cloud environments and independently from the physical host. Strict security policies are enforced on the information flow between domains and by default domains are strongly isolated. TVDs are closed environments and new members are only admitted to a domain, when they could proof that their system provides the necessary security architecture to logically isolate different domains from each other and that they adhere to the security policies of the domain. Considering the importance of cloud-based services for clients such as mobile devices, this design entails that all clients have to support the necessary security architecture to deploy TVDs. While there exist solutions for desktop and laptop machines, this is an open challenge for mobile devices.

5.2 Practical and Lightweight Domain Isolation on Android

In this section, we describe a security architecture for the popular Android OS, which is tailored towards securely integrating mobile devices into larger infrastructures and asserting the security policies of the surrounding infrastructure on the mobile platform [BDD+11b]. These infrastructure can be, for instance, Trusted Virtual Domains as developed in context of the TClouds project.

5.2.1 Introduction

The market penetration of modern smartphones is high and sophisticated mobile devices are becoming an integral part of our daily life. Remarkably, smartphones are increasingly deployed in business transactions: They provide employees a means to remain connected to the company's network thereby enabling *on the road* access to company's data. In particular, they allow employees to read and send e-mails, synchronize calendars, organize meetings, attend telephone and video conferences, obtain news, and much more. On the other hand, mobile platforms have also become an appealing target for attacks threatening not only private/personal data but also corporate data.

Until today, the Blackberry OS is the most popular operating system used in the business world. However, recent statistics manifest that Google Android is rapidly expanding its market share[3], also in the business world, where it is currently the third-most used mobile operating system after Blackberry and iOS [Car10].

[2] http://opengarden.com/

[3] At the time of writing, Android has 36% market share and belongs to the most popular mobile operating systems worldwide [Gar11].

Security Deficiencies of Android. The core security mechanisms of the (open source) Google Android OS [Goo] are *application sandboxing* and a *permission framework*. However, recent attacks show that Android's security architecture is vulnerable to *malware* of many kinds. First, uploading malicious applications on the official Android market is straight-forward, since anyone can become an Android developer by simply paying a fee of $25. Second, Google does not perform code inspection. Recent reports underline that these two design decisions have led to the spread of a number of malicious applications on the official Android Market in the past [Nil10, Goo10a, Loo10, Bra11]. Further, another attack technique against Android is privilege escalation. Basically, these attacks allow an adversary to perform unauthorized actions by breaking out of the application's sandbox. This can be achieved by exploiting a vulnerable deputy [EOM09a, DDSW10, FWM+11], or by malicious colluding applications [SZZ+11, BDD+11a]. In particular, privilege escalation attacks have been utilized to send unauthorized text messages [DDSW10], to trigger malicious downloads [LRW10, Nil10], to change the WiFi settings [FWM+11], or to perform context-aware voice recording [SZZ+11]. Finally, approaches that rely on Android's permission framework to separate private applications and data from corporate ones (such as enterproid [ent]) will likely fail due to the above-mentioned attacks. Moreover, any attack on the kernel-level will allow the adversary to circumvent such solutions.

Domain Isolation with Default Android. In the light of recent attacks, the Android OS cannot meet the security requirements of the business world. These requirements mainly comprise the security of a heterogeneous company network to which smartphones connect along with the protection of corporate data and applications (on the phone). In particular, Android lacks data isolation: For instance, standard Android only provides single database instances for SMS, Calendar, and Contacts. Hence, corporate and private data are stored in the same databases and any application allowed to read/write the database has direct access to any stored information. Apart from application sandboxing, Android provides no means to *isolate* corporate applications from private user applications in a system-centric way. Hence, an adversary could get unauthorized access to the company's network by utilizing privilege escalation attack techniques. Finally, Android fails to enforce isolation at the network-level which would enable the deployment of basic context-aware policy rules. For instance, there is no means to deny Internet access for untrusted applications while the employee is connected to the company's network.

To summarize, default Android has no means to group applications and data into *domains*, where in our context a domain compromises a set of applications and data belonging to one trust level (e.g., private, academic, enterprise, department, institution, etc.)

Existing Security Extensions to Android. Recently, a number of security extensions for Android have been proposed, the closest to our work being [OMEM09, EGC+10, OBM10, NKZS10, BDD+11a]. However, as we will elaborate in detail in related work (see Section 5.2.7), all of these solutions focus on a specific layer of the Android software stack (mainly Android's middleware) and fail if the attack occurs on a different layer, e.g., at the network layer by mounting a privilege escalation attack over socket connections [DDSW10]). Specifically, they do not address kernel-level attacks [Obe10, LRW10] that allow an adversary to access the entire file system. Having said that, attacks on the kernel-level can be mitigated by enabling SELinux on Android [SFE10]. However, SELinux only targets the kernel-level, and misses high-level semantics of Android's middleware.

In particular, we are not aware of any security extension providing efficient and scalable application and data isolation on different layers of the Android software stack, which is essential for deploying Android in the business world.

On the other hand, several virtualization-based approaches aim at providing isolation between private and corporate domains on Android [Ope, BBD+10]. However, contemporary mobile virtualization solutions suffer from practical deficiencies (see Section 5.2.7): (1) they do not scale well on resource-constrained smartphone platforms which allow only a limited number of virtual machines to be executed simultaneously; (2) more importantly, virtualization highly reduces the battery life-time, because it duplicates the whole Android operating system. This raises a severe usability problem.

Our Contribution. In this section, we present a novel security architecture, called *Trust-Droid*, that enables *practical and lightweight domain isolation on each layer* of the Android software stack. Specifically, *TrustDroid* provides application and data isolation by controlling the main communication channels in Android, namely IPC (Inter-Process Communication), files, databases, and socket connections. *TrustDroid* is lightweight, because it has a low computational overhead, and requires no duplication of Android's middleware and kernel, which is typically a must for virtualization-based approaches [Ope, BBD+10]. As a benefit, *TrustDroid* offers a good scalability in terms of the number of parallel existing domains. In particular, *TrustDroid* exploits coloring of separate and distinguishable components (this approach has its origins in information-flow theory [Rus81]). We color applications and user data (stored in shared databases) based on a (lightweight) certification scheme which can be easily integrated (as we shall show) into Android. Based on the applications colors, *TrustDroid* organizes applications along with their data into logical domains. At runtime, *TrustDroid* monitors all application communications, access to common shared databases, as well as file-system and network access, and denies any data exchange or application communication between different domains. In particular, our framework provides the following features:

- **Mediating IPC:** We extend the Android middleware and the underlying Linux kernel to deny IPC among applications belonging to different domains. Moreover, *TrustDroid* enforces data filtering on default Android databases (e.g., Contacts, SMS, etc.) so that applications have access only to the data subset of the their respective domains.

- **Filtering Network Traffic:** We modified the standard Android kernel firewall to enable network filtering and socket control. This allows us to isolate network traffic among domains and enables the deployment of basic context-based policies for the network traffic.

- **File-System Control:** We extend the current Android Linux kernel with TOMOYO Linux based mandatory access control and corresponding TOMOYO policies to enforce domain isolation at the file-system level. This allows us to constrain the access to world-wide readable files to one specific domain. To the best of our knowledge, TOMOYO has never been applied on a real Android device (e.g., Nexus One) before.

- **Integration in Trusted Infrastructures:** Our design includes essential properties and building blocks for integrating Android OS based smartphones into sophisticated trusted infrastructures, such as Trusted Virtual Domains [DEK+09].

We have tested *TrustDroid* with Android Market applications and show that it induces only a negligible runtime overhead and minimally impacts the battery life-time.

Outline. The remainder of this section is organized as follows: In Section 5.2.2 we briefly recall the Android architecture and in Section 5.2.3 we provide a problem description, present our adversary model, and elaborate on our requirements and objectives. We present the architecture

of *TrustDroid* in Section 5.2.4 and describe its implementation in Section 5.2.5. Our results are evaluated and discussed in Section 5.2.6. We summarize related work in Section 5.2.7 and conclude in Section 5.2.8.

5.2.2 Android

In the following we briefly provide background information on Android. We explain the Android software stack, the types of communications present in the system and elaborate on the specifics of Android's security mechanisms.

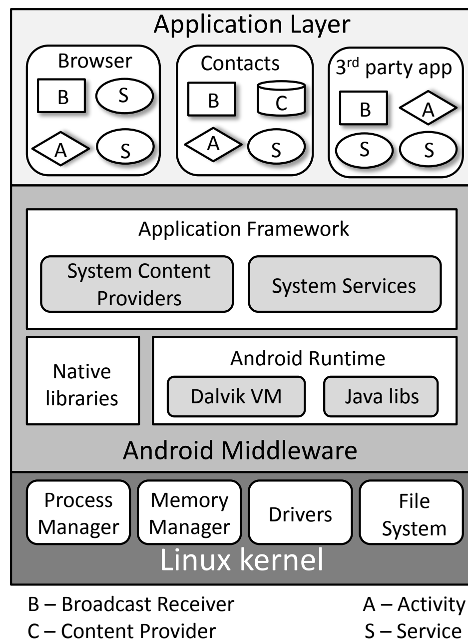


Figure 5.3: Android architecture

Software Stack

Android is an open source software stack for mobile devices, such as smartphones or tablets. It comprises of a Linux kernel, the Android middleware, and an application layer (as depicted in Figure 5.3). The Linux kernel provides basic facilities such as memory management, process scheduling, device drivers, and a file system. On top of the Linux kernel is the middleware layer, which consists of native libraries, the Android runtime environment and the application framework. The native libraries provide certain core functionalities, e.g., graphics processing. The Android runtime environment is composed of core Java libraries and the *Dalvik* Virtual Machine, which is tailored for the specific requirements of resource constrained mobile devices.

The Android application framework consists of system applications written in C/C++ or Java, such as System Content Providers and System Services. These provide the basic functionalities and the essential services of the platform, for instance, the Contacts app, the Clipboard, the System Settings, the AudioManager, the WifiManager or the LocationManager. While System Content Providers are essential databases, System Services provide the necessary high-level functions to control the device's hardware and to get information about the platform state, e.g., location or network status.

At the top of the software stack is the application layer, which contains a set of built-in core applications (e.g., *Contacts* or *Web-browser*) and third party applications installed by the user (e.g., from the Android MarketStore[4]). Applications are written in Java, but for performance reasons may include native code (C/C++) which is called through the *Java Native Interface* (JNI). In general, Android applications consist of certain components: *Activities* (user interfaces), *Services* (background processes), *Content Providers* (SQL-like databases), and *Broadcast Receivers* (mailboxes for broadcast messages).

Communication

Android provides several means for application communication. First, it implements a Binder-based[5] lightweight Inter-Process Communication (IPC), which is based on shared memory. This is the primary IPC mechanism for the communication between the application components. This mechanism has been denoted as *Inter-Component Communication* (ICC) in [EOM09b] and since then this term has been well established. For ICC, a special definition language (*Android Interface Definition Language* – AIDL) is used to define the methods and fields available to a remote caller. An example of ICC calls is the binding to a remote service, thus calling remote procedures exposed by this service. Further, explicit actions on a different application can be triggered by means of an *Intent*, a message with an URL-like target address, holding an abstract description of the task to perform (e.g., starting an Activity). Second, the Linux kernel provides the standard IPC mechanisms, e.g., based on Unix domain sockets. Third, applications with the Internet permission are allowed to create Internet sockets. Thus, they are not only able to communicate with remote hosts but also connect to other local applications.

Security Mechanisms

Android implements a number of security mechanisms, most prominently application sandboxing and a permission framework that enforces mandatory access control (MAC) on ICC calls and on the access to core functionalities. In the following, we provide a brief summary of these mechanisms and refer to [EOM09b] for a more detailed discussion

Sandboxing. In Android every installed application is sandboxed by assigning a unique user identifier (UID). Based on this UID the Linux kernel enforces discretionary access control (DAC) on low-level resources, such as files. For instance, each application has a private directory not accessible by other applications. Moreover, each application runs in its own instance of the Dalvik Virtual Machine under the assigned UID. This sandboxing mechanism also applies to native code contained in applications. However, applications from the same vendor (identified by the signature of the application package) can request a shared UID, thus basically sharing the sandbox.

Access Control. Figure 5.4 depicts the possible communication channels and their respective access control in Android. At runtime, Android enforces mandatory access control (MAC) on ICC calls between applications. The MAC mechanism is based on *Permissions* [Goo10b] which an application must request from the user and/or the system during installation. Android already contains a set of pre-defined permissions for the system services [Goo10b], but applications can also define new, custom permissions to protect their own interfaces. A reference monitor in the Android middleware checks if an application holds the necessary permissions to perform a

[4] `https://market.android.com/`
[5] Binder in Android is a reduced and custom implementation of OpenBinder[Pal05]

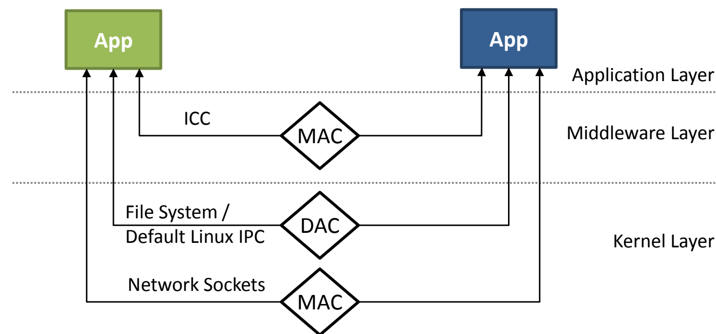


Figure 5.4: Communication channels and respective access control mechanisms in Android.

certain protected action, for instance, to bind to a protected service or to start a protected activity of another application.

On the file system, apps can decide if their files are stored in their private directory, and are thus not accessible by any other application, or in a system-wide read-/writable location. Default Linux Inter-Process Communication, for instance, Unix domain sockets or pipes, can be created with certain modes that make them accessible by other processes. The creator of such sockets/pipes decides on the mode. Thus, both file system and default IPC are under discretionary access control.

Some permissions are mapped to Linux kernel group IDs, e.g., the Internet permission, thereby relying on Linux to prevent unprivileged applications from performing privileged actions (e.g., creating Internet sockets, accessing sensitive information stored on external storage). However, since every application gets this permission granted (or not) at install time and the Android system henceforth enforces this decision, this falls under mandatory access control.

5.2.3 Problem Description and Model

We consider a corporate scenario which involves the following parties: (i) an enterprise (a company), (ii) a device (a smartphone), and (iii) an employee (the smartphone user). The enterprise issues mobile devices to its employees. The employees use their device for business related tasks, e.g., accessing the corporate network, loading and storing confidential documents, or organizing business contacts in an address book. To perform these tasks, the enterprise either deploys proprietary software, e.g., a custom VPN client including the necessary authentication credentials, on the device or provides a company-internal service, e.g., enterprise app market, from which employees can download and install those apps.

In this scenario, the enterprise is an additional stakeholder on the employees' devices and requires the protection of its delivered assets (software and data). Corporate assets may be compromised, e.g., when the user installs applications from untrusted public sources. Moreover, the employee accesses the enterprise internal network from his device and thus malware can potentially spread from the device into the corporate network.

A straightforward solution would be to prohibit any non-corporate app on the device (as proposed by, e.g., [DGLI10]). However, this is counter-intuitive to the idea of a *smartphone* and might even tempt employees to circumvent or disable this too restrictive security policy, e.g., by rooting the device. The default Android security mechanisms and recent extensions, on the other hand, are insufficient to provide enough isolation of untrusted applications and thus to protect the enterprise's assets. Virtualization can provide strong isolation between trusted and untrusted domains, but noticeably use up the battery life of the device, because major parts of the software

stack are duplicated and executed in parallel in currently available virtualization solutions.

Consequently, an isolation solution is required, which preserves the battery life by minimizing the computational overhead and still provides isolation of corporate assets from untrusted applications.

Adversary and Trust Model

We consider *software* attacks launched by the adversary on the device at different layers of the Android software stack. The adversary's goal is to get access to corporate assets, e.g., to steal confidential data, to compromise corporate applications or to infiltrate the corporate network. The adversary can penetrate the system by injecting malware (e.g., by spreading it through the Android Market) or by exploiting vulnerabilities of benign applications. Malicious applications may either be granted by the user the privileges to access sensitive resources (see Gemini [Loo10]) or try to extend their privileges by launching privilege escalation attacks [DDSW10, SZZ+11, LRW10, Goo10a, Nil10, Obe10, Bra11].

We assume that the enterprise is trusted, and that the employee is not malicious, i.e., he does not intendedly leak the assets stored on his device. However, he is prone to security-critical errors, such as installing malware or disabling security features of his device.

The device is generally untrusted, but has a trusted computing base (TCB) which is responsible for security enforcement on the platform. The TCB is trusted by the enterprise.

Objectives and Requirements

We require the integrity and confidentiality of the corporate assets on the device, while preserving the usability. Furthermore, we require that the integrity of the corporate network will be preserved even if malware infiltrated employees' devices. With respect to these objectives, we define the following requirements:

- *Isolation.* Corporate assets must be isolated in a separate domain from untrusted data and software, and any communication between different domains must be prevented. In particular, the following communication channels must be considered: IPC channels, the file system and socket connections. In addition, potential malware on the device must be prevented from accessing the corporate network.
- *Access control.* Access of applications to assets stored on the device must be controlled by the enterprise by means of access control rules defined in a security policy, e.g., a new application can be installed in the corporate domain only if the policy states that it is trusted.
- *Legacy and Transparency.* To preserve the smartphone's functionality, we require our solution to be compatible to the default Android OS and to 3rd party applications. Further, it should be transparent to the employee.
- *Low overhead.* With respect to the constrained resources of smartphones, in particular, the battery-life, our solution has to be lightweight.

Assumptions

We consider the underlying Linux kernel and the Android middleware as Trusted Computing Base (TCB), and assume that they have not been maliciously designed. Moreover, we assume the availability of mechanisms on the platform to guarantee integrity of the TCB (i.e., OS and

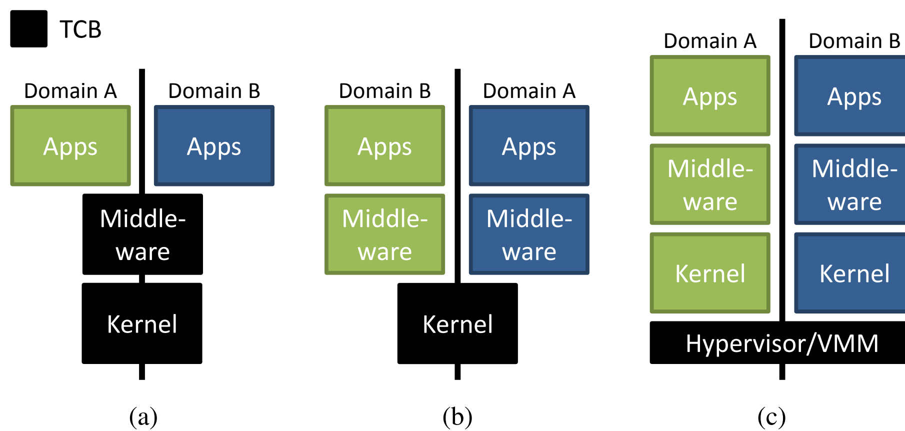


Figure 5.5: Approaches to isolation: (a) *TrustDroid*; (b) OS-level virtualization; (c) Hypervisor/VMM

firmware) on the device. For instance, this can be achieved with secure boot which is a feature of off-the-shelf hardware (e.g., M-Shield [SA05] and ARM TustZone [ARM]) or software security extensions for embedded devices (e.g., a Mobile Trusted Module (MTM) [Tru]).

5.2.4 Design of TrustDroid

In this section we describe the design and architecture of *TrustDroid*. The main idea is to group applications in isolated domains. With isolation we mean that applications in different domains are prevented from communicating with each other via ICC, Linux IPC, the file system, or a local network connection. Figure 5.5 illustrates different approaches to achieve isolation: (a) the approach taken by *TrustDroid*, which extends Android's middleware and kernel with mandatory access control; (b) OS-level virtualization, where each domain has its own middleware; (c) isolation enforced via a hypervisor and virtual machines, where each domain contains the full Android software stack. Comparing these approaches, *TrustDroid* has on the one hand the largest TCB, but on the other hand it is the most lightweight one, since it does not duplicate the Android software stack, and still provides good isolation, as we will argue in the remainder of this section.

Our extensions to the Android OS are presented in Figure 5.6. The middleware extensions consist of several components: Policy Manager, Firewall Manager, Kernel MAC Manager, an additional MAC for Inter Component Communication (ICC), and finally a modified Package Installer. The Policy Manager is responsible for determining the color for each installed application, for issuing the corresponding policies to enforce the isolation between different colors, and to enforce these policies on any kind of ICC. The Firewall Manager and the kernel-level MAC Manager are instructed by the Policy Manager to apply the corresponding rules to enforce the isolation on the network layer and the kernel layer. To enforce the latter, *TrustDroid* relies on default features of the Linux kernel, which can also be activated in Android's Linux kernel: a firewall (FW) and a Kernel-level MAC mechanism. Since we modified the Android middleware a company which wants to make use of *TrustDroid* has to roll out a customized version of Android to their employees' smartphones.

In the subsequent sections, we elaborate in more detail on the components of *TrustDroid* that enforce domain isolation.

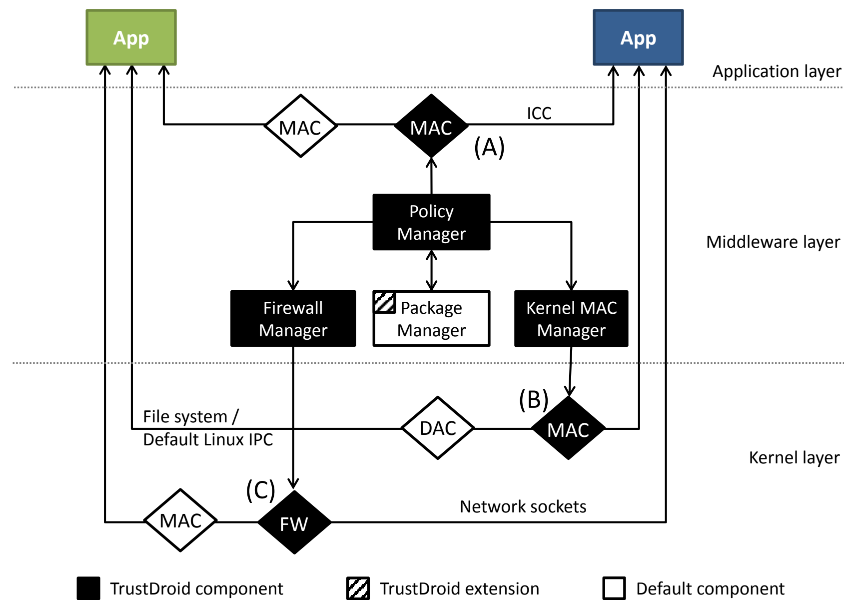


Figure 5.6: *TrustDroid* architecture with isolation of different colors **(A)** in the middleware, **(B)** at the file system/default Linux IPC level, and **(C)** at the network level.

Policy Manager

In this section we explain the Policy Manager component of *TrustDroid* and elaborate in more detail on how it colors applications and enforces domain isolation in the middleware.

Application Coloring The fundamental step in our architecture to isolate apps is to assign each app a trust level, i.e., to *color* them. In *TrustDroid*, we assume three trust levels for applications: 1) pre-installed *system* apps, which include System Content Providers and Services (cf. Section 5.2.2); 2) *trusted* third party apps provided by the enterprise; 3) *untrusted* third party apps, which are retrieved from public sources such as the Android Market. While trusted and untrusted apps must be isolated from each other, system applications usually have to be accessible by all installed applications in order to preserve correct functionality of those applications and sustain both transparency and legacy compliance of our solution.

In *TrustDroid*, system apps (i.e., pre-installed apps) are already colored during platform setup in accordance with the enterprise's security policies. Additionally installed third party apps are colored upon installation, before any code of the app is executed. In Android, the PackageManager is responsible for the installation of new applications and in *TrustDroid* we extended it to interface with the Policy Manager, such that the Policy Manager can determine the color of the new app, issue the necessary rules for its isolation in the middleware, and instruct the Firewall Manager and Kernel MAC Manager to enforce the corresponding policies on the lower levels.

Determining the color of an app can be based on various mechanisms. For instance, it can be based on a list of application hashes for each color or based on the information available about the new app, such as developer signature or requested permissions. For *TrustDroid* we opted for a certification based approach. The Policy Manager recognizes a special certificate (issued by the enterprise), which is optionally contained in the application package of apps. Based on this certificate, *TrustDroid*'s PolicyManager verifies the authenticity and integrity of the new app. Moreover, the certificate may define a platform state, (e.g., the already installed applications), in

which the certificate is only valid. A trusted service on the device is responsible for verifying these certificates. This service also measures the platform state, provides secure storage for the certificate verification keys, and maintains the verification key hierarchy such that only the enterprise can issue valid certificates. We use a Mobile Trusted Module (MTM) and as certificate format Remote Integrity Metrics (RIM) certificates, both defined by the Trusted Computing Group (TCG) [Tru]. We refer to Section 5.2.5 for more details on how we use and implement those.

If such a RIM certificate is present, it must be successfully verified to continue the installation, i.e., the certificate must have been issued by the enterprise, the application package's integrity must be verified, and the platform state defined in the certificate must be fulfilled. Otherwise, the installation is aborted. In case of a successful verification, the certificate determines the color of the new app. In our corporate scenario with only two domains, successfully verified apps are in the *trusted* corporate domain. If no certificate is found, the app is by default colored as *untrusted*. This applies, for example, to all Android Market apps.

Alternatively, the certificates can already be pre-installed on the phone and the Policy Manager checks for a pre-installed certificate corresponding to the new app.

Generating the RIM certificates for applications requires a corresponding PKI inside the company. However, almost all companies today have integrated a PKI into their IT infrastructure. For the initial setup of the mobile devices the certificates are generated and integrated once for every pre-installed trusted application. By integrating the deployment of RIM certificates into a mobile device management solution or a company internal app market the process of app-certification can be automated for updates or applications installed later.

Inter Component Communication As described in Section 5.2.2, Android uses Inter Component Communication as the primary method of communication between apps. Although ICC is technically based on IPC at the kernel level, it can be seen as a logical connection in the middleware. Thus, enforcement of isolation in the middleware has to be implemented based on access control on ICC.

In general, one can distinguish different kinds of ICC which can be used by apps for communication.

Direct ICC. The most obvious way for apps to communicate via ICC is to establish direct communication links. For instance, an app could send an Intent to another app, connect to its service, or query the content provider of another app. The *TrustDroid* MAC on ICC detects this communication and prevents it in case the sender and receiver app of the ICC have different colors. It thereby acts as an additional MAC besides the default access control of Android. As mentioned in Section 5.2.4, system apps form an exception and direct ICC is not prohibited if either sender or receiver of the ICC is a system app.

If two applications depend on each other, it is the responsibility of the certificate issuer, i.e, the enterprise in our scenario, to take care that these applications are in the same domain and to resolve any conflict in case the applications should have different trust levels according to the issuer's security policy.

Broadcast Intents. Besides the obvious direct ICC, apps are also able to send broadcast Intents, which are delivered to all registered receivers. Similar to the approaches taken in [OMEM09] and [BDD+11a], *TrustDroid* filters out all receivers of a broadcast which have a different color than the sender before the broadcast is delivered. Again, system apps are an exception and are not filtered from the receivers list.

System Content Providers. A mechanism for apps from different domains to communicate

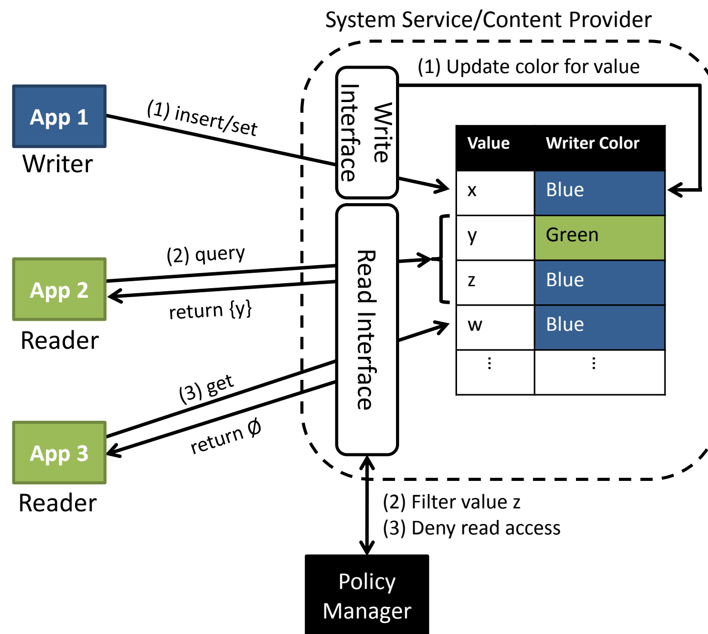


Figure 5.7: Coloring of data (1) and isolation of data from different colors in the (2) System Content Providers and (3) System Service.

indirectly is to share data in System Content Providers, such as the Contacts database, the Clipboard, or the Calendar. The ICC call to read data from such a provider does not give any information on the origin of the data, i.e., who wrote the data to the provider. We achieve domain isolation for System Providers as depicted in Figure 5.7. *TrustDroid* extends the System Content Providers such that all data is *colored* with the color of its originator app (Step (1) in Figure 5.7). Upon read access to a provider, all data colored differently than the reader app is filtered from the response (Step (2) in Figure 5.7).

System Service Providers. A covert method for apps to communicate are System Service Providers, such as the Audio Manager [SZZ+11]. However, in our adversary model (cf. Section 5.2.3), we assume that corporate apps are trusted and not malicious and thus no sender for such a covert channel exists in the trusted corporate domain. Nevertheless, data might leak via System Services from the trusted to the untrusted domain and thus isolation should be enforced here as well. Thus, as for the System Content Providers, *TrustDroid* tags the read-/writable data values of the System Service Providers with the color of the last app updating them, e.g., when setting the volume level (Step (1) in Figure 5.7). Read access to these values is denied in case the colors of the reader and the data differ (Step (3) in Figure 5.7). Although this approach does not prevent this kind of covert channel per se, it drastically reduces its bandwidth to 1-bit, because the reader only gains information if his corresponding writer changed the value or not.

Alternatively, *TrustDroid* could return a pseudo or null value instead of denying the read access. However, in contrast to System Content Providers, on which a read operation by design might return an empty response, System Services are expected to return the requested value. Thus, returning a pseudo or null value may crash the calling app, or even cause more severe harm to the hardware or user, for instance, if the app reads a very low volume level when instead the real volume level is very high.

Kernel MAC Manager

The Kernel MAC Manager is responsible for communication with and management of the MAC mechanism provided by the underlying Linux kernel. Such mechanisms, like SELinux [Nat] or TOMOYO Linux [HHT04], are already by default features of the Linux kernel and provide mandatory access control on various aspects of the OS, including the file system and the Inter-Process Communication. Thus, by employing such a MAC mechanism, *TrustDroid* achieves the isolation of domains on file system and IPC level. More explicitly, we create a MAC domain for each color and each app is added to the domain of its color upon installation. The Policy Manager instructs the Kernel MAC Manager to which domain a new application has to be added and the Kernel MAC Manager translates this instruction into low-level rules, which are inserted into the MAC mechanism and which define the isolation of domains at file system and IPC level.

File System. The file system is a further communication channel for applications. Apps are able to share files system-wide, by writing them to a system-wide readable location. Thus, a sending application can write such a file and a receiving application would simply read the same file. The mandatory access control mechanism enforces isolation on the file system in addition to the discretionary access control applied by default. *TrustDroid* applies rules, which enforce that a system-wide readable file can be only read by a another app of the same color as the writer. Thus, if an app declares a file system-wide readable, it is shared only within the domain of the writer.

Moreover, mandatory access control can, with corresponding policies, even be used to constrain the superuser account. Hence, even if a malicious application gains superuser privileges, it's file system scope could be limited to it's domain.

Inter-Process Communication. To prevent any communication of apps through Linux IPC (e.g., pipes, sockets, messages, or shared memory), *TrustDroid* leverages the same domains already established for the file system access control. Thus, apps are not able to establish IPC with differently colored apps. However, system applications form an exception, since denial of communication to system apps renders any application dysfunctional.

Potentially, ICC, which is based on Binder (and hence on shared memory based IPC), can be essentially addressed with kernel level MAC. However, in this case the policy enforcement would be limited to direct ICC between apps and would miss indirect communications, e.g., via Content Providers or Broadcast Intents. In this sense, MAC on shared memory based IPC is supplementary to the ICC MAC, because it enforces policies even in case (malicious) applications manage to disable the ICC MAC.

Firewall Manager

A further channel that has to be considered is Internet networking, i.e., network sockets used for communication via Internet protocols (such as TCP/IP). Based on these sockets applications are able to communicate with remote hosts, but also with other applications on the same platform. Thus, isolation with respect to the corporate smartphone scenario has to take both local and remote communication into consideration. To enforce isolation, *TrustDroid* employs a firewall to modify or block Internet socket based communication. Managing the firewall rules based on the policies from the Policy Manager is the responsibility of the Firewall Manager component.

Local Isolation. To locally enforce isolation between domains on the platform, *TrustDroid* prohibits any communication from a local network socket of an untrusted application to another local network socket. Although, on first glance, this might appear over-restrictive, it is a reasonable enforcement, because applications residing on the same platform usually employ

lightweight ICC to communicate instead of network channels.

Alternatively, network communication within each domain could be allowed and only cross-domain traffic be prevented. However, this would require that the Firewall Manager knows which Internet socket belongs to which application and which address has been assigned to each socket.

Remote Isolation and Context-Awareness. Enforcing isolation between domains on the network traffic between the platform and remote hosts, e.g., web-servers, is a harder problem than local enforcement. All data that leaves the phone is beyond the policy enforcement capabilities of *TrustDroid*. For instance, applications in different domains could exchange data via a remote web-service. Moreover, with respect to the corporate scenario, one must consider that malware on the phone might spread into the corporate network once the phone connects to it.

To address the former problem, *TrustDroid* uses a firewall that is able to tag (*color*) the network traffic, e.g., VLAN. If the network infrastructure supports the isolation of traffic, for instance in Trusted Virtual Domains (TVDs) [DEK+09], the policy enforcement is extended beyond the mobile platform.

To address the latter problem, *TrustDroid* employs context-aware policy enforcement on outgoing traffic. The context can be composed of various factors, for instance, the absence/presence of a user, the temperature of the device, or the network state. In *TrustDroid*, the context means the physical location of the device and the network the device is connected to. Each context definition is associated with a policy that defines how to proceed with the network traffic of untrusted applications, e.g., blocking all traffic or manipulating it in a particular way. Thus, if the platform is physically on corporate premises or connected to the corporate network, all untrusted, non-corporate apps could be denied network access or their traffic can be manipulated, for instance, to reroute it to a security proxy or an isolated guest network.

5.2.5 Implementation and Evaluation

Implementation

We implemented *TrustDroid* based on the Android 2.2.1 sources and the Android Linux kernel version 2.6.32.

We extended the default Android ActivityManager with a new component for the *TrustDroid* Policy Manager and the additional policy enforcement on ICC. We implemented the Firewall Manager and Kernel MAC Manager as new packages in the system services in the middleware.

The Policy Manager contains a minimal native MTM implementation, which is loaded as a shared library and called via the Java Native Interface (JNI). Alternatively, *TrustDroid* could use more sophisticated and secure MTM implementations as proposed in [EB09, Win08, ZAS07]. The MTM provides the means to verify Remote Integrity Metrics (RIM) certificates, to measure the software state of the platform, and to securely maintain monotonic counters.

Figure 5.8 illustrates the control flow for coloring a new application during installation and mapping the policies from the Policy Manager to the kernel and network level. Solid lines illustrate the control flow in case the application package contains a RIM certificate. Dashed lines show the deviation from this flow in case no RIM certificate is included in the package.

Application Coloring. To color new apps during installation, we extended the Android PackageManager to call the *TrustDroid* Policy Manager during the early installation procedure (step 1 in Figure 5.8) in order to verify the certificate potentially included in the application package (denoted APK) and determine the color of the new app. Therefore, the certificate is first extracted from the APK (steps 2 and 3a) and the resulting APK is verified with this certificate (steps 4a and 5a). In case the verification fails, the installation is aborted by throwing a Security

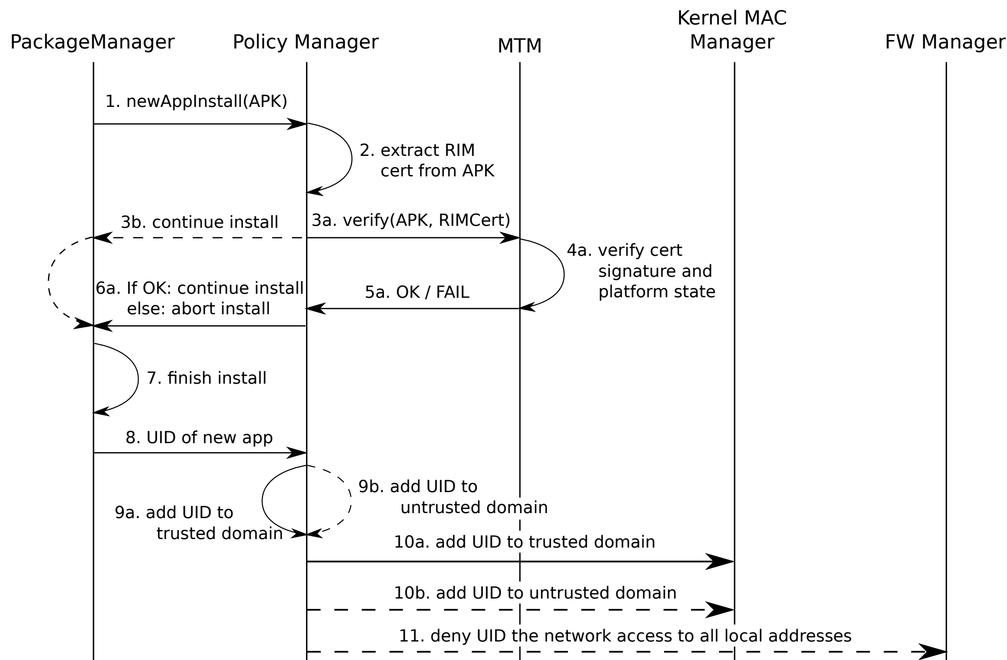


Figure 5.8: Control flow for the installation of a new application in case the installation package contains a RIM certificate (solid lines). If no RIM certificate is included in the package, this flow deviates (dashed lines).

Exception back to the PackageManager (step 6a). In case no RIM certificate is contained in the APK, the installation proceeds normally (step 3b). If the installation is continued and succeeds (step 7), a second remote call from the PackageManager informs the Policy Manager of this success (step 8) and thus triggers the issuing of corresponding policies to isolate the new app from other apps with a different color at ICC level (steps 9a and 9b), at file system and IPC level (steps 10a and 10b), and the network level (step 11).

RIM Certificates and life-cycle management. As certificate format, we chose the RIM certificates as defined in the TCG Mobile Trusted Module (MTM) specifications [Tru]. In addition to the authenticity and integrity verification provided by other certificate standards such as X.509, RIM certificates additionally provide valuable features for a trusted life-cycle management. RIM certificates define a platform state in which the certificate is valid. This state is composed of monotonic counter values of the MTM and the measured software state. If either the counter value or software state defined in a RIM certificate mismatches the corresponding value of the MTM, the certificate verification fails. RIM certificates are signed with so-called *verification keys*. These verification keys form a key hierarchy, whose root key can be exclusively controlled by a particular entity, the enterprise in our scenario. Thus, only the enterprise is able to create valid RIM certificates for it's employees' devices and thus only successfully certified apps are considered as trusted. Examples for MTM-based enhanced life-cycle management of apps are the prevention of version rollback attacks based on monotonic MTM counters, the binding of the installation to a certain platform state, or the trustworthy reporting of the software state, i.e., installed applications.

To certify APKs, we developed a small tool written in Java and that makes use of the jTSS[6].

Network, Default IPC, and File System Isolation. To implement isolation at network,

[6] http://trustedjava.sourceforge.net/

default Linux IPC, and file system level, our implementation employs *netfilter*[7] present in the kernel and *TOMOYO Linux*[8] v1.8 available as a kernel patch. To maintain them from the Firewall Manager and Kernel MAC Manager, respectively, we cross-compiled and adapted the user-space tools *iptables* and *ccs-tools*. The former is used to administrate netfilter and the latter for TOMOYO Linux policy management.

In *TrustDroid*, we created two TOMOYO Linux domains for third party apps, *trusted* and *untrusted*, and policies that isolate these domains on file system and default Linux IPC level. Upon installation of a new application, the UID of the new application is inserted into either one of those domains (steps 10a and 10b in Figure 5.8). A third domain for system apps is accessible by both the trusted and untrusted domains.

By default, *TrustDroid* denies all untrusted applications network communication to local addresses on the phone and the Policy Manager instructs the FW Manager to enforce this isolation also for newly installed untrusted applications (step 11 in Figure 5.8). Thus, any local network communication between trusted apps is isolated from untrusted apps.

A particular technical challenge was the adaption of the TOMOYO Linux user-space programs, in order to be able to maintain the TOMOYO Linux policies locally on the device. Recent documentation for TOMOYO Linux on the Android emulator describes the policy administration from a remote host instead of locally on the device and thus required certain adaption for *TrustDroid*.

Although TOMOYO Linux in version 1.8 provides MAC for Internet sockets as well, thus the means to exclude applications with fine-grained policies (e.g., UID, port or IP address) from Internet access, we opted for netfilter for two major reasons: 1) Unexpectedly denying access to sockets is much more likely to crash affected applications, in contrast to simply blocking the outgoing traffic and thus faking a disabled network connection; 2) netfilter provides much more flexibility than simply access control, e.g., manipulating or tagging network traffic for advanced security infrastructures such as TVDs or security proxies.

An alternative building block to TOMOYO Linux would be SELinux, which is based on extended file attributes and thus provides a more intuitive solution for domain isolation at the file system level. On the other hand, it is more complex to administer than TOMOYO Linux and requires modifications to the default Android file system, because the default file system does not support extended file attributes.

Context-Awareness. A context in our current implementation is simply the definition of a WiFi state and/or location. For instance, it could be the SSID of the wireless network, the MAC address of the access point, a certain latitude/longitude range, or proximity to a certain location.

To implement the context-aware management of the netfilter rules (cf. Section 5.2.4), the current Firewall Manager uses two state listeners – one for changes of the WiFi state and one for updates on the location. The former is simply a receiver for notification broadcasts about the changed Wifi state. The latter is an LocationListener thread registered at the LocationManager. In case one of the two listeners is triggered, the new state is compared with the installed contexts and the policies of any matched context are activated. The active policies of contexts that are not fulfilled anymore are revoked.

Middleware Isolation. The implementation of the additional policy enforcement on ICC is based on the *XManDroid* framework presented in [BDD+11a], which provides the necessary hooks in the Android middleware to easily implement policy enforcement on direct ICC, broadcast Intents, and channels via System Content Providers and Services.

[7] http://www.netfilter.org/
[8] http://tomoyo.sourceforge.jp/

To prevent direct ICC between applications with different colors, we wrapped the *checkPermission* function of the ActivityManager, which is called every time a new ICC channel shall be established. If the default MAC of Android permits the new ICC, *TrustDroid* performs an additional check to compare the colors of the caller and callee. On mismatch, the previous decision is overruled and the ICC denied. To prevent data flow between different domains via Broadcast Intents, our implementation is similar to [OMEM09] and implements hooks in the broadcast management in the ActivityManager to filter out all receivers of a broadcast that do not have the same color as the sender. As described in Section 5.2.4, we extended the interfaces of System Content Providers, such as the Settings or Contacts, and of the System Services, such as the Audio Manager, to color data upon write access and filter data/deny access upon read access.

Moreover, the PackageManagerService allows applications to iterate over the information of installed packages, e.g., to find a specific application that might provide supplementary services. In *TrustDroid* we extended this functionality with additional filters, such that applications can only receive a list of and information about applications of the same color or about system applications.

Evaluation

We evaluated the performance overhead and memory footprint of our extensions to the middleware with 50 apps from the Android Market, categorized in two domains (plus one domain for system apps). On average our additional policy enforcement on ICC added $0.170ms$ to the decision process on whether or not an ICC is allowed (default Android requires on average $0.184ms$). The standard deviation in this case was $1.910ms$, caused by high system-load due to heavy multi-threading during some measurements. The verification of the RIM certificate during the installation of new packages required on average $869.750ms$ with a standard deviation of $645.313ms$. The average memory footprint of our extensions to the Android system server was 348.2 KB with a standard deviation of 200.8 KB, which is comparatively small to the default footprint of approximately 2 MB.

In our prototype implementation, the policy file of TOMOYO consumes on average a little more than 200 KB of memory. The policy file includes access control rules for file system and standard IPC mechanisms e.g. communication based on Unix domain sockets.

5.2.6 Discussion

In this section we discuss the security of *TrustDroid* and highlight possible extensions.

Security Considerations

The main goal of *TrustDroid* is to provide an efficient and practical means to enforce domain isolation on Android. In particular, *TrustDroid* isolates applications by their respective trust levels, meaning that applications have no means to communicate with each other if their trust levels mismatch. Our requirement of access control is achieved by including certificates into an application package. Further, to control as many communication channels as possible, *TrustDroid* targets different layers of the Android software stack. First, IPC traffic (in the middleware and the kernel) is completely mediated by *TrustDroid* and is target to domain policies. Hence, malicious applications cannot use interfaces of applications belonging to other domains, even if the interfaces are exposed as public. Thereby *TrustDroid* prevents privilege escalation attacks from affecting other domains. Second, *TrustDroid* prevents unauthorized

data access, by performing fine-grained data filtering on application data and data stored in common databases (SMS, Contacts, etc.). In particular, this prevents malicious applications from reading data of the corporate domain, as long as the malicious application has not been issued by the enterprise itself, which is excluded in our Adversary Model (cf. Section 5.2.3). Third, *TrustDroid* successfully mitigates the impact of kernel-exploits, because our TOMOYO policies prevent an adversary from accessing files of another domain. Finally, communication over socket connections are constrained to the domain boundary.

Although our approach is lightweight and practical, it does not provide the same degree of isolation as full-virtualization would do. In particular, *TrustDroid* only mitigates kernel-level attacks by restricting access to the file-system, but in general, it cannot prevent an adversary from compromising the Trusted Computing Base (TCB), which for *TrustDroid* includes the underlying Linux kernel and the Android middleware (see Section 5.2.3). In practice, static integrity of the TCB can be insured by means of secure boot. However, the TCB is still vulnerable to runtime attacks subsequent to a secure boot. Solving this problem is orthogonal to the solution presented in this section.

The primary cause for runtime attacks on Android is the deployment of native code (shared C/C++ libraries) [Obe10]. Although Android applications are written in Java, a type-safe language, the application developers may also include (custom) native libraries via the Java Native Interface (JNI). Moreover, many native system libraries are mapped by default to the program memory space.

A straightforward countermeasure against native code attacks would be to prohibit the installation of applications that include native code. However, this is rather over restrictive and, similar to prohibiting any non-corporate app (cf. Section 5.2.3), contradicts the actual purpose of smartphones or might even tempt the phone user to break the security mechanisms in place.

Another approach to address native code attacks is Native Client [SMB+10], which provides an isolated sandbox for native code. However, this solution requires the recompilation of all available applications that contain native code.

Moreover, as argued and shown in [ZSA10], mandatory access control can also be efficiently deployed on mobile platforms to enforce isolation for the complete Linux kernel. We consider this as a valuable extension to *TrustDroid* to mitigate kernel attacks, which could easily be integrated in *TrustDroid*, since a kernel-level MAC mechanism is already a building block of our design (see Section 5.2.4).

Finally, *TrustDroid* uses a separate, accessible domain for system applications and services, which is due to the fact that all applications require these system apps to work correctly. If an adversary identifies a vulnerability in one of these applications, he may potentially circumvent domain isolation and access data not belonging to his domain. However, until today, vulnerabilities of system applications were constrained to confused deputy attacks and did not allow an adversary to access sensitive data [FWM+11]. Protecting system applications and services from being exploited is orthogonal to harden the kernel, and we aim to consider this in our future work. Alternatively, one could deploy apps in the trusted domain which offer the functionality of certain system apps (e.g., business contacts app or enterprise browser; cf. [ent]) and isolate the now redundant system apps by classifying them as untrusted.

Trusted Computing

Our *TrustDroid* design leans towards possible extensions with Trusted Computing functionality.

Currently, we leverage a Mobile Trusted Module (MTM) to validate application installation packages and to determine their color. The features of the employed RIM certificates in contrast

to established certification standards such as X.509 provide the means for an enhanced life-cycle management based on monotonic counters and the platform state, e.g., version rollback prevention. The current implementation of our MTM is simple, but more sophisticated approaches may be integrated into our current design [EB09, Win08, ZAS07].

Moreover, our design includes the fundamentals for the integration of Trusted Computing Group (TCG) mechanisms such as the attestation of the domains [NKZS10], e.g., in the context of Trusted Network Connect [Tru09], or the isolation of network traffic for infrastructures like Trusted Virtual Domains [DEK+09].

5.2.7 Related work

In this section we provide an overview of related work with respect to the establishment of domains and policy enforcement on Android.

Virtualization

A "classical" approach from the desktop/server area to establish isolated domains on the same platform is based on virtualization technologies. This approach has been ported to the mobile area [Ope, BBD+10, SSFG10]. Although virtualization provides strong isolation, it duplicates the entire Android software stack, which renders those approaches quite heavy-weight in consideration of the scarce battery life of smartphones. Possible approaches to mitigate this problem could be the automatic hibernation of VMs currently not displayed to the user or the application of a *just-enough-OS/Middleware* to minimize the resident memory footprint of domains. However, currently available mobile virtualization technology does not provide these features. In contrast, our solution is more lightweight, since the creation of a new domain simply requires the definition of a new string value and deployment of a new MTM verification key. Moreover, from our past experience with mobile virtualization technology [DDKW11], we conclude that our solution is more practical in the sense that it is more portable to new hardware, because we can re-use the provided proprietary hardware drivers, while virtualization requires new (re-implemented) drivers or an additional *driver-domain* that multiplexes the hardware between the VMs (e.g., *dom0* in Xen [HSH+08]).

Kernel-level Mandatory Access Control

Another well established mechanism, that is now being ported to the Android platform, is kernel-level mandatory access control like SELinux or TOMOYO [SFE10, DT]. These mechanisms allow, e.g., policy enforcement on processes, the file system, sockets, or IPC. In *SEIP* [ZSA10], SELinux was used to establish trusted and untrusted domains on the LiMo platform in order to protect the platform integrity against malicious third party software. The work further shows how unique features of mobile devices can be leveraged to identify the borderline between trusted/untrusted domains and to simplify the policy specification, while maintaining a high level of platform integrity. The authors of [RJ09] show how policies in the context of multiple mobile platform stakeholders can be created dynamically and present a prototype based on SELinux. Low-level mandatory access control is an essential building block in our design (see Section 5.2.4). However, it is insufficient for isolating domains because it does not consider the Android middleware system components, such as System Content Providers/Services or Broadcast Intents, as communication channels between domains (see Section 5.2.3). Without

high-level policy enforcement in the middleware, low-level MAC mechanisms can only grant/-deny applications the access to System Content Providers and Services as a whole. However, generally denying an app access to system components most likely crashes this app or at least renders it dysfunctional. Moreover, although these mechanisms allow to some extend fine-grained access control policies on the network, they do not support the manipulation of network packets like netfilter does (cf. Section 5.2.4). Nevertheless, the approach of [ZSA10] could enhance the integrity protection of our TCB (see Section 5.2.6).

Android Security Extensions

In the last few years, a number of security extensions to the Android security mechanisms have been introduced [CNC10, NKZ10, OBM10, EOM09a, EGC+10, BDD+11a]. Based on very similar incentives to *TrustDroid*, *Porscha* proposes a DRM mechanism to enforce access control on specifically tagged data, such as SMS, on the phone. However, this approach is limited to isolate data assets, but is not suitable to isolate particular (sets of) apps.

Similarly, the *TaintDroid* framework [EGC+10] tracks the propagation of tainted data from sensible sources (in program variables, files, and IPC) on the phone and detects unauthorized leakage of this data. However, it is limited to tracking data flows and does not consider control flows. Moreover, it does not enforce policies to prevent illegal data flows, but notifies the user in case an illegal flow was discovered. Nevertheless, TaintDroid could form a very valuable building block in our *TrustDroid* design to isolate data assets, if it would be extended with policy enforcement.

Both APEX [NKZ10] and CRePE [CNC10] focus on enabling and disabling functionalities and enforcing runtime constraints. While APEX provides the user with the means to selectively choose the permissions and runtime constraints (e.g., limited number of text messages per day) each application has, CRePE enables the enforcement of context-related policies of the user or a third party (e.g., disabling bluetooth discovery). In this sense, both are related to our design goal to isolate untrusted applications based on the context (cf. Section 5.2.4) or protect data assets in shared resources like System Content Providers. However, the enforcement described in [NKZ10] and [CNC10] is too coarse-grained. For instance, networking would be disabled for all applications, not just particular ones, or not only the access to certain data but to the entire Content Provider would be denied to selected applications.

Saint [OMEM09] introduces a fine-grained, context-aware access control model to enable developers to install policies to protect the interfaces of their apps. Although Saint could, with a corresponding system centric policy, provide the isolation of apps on direct and broadcast ICC, it can not prevent indirect communication via System Components (see Section 5.2.4).

XManDroid [BDD+11a] addresses the problem of ICC-based privilege escalation by colluding apps and is also able to enforce policies on ICC channels via System Components. The XManDroid framework formed the basis for our *TrustDroid* implementation, but had to be extended to enable application coloring and mapping of policies for domain isolation from the middleware onto the network and kernel level.

In general, none of these extensions provides any policy enforcement on the file system, IPC, or local Internet socket connections in order to enforce isolation of domains. However, TaintDroid with its data flow tracking mechanism has the potential to implement fine-grained policy enforcement.

5.2.8 Conclusion

Existing smartphone operating systems, such as Google Android, provide *no data and application isolation* between domains of different trust levels. In particular, there exists no efficient solution to isolate corporate and private applications and data on Android: the existing security extensions for Android only focus on one specific layer of the Android software stack, and hence, do not provide a general and system-wide solution for isolation.

In this section we present *TrustDroid*, a framework which provides *practical and lightweight domain isolation* on Android, i.e., it does not affect the battery life-time significantly, requires no duplication of Android's software stack, and supports a large number of domains. In contrast to existing security extensions, *TrustDroid* enforces isolation on many abstraction layers: (1) in the middleware and kernel layer to constrain IPC traffic to a single domain, and to enforce data filtering for common databases such as Contacts, (2) at the kernel layer by enforcing mandatory access control on the file system, and (3) at the network layer to regulate network traffic, e.g., denying Internet access by untrusted applications while the employee is connected to the corporate network. Our evaluation results demonstrate that our solution adds a negligible runtime overhead, and in contrast to contemporary virtualization-based approaches [Ope, BBD+10], only minimally affects the battery's life-time.

We also provide a detailed discussion on the design of *TrustDroid* and argue that *TrustDroid* can be used as a foundation for Trusted Computing enhanced concepts such as Trusted Virtual Domains (TVD), a distributed isolation concept known from the desktop world. In our future work, we aim to adopt domain isolation on the underlying Linux kernel so that an adversary can no longer exploit kernel vulnerabilities to circumvent domain isolation.

Chapter 6

Initialization and Update Mechanisms for TrustedServer

Chapter Authors:
Alexander Kasper (SRX)

In this chapter we describe the initialization and update mechanisms for a TrustedServer as introduced in Deliverables D2.1.2 Chapter 12 and D2.4.2 Section 7.5. The TrustedServer is managed solely by the TrustedObjects Manager (TOM) which communicates securely via the TrustedChannel. Trusted Computing technology is employed to secure the communication and to guarantee integrity of the TrustedServer. During initialization the TrustedServer is bound to a dedicated TOM which thenafter is responsible for the complete management and maintainence of the TrustedServer including updates. In this architecture there is no need for manual administration of the TrustedServer and thus no need for an elevated root account for an administrator of the TrustedServer. The only administration port is via the TrustedChannel to the TOM and is completely automated. Hence this architecture provides a novel trustmodel suitable for cloud computing. The (remote) administrator of the cloud infrastructure no longer has to be trusted, as she has no privileges on a TrustedServer.

6.1 Introduction to Initilization

The target of initialization is to configure the hardware to meet the requirements of a TrustedServer. The initialization is split up into the configuration of the BIOS or any other initial system, TPM (Trusted Platform Module) initialization which includes taking the ownership, preparation of the blockdevice which holds the OS, the installation of valid Platform Configuration Register (PCR) certificates and the initial connection to a TrustedObjects Manager (TOM) (cf. Deliverable D2.4.1 Section 9.2), the management component. In this first section we describe how these separate phases lead to a trustworthy server. The later sections provide some technical details how to achieve those targets and give some examples with reasonable defaults for the necessary tools. The last sections describe an update system to be able to update the TrustedServer without the need for reinstallation.

6.2 Initilization

To install a TrustedServer the TPM must be activated and support for hardware virtualization must be enabled. The TPM is used to seal the blockdevice. If Intel TXT is enabled it can be used to establish a Dynamic Root of Trust Measurement. Currently (June 2012) most systems run a BIOS to initialize the hardware. If an Unified Extensible Firmware Interface (UEFI) is

used appropriate actions must be performed to activate the needed components. Trusted boot is designed to establish a trusted bootchain where no secure bootchain is possible. The secure boot technology, which is part of the UEFI specification, can enforce a specific secure bootchain. Durring a secure boot the signature of every binary executed durring boot is verified agains a CA before the control of the bootprocess is handed over to the binary. In a trusted boot environment the binary executed durring boot are measured. As one of the final commands the TPM decrypts the harddisc decryptionkey if the measured values matches predefined values.

During the setup of a TrustedServer three blockdevices are necessary. The first blockdevice provides storage for the configuration of the bootloader and holds any additional stages needed by the bootloaderchain. The second blockdevice holds the kernel of the operating system and a minimal OS, or in general the code which is executed to start and run the TrustedServer OS. The minimal system is used to unseal the third blockdevice. The third blockdevice holds the entire system which is furthermore separated into three sections. Those sections are configuration, operating system and compartment data (see Figure 6.1)

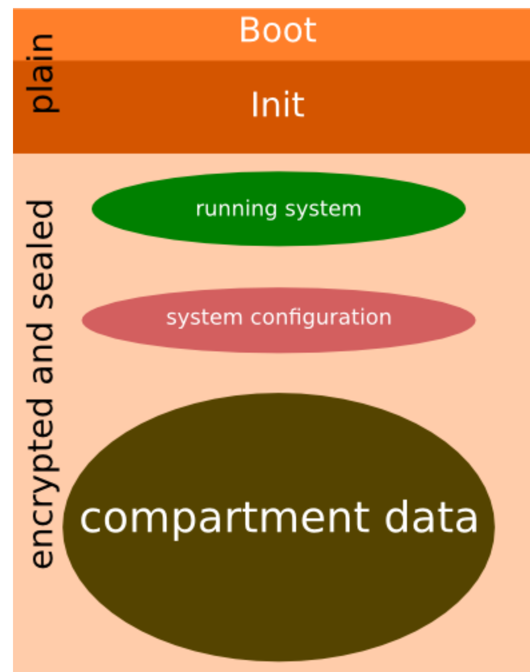


Figure 6.1: High-Level disk layout of a TrustedServer

The TrustedServer generates an unique update-, identity- and a TLS-keys. The identity key is used for remote attestation. The TLS-key is used to generate TLS-client certificates which are signed with the identity key. The update key is used to decrypt updates for the operation system. The public parts of the update key and the identity key are transmitted to the TOM. Finaly PCR-Certificates - which are signed by a configuration CA - must be provided by the initialization process. On the lower level the Trusted Server is managed by a dedicated system account. This system account is not able to login via remote or local terminal sessions. No user or admin account must have the possibility to login to the Trusted Server. The only way to manage and administrate the Trusted Server is via TOM.

6.3 HDD Layout

To provide different blockdevices to the kernel we structure the harddrive disk (HDD) into partitions. Every HDD in a TrustedServer does have three partitions. The partition table layout follows the MS-DOS layout. It is also possible to use the GPT layout of the EFI-Specification. The MS-DOS partion table layout limits the accessible storage to 2 terrabytes. The unit used to describe size or offset of a partition is megabytes. One megabyte can be expressed as 2048 blocks of 512 bytes or as 256 blocks of 4096 bytes. Using one megabyte as smallest unit leads to the opportunity to treat drives with different physical blocksizes equally. The first partition must start at megabyte 1. The offset of 1 megabyte from the beginning of the HDD is needed to provide some storage for the first stage of the bootloader. To install the MS-DOS partition layout the following command can be use.

```
parted /dev/sda mktable msdos
```

For a GPT layout one can use

```
parted /dev/sda mktable gpt
```

6.3.1 Bootloader partition

The bootloader partition holds the configuration and the second stage of the bootloader and is expected to change not very frequently. The bootloader partition can be small. This partition is not needed during normal operation of the TrustedServer. The filesystem on this partition can be very simple. There is no need for filesystem features like journaling, de-duplication or access control mechanisms For creating the bootpartition we use following commands.

```
parted -a optimal -s /dev/sda mkpart primary 1 11
```

It is also necessary to set the bootflag for this partition.

```
parted -a optimal -s /dev/sda set 1 boot on
```

The kernel must be informed to reread the partition table.

```
sfdisk --re-read /dev/sda
```

Finally we need to create a filesystem on that new partition.

```
mkfs.ext2 /dev/sda1
```

6.3.2 Init partition

The "Init" partition holds a minimal initial system and the kernel which is used during the runtime of the TrustedServer. This separation from the bootloader is used to establish some advanced full disc encryption technologies and to seal and unseal the third partition. The "Init" partition with kernel and initrd must be referenced by the configuration of the bootloader. The filesystem on this partition can be very simple - eg without journaling, without deduplication and even without any access control mechanism. The setup of that partition can be established with the following comandlines.

```
parted -a optimal -s /dev/sda mkpart primary 11 111
sfdisk --re-read /dev/sda
mkfs.ext2 /dev/sda2
```

This creates an 100 megabyte partition right after the boot partition.

6.4 HDD Encryption (FDE)

The third partition is entirely encrypted on blockdevice level. But it is also necessary to separate firmware - e.g. the operating system - the configuration and the compartment data. To achieve this logical volumes inside the encrypted partition are used. The entire remaining free space is used for the last partition.

```
 parted /dev/sda unit MB print free
```

Shows the remaining free size, in our example the remaining of a 160GB blockdevice.

```
parted -a optimal -s /dev/sda mkpart primary 111 161900
```

This final partition is encrypted on blockdevice level. For the encryption AES with a keysize of 128Bits in XTS mode is used. [1] [2]. The plaintext keyfile is generated via a cryptographic secure random number generator. The random number generator from the TPM can be used.

```
cryptsetup luksFormat /dev/sda3 \
    -c aes-xts-plain \
    -s 256 \
    --key-file=keyfile.plaintext
```

In the second step this encrypted partition is opened to have access to the plaintext data.

```
cryptsetup luksOpen /dev/sda3 crturaya --key-file=keyfile.plaintext
```

This creates a new blockdevice `/dev/mapper/crturaya`. To provide a different blockdevice which resists inside the encrypted blockdevice, LVM in version two or higher is used. As mentioned before three blockdevices inside the encrypted container are needed. A volumegroup containing those three volumes resp blockdevices needs to be create. First it is necessary to mark the encrypted partition as a physical member of a volumegroup.

```
pvcreate /dev/mapper/crturaya
```

Now the initialization of the volumegroup takes place.

```
vgcreate vgturaya /dev/mapper/crturaya
```

The needed logical volumes are created as follows

```
 lvcreate -n ROOT -L 10G vgturaya lvcreate -n CONFIG -L 10G vgturaya
lvcreate -n COMP -l 100%FREE vgturaya
```

[1] Finally, in January, 2010, NIST added XTS-AES in SP800-38E, recommendation for block cipher modes of operation: The XTS-AES mode for confidentiality on storage devices; The XTS-AES mode was not designed for other purposes, such as the encryption of data in transit.

[2] Luks was choosen because of the following provided features: compatiblity via standardization (TKS1 - An anti-forensic, two level, and iterated key setup scheme), secure against low entropy attacks, support for multiple keys, effective passphrase revocation

The requirements for the filesystems of the running system are higher then those of the boot process due to the contiguous access. Journaling is used to recover the system from a power lost. Access control is needed to further limit permissions to certain systemusers. Whereby the core permissions are enforced via SELinux. SELinux stores policies in the extended attributes of the filesystem. There are many filesystems available which supports all three features, but the linux ext4 filesystem was used for longterm support. Interresting features not provided by ext4 are transparent de-duplication on blocklevel and transparent compression of files. But these features are only needed for the compartment aka userdata partition. So it is possible to use Btrfs in the future.

```
mkfs.ext4 /dev/mapper/vgturaya-ROOT
mkfs.ext4 /dev/mapper/vgturaya-CONFIG
mkfs.ext4 /dev/mapper/vgturaya-COMP
```

6.5 TPM initialization

The sealing process relies on an inialized Trusted Platform Module. The TPM must be enabled, active and unhidden in the BIOS. It should not be possible to reset the TPM from inside the OS. As part of the initialization process of the TrustedServer it is assumed, that ownership of the TPM is taken. Furthermore a storage root key (SRK) must be create inside the TPM. The NVRAM should be empty. An attestation identity key is installed inside the NVRAM to remotely identify the TrustedServer. A not cryptographical secure representation of the AIK is the serial number of the TrustedServer.

6.6 Public Key Infrastructure (PKI)

The Public Key Infrastructure ensures that only a valid chain of trust can be establish. From a root CA the intermediate CA is derived. This intermediate CA is signed by the root CA and is called configuration CA. The platform certificates (see below) are verified against the root CA. As a side effect it is possible to reflect different continuous integration states with different intermediate CAs.

6.7 Platform Configuration Certificates

Platform configuration certificates are kept in extensions of the X509 structure. Those extensions are identified by OIDs. The private part of the certificates are not needed since those certificates will never be used to encrypt anything. Only the configuration and the signature will be verified. Different stages of the bootprocess are kept in different configuration certificates. The four different PCR certificates are needed for the core root of trust for measurement (CRTM), the hardware configuration, the bootloader and the operating system. As an example consider the following. The certificate for the CRTM includes the value of platform configuration register with index zero. It also includes the type “crtm” as an integer encoding. To establish a chain of trust the successor type “hardware” and the desired pcr index values 1, 2, 3 are included. See Figure 6.2 for a common configuration.

NAME	TYPE	MASTER HASH	E.TYPE	E.DEST
CRTM	0	$sha1(PCR_0)$	1	1,2,3
Hardware	1	$sha1(PCR_1 + PCR_2 + PCR_3)$	2	4,8,9,12
Bootloader	2	$sha1(PCR_4 + PCR_8 + PCR_9 + PCR_{12})$	3	14
Operating System	3	$sha1(PCR_14)$		

Figure 6.2: CoData to be included in the client certificate for Key-Based TLS

It is also possible to setup a more complex PCR configuration chain as shown in Figure 6.3.

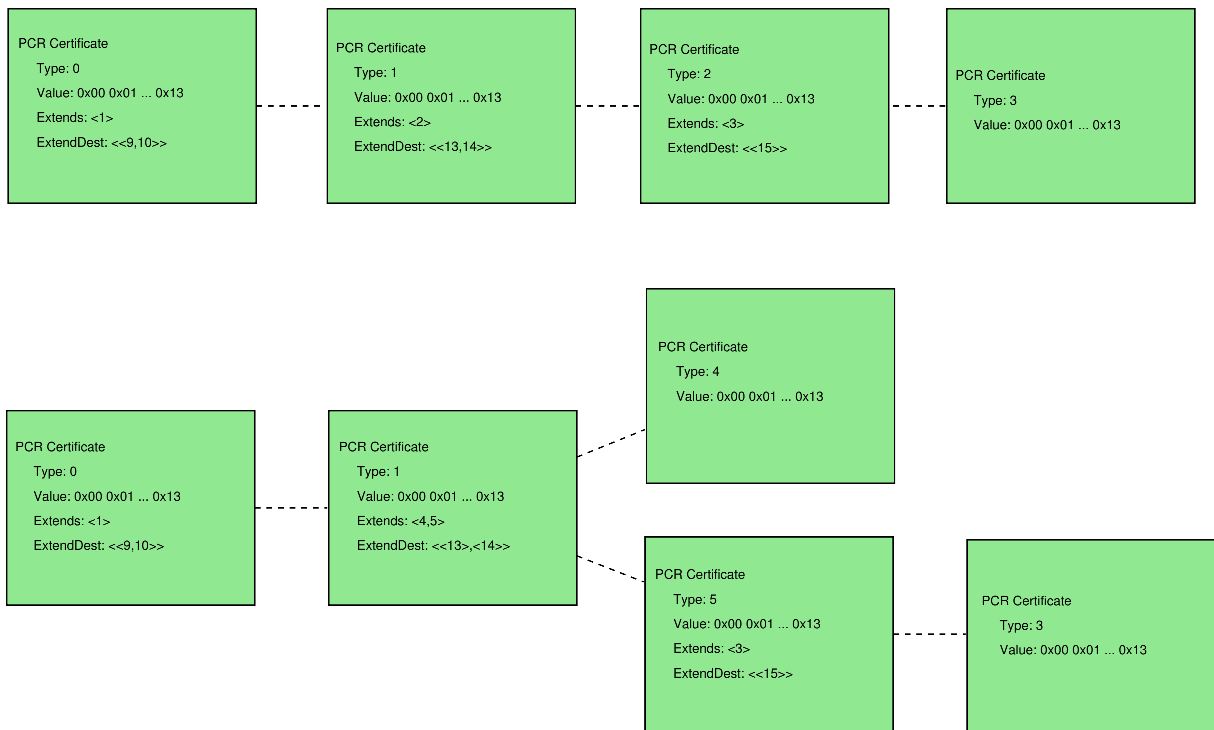


Figure 6.3: Two example PCR chains

With those platform certificates, a trusted boot and remote attestation, which takes place durring every connection attempt of the TrustedChannel to the TrustedObjectManager the trusted boot is directly linked to the TrustedChannel. For instance if the CRTM is changed the TrustedChannel can no longer be established since the TPM `quote` does no match the certificates. The certificates can not be altered under the assumption that it is hard to find a collision for the signed certificate.

TYPE	DESCRIPTION
live	Perform a live measurement (the PCR as it is at the time of sealing)
file	Measure the file given by the filename parameter (optional params offset and length)
mbr	Measure the MBR of the given block device
luks	Measure the LUKS Master Digest of the given block device or container
static	Specify the measurement value statically as a hash - This value will be extended eg sha1(oldpcr + hexvalue)
evsep	Measure an event separator (constant string 0xFF 0xFF 0xFF 0xFF)
evcall19h	Measure a call INT 19h event (constant string "Calling INT 19h")
evret19h	Measure an event separator (constant string "Returned INT 19h")
fixed	assumes the measurement and extensions has been performed elseware - this value is to be expected in the PCR

Table 6.1: Options for sealing configuration

```
# pcrIndex      measurementType        measurementParameters (optional)
0               live
17              fixed                  ffffffffffffffffffffffffffffffffffffffff
4               mbr                    /dev/sda
10              file                   tests/testFile1
14              static                 924e1ffd32bb1d4761ae653934c700d430f53713
15              evcall19h
15              evsep
15              evret19h
```

Figure 6.4: Configuration example 1 for sealing

6.8 Key sealing

The TrustedServer is able to operate without an established connection to a TrustedObjectManager for an accurately defined time. To enforce this operation window it must be ensured the TrustedServer was booted with a proper TrustedBoot. To enforce the TrustedBoot the HDD encryption key is sealed (see TPM 1.2 Specifications) against every PCR register specified in the PCR certificate chain (see above) plus the luks header of the encrypted partition (cf. section 6.4). Since the size for the sealed data is bound to the size of the SRK it is necessary to define a key encryption key (KeK). This KeK is used to encrypt the HDD encryption key. So the HDD encryption key is not limited in size. The KeK encrypts the HDD encryption key with AES128 in CBC mode. The PCR-values are partly measured during the sealing of the KeK and partly precomputed. It is possible to precompute every needed pcr-value but different hardware vendors are using different optional roms to initialize hardware. Those optional roms are messuared durring the boot process. To reflect this behavior the sealing process needs some configuration which is done in a configuration file. The table Table 6.1 shows all supported options while Figure 6.4 and Figure 6.5 provide some example configurations.

6.9 Updatekey and Updates

The updatekey is generated during the initialization but before the initial first contact to a TrustedObjectManager (TOM). The key is generated by the TPM and marked as a legacy key. During the first contact to the TOM this key is transfered via the TrustedChannel (cf. Deliverable

```
0           live
1           live
2           live
3           live
4           static          95abfad66465021b58258c6171e4c0e174faa64b
4           evsep
4           evret19h
4           mbr             /dev/sda
8           file            /boot/grub/stage2       0       512
9           file            /boot/grub/stage2       512
12          grubcfg         /boot/grub/menu.cfg
14          file            /boot/kernel.img
14          file            /boot/initrd.img
15          luks            /dev/sda3
```

Figure 6.5: Configuration example 2 for sealing

2.4.1 Section 9.3) to the TOM. Any incremental update is encrypted with a masterkey which is unique for one update. To apply an update to a specific TrustedServer the masterkey for the update is encrypted with the updatekey of the specific TrustedServer.

6.10 Attestation and Identity Key

This section is dedicated to describe the initial setup of the Trusted Channel which also provides remote attestation. In this authentication mode, the client has an *attestation identity key (AIK)* which is an identity keypair $K = (pK, sK)$. The secret key sK resides within a TPM and can be used for remote attestation and key certification. The public key pK is well known. Furthermore, the client has a *TLS keypair* $TK = (pTK, sTK)$. The secret key sTK also resides in the TPM. This keypair can be generated on-the-fly or kept statically and is not known to the server beforehand.

Prior to any communication between the server and the client, the serial number (or fingerprint) S of the public key pK must be authorized within the server (e.g. through a system administrator).

Once this step is complete, the serial number S and hence the keypair (pK, sK) is considered trustworthy by the server. Furthermore, the client has a list of certificate chains that are trusted for signing server certificates .

With these prerequisites, mutual authentication can be achieved with standard TLS using client and server certificates.

In the following, the different steps of the TLS handshake are explained in detail.

(1) Client/Server Hello To establish a connection between the client and the server, the client first issues a *TLS Client Hello* to the server. The server responds with a *TLS Server Hello*. By the TLS specification, both messages contain a certain amount of random data (28 byte randomness + 4 byte timestamp). By $nonce_S$ we denote certain amount of bits (at least 160) of the randomness included in the *server hello* message.

(2) Server Authentication The server sends its certificate to the client, including the server public key pK_S. This certificate is signed by a trusted authority and can hence be validated by the client. Furthermore, the server sends the *ServerKeyExchange* message, which in general contains all necessary parameters for a key exchange. In our case, we perform the key exchange using RSA and this message contains the necessary RSA parameters (e.g. modulus, exponent). This message is signed by the server.

(3) Client Authentication The server now requests a certificate from the client for authentication. At this point, the client obtains the secondary keypair $TK = (pTK, sTK)$ residing in the TPM and constructs an X509 certificate, including the information as described in table Figure 6.6. The certificate is *self-signed*.

(4) Handshake Completion In order to complete the handshake, the client must now encrypt a *pre-master secret* using pK_S and send it to the server. After the handshake is completed, both parties can use this to derive the *master secret*. Finally, the client sends the *CertificateVerify* message. This message contains a signature over all previous handshake messages and is performed using sTK. The client performs this message both to prove that it possesses the secret key that matches the public key shown in the client certificate and to ensure the integrity of all previous client messages.

Information	**Purpose**
Public Key pTK	Public key of X509 Certificate S (**signing**)
Public Key Info/Signature $ci(pK, pTK)$	Certification for pTK by pK (**key authentication**)
Public Key pK	Allow the server to derive the serial number S (**identity**)
Trusted Version Information	Allow the server to validate our configuration during remote attestation (PCRs)
Actual Version Information	Provide the server with information about current configuration (PCRs) (**remote attestation**).
$nonce_S$	The nonce that is used during the handshake and for quote (informative, additional **replay protection**)
sign(sTK, Hash(ClientCertificate))	Prove that the certificate originates from the client and was not altered (part of X509, **integrity**)

Figure 6.6: Data to be included in the client certificate for Key-Based TLS

By $ci(pK, pTK)$ we denote the output of a TPM_CertifyKey call to authenticate the key pTK through the known TPM key pK. Without this call, there is would be no binding between these two keys and no assurance that the key pTK is really in possession of the client.

Inclusion of remote attestation specific data (Trusted/Actual Version Information) is optional. The server may enforce the presence of this data for certain types of clients (e.g. appliances).

The presence of the $nonce_S$ within the client certificate makes the whole certificate *unique* for this session and prevents replay attacks on the certificate .

Important: The primary keypair $K = (pK, sK)$ **must** be an attestation identity key (AIK). If a `TPM_SS_RSASSAPKCS1v15_SHA1` signing key would be in use, the attacker could fake the output of TPM_Quote using this key, as there is no way to distinguish the output of TPM_Sign and TPM_Quote with this key type.

The whole process is also depicted in Figure 6.7 for an appliance that also performs remote attestation.

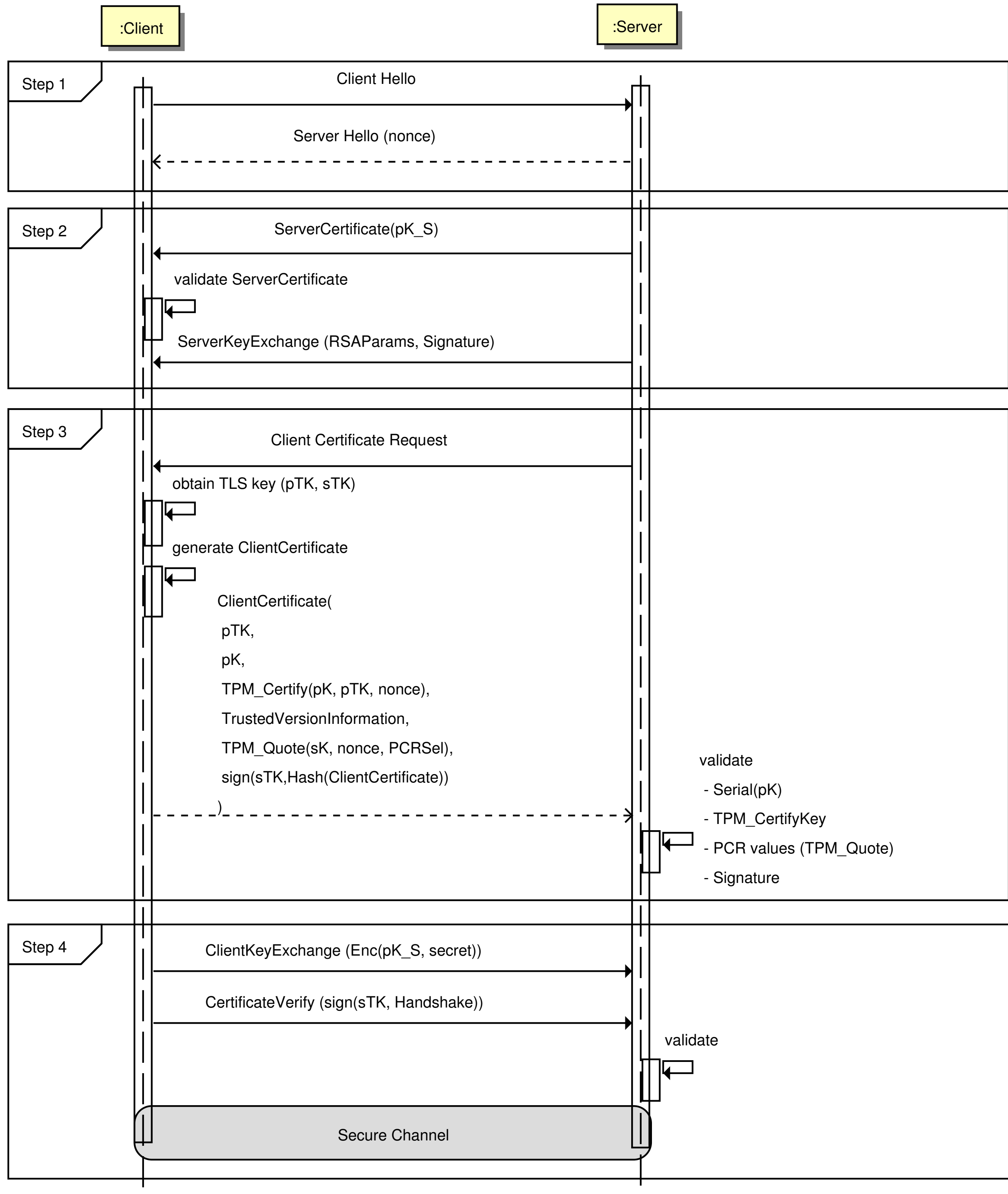


Figure 6.7: Sequence diagram for a successful key-based authentication

Chapter 7

Cheap BFT

Chapter Authors:
Rüdiger Kapitza, Johannes Behl, Klaus Stengel (TUBS),
Tobias Distler, Simon Kuhnle, Wolfgang Schröder-Preikschat (FAU),
Christian Cachin (IBM),
Seyed Vahid Mohammadi (KTH Royal Institute of Technology)

One of the main reasons why Byzantine fault-tolerant (BFT) systems are not widely used lies in their high resource consumption: $3f+1$ replicas are necessary to tolerate only f faults. Recent works have been able to reduce the minimum number of replicas to $2f+1$ by relying on a trusted subsystem that prevents a replica from making conflicting statements to other replicas without being detected. Nevertheless, having been designed with the focus on fault handling, these systems still employ a majority of replicas during normal-case operation for seemingly redundant work. Furthermore, the trusted subsystems available trade off performance for security; that is, they either achieve high throughput or they come with a small trusted computing base.

In this chapter we present CheapBFT, a BFT system that, for the first time, tolerates that *all but one* of the replicas active in normal-case operation become faulty. CheapBFT runs a composite agreement protocol and exploits passive replication to save resources; in the absence of faults, it requires that only $f+1$ replicas actively agree on client requests and execute them. In case of suspected faulty behavior, CheapBFT triggers a transition protocol that activates f extra passive replicas and brings all non-faulty replicas into a consistent state again. This approach, for example, allows the system to safely switch to another, more resilient agreement protocol. CheapBFT relies on an FPGA-based trusted subsystem for the authentication of protocol messages that provides high performance and comprises a small trusted computing base.

7.1 Introduction

In an ongoing process, conventional computing infrastructure is increasingly replaced by services accessible over the Internet. On the one hand, this development is convenient for both users and providers as availability increases while provisioning costs decrease. On the other hand, it makes our society more and more dependent on the well-functioning of these services, which becomes evident when services fail or deliver faulty results to users.

Today, the fault-tolerance techniques applied in practice are almost solely dedicated to handling crash-stop failures, for example, by employing replication. Apart from that, only specific techniques are used to selectively address the most common or most severe non-crash faults, for example, by using checksums to detect bit flips. In consequence, a wide spectrum of threats remains largely unaddressed, including software bugs, spurious hardware errors, viruses,

and intrusions. Handling such arbitrary faults in a generic fashion requires Byzantine fault tolerance (BFT).

In the past, Byzantine fault-tolerant systems have mainly been considered of theoretical interest. However, numerous research efforts in recent years have contributed to making BFT systems practical: their performance has become much better [CL02, KD04, KAD+09, DK11], the number of required replicas has been reduced [YMV+03, CNV04, WSV+11], and methods for adding diversity and for realizing intrinsically different replicas with varying attack surfaces have been introduced [Cac01, SZ05]. Therefore, a debate has been started lately on why, despite all this progress, industry is reluctant to actually exploit the available research [CMW+08, KR09]. A key outcome of this debate is that economical reasons, mainly the systems' high resource demand, prevent current BFT systems from being widely used. Based on this assessment, our work aims at building resource-efficient BFT systems.

Traditional BFT systems, like PBFT [CL02], require $3f + 1$ replicas to tolerate up to f faults. By separating request ordering (i. e., the *agreement stage*) from request processing (i. e., the *execution stage*), the number of execution replicas can be reduced to $2f + 1$ [YMV+03]. Nevertheless, $3f + 1$ replicas still need to take part in the agreement of requests. To further decrease the number of replicas, systems with a hybrid fault model have been proposed that consist of *untrusted* parts that may fail arbitrarily and *trusted* parts which are assumed to only fail by crashing [CNV04, CMSK07, RK07, LDLM09, VCB+11, VCBL10, WSV+11]. Applying this approach, virtualization-based BFT systems can be built that comprise only $f + 1$ execution replicas [WSV+11]. Other systems [CNV04, CMSK07, VCB+11, VCBL10] make use of a hybrid fault model to reduce the number of replicas at both stages to $2f + 1$ by relying on a trusted subsystem to prevent *equivocation*; that is, the ability of a replica to make conflicting statements.

Although they reduce the provisioning costs for BFT, these state-of-the-art systems have a major disadvantage: they either require a large trusted computing base, which includes the complete virtualization layer [RK07, VCBL10, WSV+11], for example, or they rely on trusted subsystems for authenticating messages, such as a trusted platform module (TPM) or a smart card [LDLM09, VCB+11]. These subsystems impose a major performance bottleneck, however. To address these issues, we present *CheapBFT*, a resource-efficient BFT system that relies on a novel FPGA-based trusted subsystem called *CASH*. Our current implementation of CASH is able to authenticate more than 17,500 messages per second and has a small trusted computing base of only about 21,500 lines of code.

In addition, CheapBFT advances the state of the art in resource-efficient BFT systems by running a composite agreement protocol that requires only $f + 1$ actively participating replicas for agreeing on requests during normal-case operation. The agreement protocol of CheapBFT consists of three subprotocols: the normal-case protocol *CheapTiny*, the transition protocol *CheapSwitch*, and the fall-back protocol *MinBFT* [VCB+11]. During normal-case operation, CheapTiny makes use of passive replication to save resources; it is the first Byzantine fault-tolerant agreement protocol that requires only $f + 1$ *active* replicas. However, CheapTiny is not able to tolerate faults, so that in case of suspected or detected faulty behavior of replicas, CheapBFT runs CheapSwitch to bring all non-faulty replicas into a consistent state. Having completed CheapSwitch, the replicas temporarily execute the MinBFT protocol, which involves $2f + 1$ active replicas (i. e., it can tolerate up to f faults), before eventually switching back to CheapTiny.

The particular contributions of this chapter are:

- To present and evaluate the CASH subsystem (Section 7.2). CASH prevents equivocation

and is used by CheapBFT for message authentication and verification.

- To describe CheapBFT's normal-case agreement protocol CheapTiny, which uses passive replication to save resources (Section 7.4). CheapTiny works together with the novel transition protocol CheapSwitch, which allows to abort CheapTiny in favor of a more resilient protocol when faults have been suspected or detected (Section 7.5).

- To evaluate CheapBFT and related BFT systems with different workloads and a Byzantine fault-tolerant variant of the ZooKeeper [HKJR10] coordination service (Section 7.7).

In addition, Section 7.3 provides an overview of CheapBFT and its system model. Section 7.6 outlines the integration of MinBFT [VCB+11]. Section 7.8 discusses design decisions, Section 7.9 presents related work, and Section 7.10 concludes.

7.2 Preventing Equivocation

Our proposal of a resource-efficient BFT system is based on a trusted subsystem that prevents *equivocation*; that is, the ability of a node to make conflicting statements to different participants in a distributed protocol. In this section, we give background information on why preventing equivocation allows one to reduce the minimum number of replicas in a BFT system from $3f + 1$ to $2f + 1$. Furthermore, we present and evaluate CheapBFT's FPGA-based CASH subsystem used for message authentication and verification.

7.2.1 From $3f + 1$ Replicas to $2f + 1$ Replicas

In traditional BFT protocols like PBFT [CL02], a dedicated replica, the *leader*, proposes the order in which to execute requests. As a malicious leader may send conflicting proposals to different replicas (equivocation), the protocol requires an additional communication round to ensure that all non-faulty replicas act on the same proposal. In this round, each non-faulty replica echoes the proposal it has received from the leader by broadcasting it to all other replicas, enabling all non-faulty replicas to confirm the proposal.

In recent years, alternative solutions have been introduced to prevent equivocation, which eliminate the need for the additional round of communication [VCB+11] and/or reduce the minimum number of replicas in a BFT system from $3f + 1$ to $2f + 1$ [CMSK07, CNV04, VCB+11]. Chun et al. [CMSK07], for example, present an attested append-only memory (A2M) that provides a trusted log for recording the messages transmitted in a protocol. As every replica may access the log independently to validate the messages, non-faulty replicas are able to detect when a leader sends conflicting proposals.

Levin et al. [LDLM09] show that it is sufficient for a trusted subsystem to provide a monotonically increasing counter. In their approach, the subsystem securely assigns a unique counter value to each message and guarantees that it will never bind the same counter value to a different message. Hence, when a replica receives a message, it can be sure that no other replica ever sees a message with the same counter value but different content. As each non-faulty replica validates that the sequence of counter values of messages received from another replica does not contain gaps, malicious replicas cannot equivocate messages. Levin et al. used the trusted counter to build A2M, from which a BFT system with $2f + 1$ replicas has been realized.

We propose CheapBFT, a system with only $f + 1$ active replicas, built directly from the trusted counter. In the following, we present the trusted counter service in CheapBFT.

7.2.2 The CASH Subsystem

The *CASH* (*C*ounter *A*ssignment *S*ervice in *H*ardware) subsystem is used by CheapBFT for message authentication and verification. To prevent equivocation, we require each replica to comprise a trusted CASH subsystem; it is initialized with a secret key and uniquely identified by a subsystem id, which corresponds to the replica that hosts the subsystem. The secret key is shared among the subsystems of all replicas. Apart from the secret key, the internal state of a subsystem as well as the algorithm used to authenticate messages may be known publicly.

For now, we assume that the secret key is manually installed before system startup. In a future version, every CASH subsystem would maintain a private key and expose the corresponding public key. A shared secret key for every protocol instance may be generated during initialization, encrypted under the public key of every subsystem, and transported securely to every replica.

Trusted Counter Service

CASH prevents equivocation by issuing *message certificates* for protocol messages. A message certificate is a cryptographically protected proof that a certain CASH instance has bound a unique counter value to a message. It comprises the id of the subsystem that issued the certificate, the counter value assigned, and a message authentication code (MAC) generated with the secret key. Note that CASH only needs symmetric-key cryptographic operations for message authentication and verification, which are much faster than public-key operations.

The basic version of CASH provides functions for creating ($createMC$) and verifying ($checkMC$) message certificates (see Figure 7.1). When called with a message m, the $createMC$ function increments the local counter and uses the secret key K to generate a MAC a covering the local subsystem id S, the current counter value c, and the message (L. 7-8). The message certificate mc is then created by appending S, c, and a (L. 9). To attest a certificate issued by another subsystem s, the $checkMC$ function verifies the certificate's MAC and uses a function $isNext()$ to validate that the sequence of messages the local subsystem has received from subsystem s contains no gaps (L. 14). Internally, the $isNext()$ function keeps track of the latest counter values of all subsystems and is therefore able to decide whether a counter value c_s assigned to a message is the next in line for subsystem s. If this is the case, the $isNext()$ function increments the counter corresponding to subsystem s and returns success; otherwise, the counter remains unchanged.

To support distinct counter instances in a protocol and several concurrent protocols, the full version of CASH supports multiple counters, each specified by a different *counter name*. All counters to be used have to be provisioned during initialization. In the counter implementation, the name becomes a part of the argument passed to the MAC for the creation and verification of message certificates. In the remainder of this chapter, the counter name is written as a subscript to CASH operations (e. g., $createMC_c$ for counter c).

Furthermore, CASH provides operations for verifying a certificate without checking the correspondence of the counter values and without the side-effect of incrementing the counter in $isNext()$; there are also administrative operations for reading the subsystem id, the configured counter names, and the values of all internal counters. These operations are omitted from Figure 7.1. There are no means for the host system to modify subsystem id, counter names, or counter values after the initialization stage.

Implementation

We developed CASH to meet the following design goals:

```
1  upon initialization do
2      K := secret key;
3      S := local subsystem id;
4      c := 0;

6  upon call createMC(m) do
7      c := c + 1;
8      a := MAC(K, S||c||m);
9      mc := (S, c, a);
10     return mc;

12 upon call checkMC(mc, m) do
13     (s, c_s, a) := mc;
14     if MAC(K, s||c_s||m) = a and isNext(s, c_s) do
15         return TRUE;
16     else
17         return FALSE;
```

Figure 7.1: Implementation of CASH's trusted counter.

- **Minimal trusted computing base**: The code size of CASH must be small to reduce the probability of program errors that could be exploited by attackers. Given its limited functionality, there is no need to trust an entire (hardened) Linux kernel [CNV04] or hypervisor [RK07].

- **High performance**: As every interaction between replicas involves authenticated messages, we require CASH to handle thousands of messages per second. Therefore, the use of trusted platform modules or smart cards is not an option, as on such systems a single authentication operation takes more than 100 milliseconds [LDLM09, VCB+11].

Our implementation of CASH is based on a commodity Xilinx Spartan-3 XC3S1500 FPGA mounted on a dedicated PCI card. Both the program code and the secret key are stored on the FPGA and cannot be accessed or modified by the operating system of the host machine. The only way to reprogram the subsystem is by attaching an FPGA programmer, which requires physical access to the machine.

As depicted in Figure 7.2, applications communicate with the FPGA via a character device (i. e., `/dev/cash`). To authenticate a message, for example, the application first writes both a CREATEMC op code and the message to the device, and then retrieves the message certificate as soon it becomes available. Our current prototype uses an HMAC-SHA-256 for the authentication of messages.

Integration with CheapBFT

In CheapBFT, replicas use the CASH subsystem to authenticate all messages intended for other replicas. However, this does not apply to messages sent to clients, as those messages are not subject to equivocation. To authenticate a message, a replica first calculates a hash of the message and then passes the hash to CASH's $createMC$ function. Creating a message certificate for the message hash instead of the full message increases the throughput of the subsystem, especially for large messages, as less data has to be transferred to the FPGA. To verify a message received from another replica, a replica calls the $checkMC$ function of its local CASH instance, passing

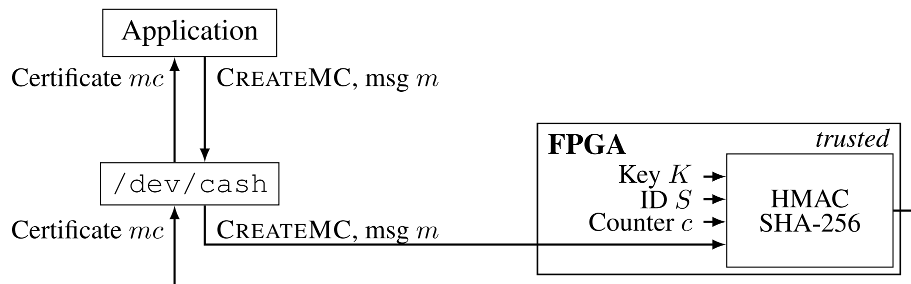


Figure 7.2: Creation of a message certificate mc for a message m using the FPGA-based trusted CASH subsystem.

the message certificate received as well as a hash of the message. Note that, for simplicity, we omit the use of this hash in the description of CheapBFT.

Performance Evaluation

We evaluate the performance of the CASH subsystem integrated with an 8-core machine (2.3 GHz, 8 GB RAM) and compare CASH with three other subsystems that provide the same service of assigning counter values to messages:

- **SoftLib** is a library that performs message authentication and verification completely in software. As it runs in the same process as the replica and therefore does not require any additional communication, we consider its overhead to be minimal. Note, however, that it is not feasible to use SoftLib in a BFT setting with $2f + 1$ replicas because trusting SoftLib would imply trusting the whole replica.

- **SSL** is a local OpenSSL server running in a separate process on the replica host. Like SoftLib, we evaluate SSL only for comparison, as it would also not be safe to use this subsystem in a BFT system with $2f + 1$ replicas.

- **VM-SSL** is a variant of SSL, in which the OpenSSL server runs in a Xen domain on the same host, similar to the approach used in [VCBL10]. Relying on VM-SSL requires one to trust that the hypervisor enforces isolation.

In this experiment, we measure the time it takes each subsystem variant to create certificates for messages of different sizes, which includes computing a SHA-256 hash (32 bytes) over a message and then authenticating only the hash, not the full message (see Section 7.2.2). In addition, we evaluate the verification of message certificates. Table 7.1 presents the results for message authentication and verification for the four subsystems evaluated. The first set of values excludes the computation of the message hash and only reports the times it takes the subsystems to authenticate/verify a hash. With all four trusted counter service implementations only relying on symmetric-key cryptographic operations, the results in Tables 7.0(a) and 7.0(b) show a similar picture.

In the VM-SSL subsystem, the overhead for communication with the virtual machine dominates the authentication process and leads to results of more than a millisecond, independent of message size. Executing the same binary as VM-SSL but requiring only local socket communication, SSL achieves a performance in the microseconds range. In SoftLib, which does not involve any inter-process communication, the processing time significantly increases with message size. In our CASH subsystem, creating a certificate for a message hash takes 57 microseconds, which

(a) Creation overhead for a certificate depending on message size.

Subsystem	Message size			
	32 B (no hashing)	32 B	1 KB	4 KB
VM-SSL	1013	1014	1015	1014
SSL	67	69	86	139
SoftLib	4	4	17	55
CASH	57	58	77	131

(b) Verification overhead for a certificate depending on message size.

Subsystem	Message size			
	32 B (no hashing)	32 B	1 KB	4 KB
VM-SSL	1013	1013	1013	1012
SSL	67	69	87	140
SoftLib	4	4	17	55
CASH	60	62	80	134

Table 7.1: Overhead (in microseconds) for creating and verifying a message certificate in different subsystems.

is mainly due to the costs for communication with the FPGA. As a result, CASH is able to authenticate more than 17,500 messages per second. Depending on the message size, computing the hash adds up 1 to 74 microseconds per operation; however, as hash creation is done in software, this can be done in parallel with the FPGA authenticating another message hash. The results in Table 7.0(b) show that in CASH the verification of a certificate for a message hash takes about 5% longer than its creation. This is due to the fact that in order to check a certificate, the FPGA not only has to recompute the certificate but also needs to perform a comparison.

Note that we did not evaluate a subsystem based on a trusted platform module (TPM), as the TPMs currently available only allow a single increment operation every 3.5 seconds to protect their internal counter from burning out too soon [VCB+11]. A TPM implementation based on reconfigurable hardware that could be adapted to overcome this issue did not reach the prototype status due to hardware limitations [EGP+07]. Alternative implementations either perform substantial parts in software, which makes them comparable to the software-based systems we presented, or suffer from the same problems as commodity solutions [BCG+06, EL08].

Furthermore, we did not measure the performance of a smart-card-based subsystem: in [LDLM09], Levin et al. report a single authentication operation with 3-DES to take 129 milliseconds, and the verification operation to take 86 milliseconds using a smart card. This is orders of magnitude slower than the performance of CASH.

Trusted Computing Base

Besides performance, the complexity of a trusted subsystem is crucial: the more complex a subsystem, the more likely it is to fail in an arbitrary way, for example, due to an attacker exploiting a vulnerability. In consequence, to justify the assumption of the subsystem being trusted, it is essential to minimize its trusted computing base.

Table 7.2 outlines that the basic counter logic and the routines necessary to create and check message certificates are similar in complexity for both SSL variants and CASH. However, the software-based isolation and execution substrate for SSL and VM-SSL are clearly larger albeit we use the conservative values presented by Steinberg and Kauer [SK10]. In contrast, the trusted computing base of a TPM is rather small: based on the TPM emulator implementation of Strasser and Stamer [SS08], we estimate its size to be about 20 KLOC, which is only slightly smaller than the trusted computing base of CASH. For a smartcard-based solution, we assume similar values for the counter logic and certificate handling as for CASH. In addition some runtime support has to be accounted.

Going one step beyond approximating code complexity, it has to be noted that FPGAs, as used by CASH, per se are less resilient to single event upsets (e. g., bit flips caused by radiation) compared to dedicated hardware. However, fault-tolerance schemes can be applied that enable the use of FPGAs even in the space and nuclear sector [SSC10]. Regarding code generation

Subsystem	Components	KLOC	Total
SSL	Linux	200.0	
	Counter logic	0.3	
	Cryptographic functions	0.4	**200.7**
VM-SSL	Virtualization	100.0	**300.7**
CASH	PCI core	18.5	
	Counter logic	2.2	
	Cryptographic functions	0.8	**21.5**

Table 7.2: Size comparison of the trusted computing bases of different subsystems in thousands of lines of code.

and the verifiability of code, similar tool chains can be used for CASH and building TPMs. Accordingly, their trustworthiness should be comparable.

In summary, our CASH subsystem comprises a small trusted computing base, which is comparable in size to the trusted computing base of a TPM, and similarly resilient to faults, while providing a much higher performance than readily available TPM implementations (see Section 7.2.2).

7.3 CheapBFT

This section presents our system model and gives an overview of the composite agreement protocol used in CheapBFT to save resources during normal-case operation; the subprotocols are detailed in Sections 7.4 to 7.6.

7.3.1 System Model

We assume the system model used for most BFT systems based on state-machine replication [CL02, YMV+03, KD04, KAD+09, VCBL09, VCB+11, VCBL10] according to which up to f replicas and an unlimited number of clients may fail arbitrarily (i. e., exhibit Byzantine faults). Every replica hosts a trusted CASH subsystem with its subsystem id set to the replica's identity. The trusted CASH subsystem may fail only by crashing and its key remains secret even at Byzantine replicas. As discussed in Section 7.2.2, this implies that an attacker cannot gain physical access to a replica. In accordance with other BFT systems, we assume that replicas only process requests of authenticated clients and ignore any messages sent by other clients.

The network used for communication between clients and replicas may drop messages, delay them, or deliver them out of order. However, for simplicity, we use the abstraction of FIFO channels, assumed to be provided by a lower layer, in the description of the CheapBFT protocols. For authenticating point-to-point messages where needed, the operations of CASH are invoked. Our system is safe in an asynchronous environment; to guarantee liveness, we require the network and processes to be partially synchronous.

7.3.2 Resource-efficient Replication

CheapBFT has been designed with a focus on saving resources. Compared with BFT systems like PBFT [CL02, YMV+03, KD04, KAD+09, VCBL09], it achieves better resource efficiency

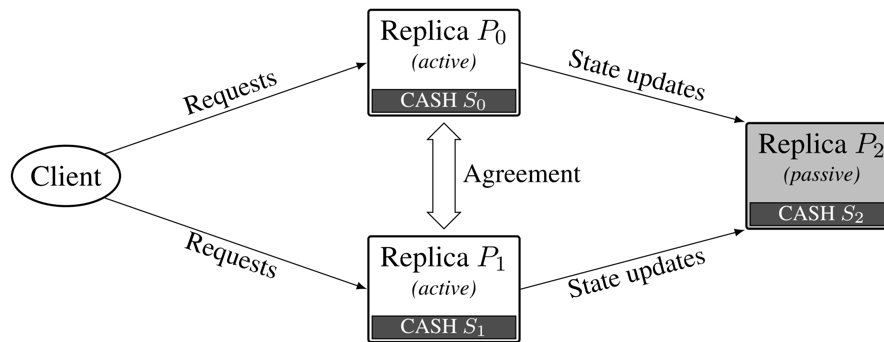


Figure 7.3: CheapBFT architecture with two active replicas and a passive replica ($f = 1$) for normal-case operation.

thanks to two major design changes: First, each CheapBFT replica has a small trusted CASH subsystem that prevents equivocation (see Section 7.2); this not only allows us to reduce the minimum number of replicas from $3f + 1$ to $2f + 1$ but also minimizes the number of protocol messages [YMV+03, CNV04, CMSK07, LDLM09, VCB+11, VCBL10]. Second, CheapBFT uses a composite agreement protocol that saves resources during normal-case operation by supporting *passive replication*.

In traditional BFT systems [CL02, KD04, KAD+09, VCBL09], all (non-faulty) replicas participate in both the agreement and the execution of requests. As recent work has shown [DK11, WSV+11], in the absence of faults, it is sufficient to actually process a request on only $f + 1$ replicas as long as it is guaranteed that all other replicas are able to safely obtain changes to the application state. In CheapBFT, we take this idea even further and propose our CheapTiny protocol, in which only $f + 1$ *active* replicas take part in the agreement stage during normal-case operation (see Figure 7.3). The other f replicas remain *passive*, that is, they neither agree on requests nor execute requests. Instead, passive replicas modify their states by processing validated *state updates* provided by the active replicas. This approach minimizes not only the number of executions but also the number of protocol messages.

7.3.3 Fault Handling

With only $f + 1$ replicas actively participating in the protocol, CheapTiny is not able to tolerate faults. Therefore, in case of suspected or detected faulty behavior of one or more active replicas, CheapBFT abandons CheapTiny in favor of a more resilient protocol. The current CheapBFT prototype relies on MinBFT [VCB+11] for this purpose, but we could have selected other BFT protocols (e. g., A2M-PBFT-EA [CMSK07]) that make use of $2f + 1$ replicas to tolerate f faults.

During the protocol switch to MinBFT, CheapBFT runs the CheapSwitch transition protocol to ensure that replicas start the new MinBFT protocol instance in a consistent state. The main task of non-faulty replicas in CheapSwitch is to agree on a CheapTiny *abort history*. An abort history is a list of protocol messages that indicates the status of pending requests and therefore allows the remaining non-faulty replicas to safely continue agreement. In contrast to Abstract [GKQV10], which relies on a similar technique to change protocols, an abort history in CheapBFT can be verified to be correct even if it has only been provided by a single replica.

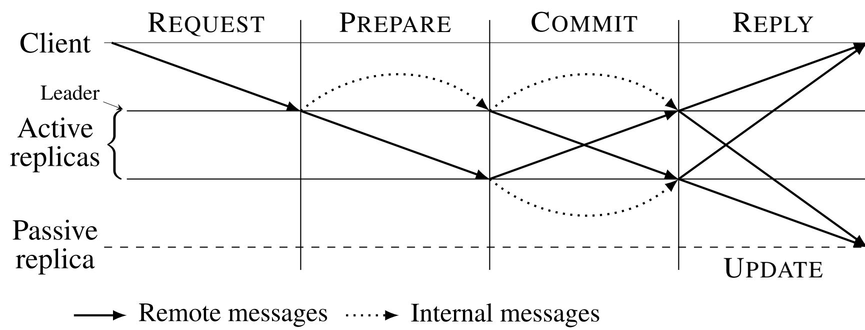


Figure 7.4: CheapTiny protocol messages exchanged between a client, two active replicas, and a passive replica ($f = 1$).

7.4 Normal-case Protocol: CheapTiny

CheapTiny is the default protocol of CheapBFT and designed to save resources in the absence of faults by making use of passive replication. It comprises a total of four phases of communication (see Figure 7.4), which resemble the phases in PBFT [CL02]. However, as CheapBFT replicas rely on a trusted subsystem to prevent equivocation, the CheapTiny protocol does not require a pre-prepare phase.

7.4.1 Client

During normal-case operation, clients in CheapBFT behave similar to clients in other BFT state-machine-replication protocols: Upon each new request, a client sends a $\langle \text{REQUEST}, m \rangle$ message authenticated by the client's key to the leader; m is a request object containing the id of the client, the command to be executed, as well as a client-specific sequence number that is used by the replicas to ensure exactly-once semantics. After sending the request, the client waits until it has received $f + 1$ matching replies from different replicas, which form a proof for the correctness of the reply in the presence of at most f faults.

7.4.2 Replica

Taking up the separation introduced by Yin et al. [YMV+03], the internal architecture of an active CheapBFT replica can be logically divided into two stages: the *agreement stage* establishes a stable total order on client requests, whereas the *execution stage* is responsible for processing requests and for providing state updates to passive replicas. Note that as passive replicas do not take part in the agreement of requests, they also do not execute the CheapTiny agreement stage.

Both stages draw on the CASH subsystem to authenticate messages intended for other replicas. To decouple agreement messages from state updates, a replica uses two trusted counters, called ag and up.

Agreement Stage

During protocol initialization, each replica is assigned a unique id (see Figure 7.5, L. 2). Furthermore, a set of $f + 1$ active replicas is selected in a deterministic way. The active replica with the lowest id becomes the leader (L. 3-5). Similarly to other PBFT-inspired agreement protocols, the

```
1  upon initialization do
2      P := local replica id;
3      active := {p_0, p_1, ... p_f};
4      passive := {p_{f+1}, p_{f+2}, ..., p_{2f}};
5      leader := select_leader(active);

7  upon receiving ⟨REQUEST, m⟩ such that P = leader do
8      mc_L := createMC_ag(m);
9      send ⟨PREPARE, m, mc_L⟩ to all in active;

11 upon receiving ⟨PREPARE, m, mc_L⟩ such that (mc_L = (leader, ·, ·)) and checkMC_ag(mc_L, m) do
12     mc_P := createMC_ag(m||mc_L);
13     send ⟨COMMIT, m, mc_L, mc_P⟩ to all in active;

15 upon receiving C := { ⟨COMMIT, m, mc_L, mc_p⟩ with mc_p = (p, ·, ·) from every p in active such that
         checkMC_ag(mc_p, m||mc_L) and all m are equal } do
16     execute(m, C);
```

Figure 7.5: CheapTiny agreement protocol for active replicas.

leader in CheapTiny is responsible for proposing the order in which requests from clients are to be executed. When all $f + 1$ active replicas have accepted a proposed request, the request becomes *committed* and can be processed safely.

When the leader receives a client request, it first verifies the authenticity of the request (omitted in Figure 7.5). If the request is valid and originates from an authenticated client, the leader then broadcasts a $\langle \text{PREPARE}, m, mc_L \rangle$ message to all active replicas (L. 7-9). The PREPARE contains the client request m and a message certificate mc_L issued by the local trusted CASH subsystem. The certificate uses the agreement-stage-specific counter ag and contains the leader's identity in the form of the subsystem id.

Upon receiving a PREPARE (L. 11), an active replica asks CASH to verify that it originates from the leader, that the message certificate is valid, and that the PREPARE is the next message sent by the leader, as indicated by the assigned counter value. This procedure guarantees that the replica only accepts the PREPARE if the sequence of messages received from the leader contains no gaps. If the message certificate has been successfully verified, the replica sends a $\langle \text{COMMIT}, m, mc_L, mc_P \rangle$ message to all active replicas (L. 13). As part of the COMMIT, the replica propagates its own message certificate mc_P for the request m, which is created by authenticating the concatenation of m and the leader's certificate mc_L (L. 12). Note that issuing a combined certificate for m and mc_L helps replicas determine the status of pending requests in case of a protocol abort, as the certificate is a proof that the replica has received and accepted both m and mc_L (see Section 7.5.3).

When an active replica receives a COMMIT message, it extracts the sender p from mc_p and verifies that the message certificate mc_p is valid (L. 15). As soon as the replica has obtained a set $\mathcal{C}$ of $f + 1$ valid COMMITs for the same request m (one from each active replica, as determined by the subsystem id found in the message certificates), the request is committed and the replica forwards m to the execution stage (L. 15-16). Because of our assumption of FIFO channels and because of the fact that COMMITs from all $f + 1$ active replicas have to be available, CheapTiny guarantees that requests are committed in the order proposed by the leader without explicit use of a sequence number.

```
1 upon call execute(m, C) do
2     (r, u) := process(m);
3     uc_P := createMC_up(r||u||C);
4     send ⟨UPDATE, r, u, C, uc_P⟩ to all in passive;
5     send ⟨REPLY, P, r⟩ to client;
```

Figure 7.6: CheapTiny execution-stage protocol run by active replicas to execute requests and distribute state updates.

```
1 upon receiving {
        ⟨UPDATE, r, u, C, uc_p⟩ with uc_p = (p, ·, ·)
        from every p in active
        such that checkMC_up(uc_p, r||u||C)
        and all r are equal and all u are equal
    } do
2     process(u);
```

Figure 7.7: CheapTiny execution-stage protocol run by passive replicas to process updates provided by active replicas.

Execution Stage

Processing a request m in CheapBFT requires the application to provide two objects (see Figure 7.6, L. 2): a reply r intended for the client and a state update u that reflects the changes to the application state caused by the execution of m. Having processed a request, an active replica asks the CASH subsystem to create an update certificate uc_P for the concatenation of r, u, and the set of COMMITs $\mathcal{C}$ confirming that m has been committed (L. 3). The update certificate is generated using the counter up, which is dedicated to the execution stage. Next, the active replica sends an $\langle \text{UPDATE}, r, u, \mathcal{C}, uc_P \rangle$ message to all passive replicas (L. 4), and finally forwards the reply to the client (L. 5).

Upon receiving an UPDATE, a passive replica confirms that the update certificate is correct and that its assigned counter value indicates no gaps (see Figure 7.7, L. 1). When the replica has received $f + 1$ matching UPDATEs from all active replicas for the same reply and state update, the replica adjusts its application state by processing the state update (L. 1-2).

Checkpoints and Garbage Collection

In case of a protocol switch, active replicas must be able to provide an abort history indicating the agreement status of pending requests (see Section 7.5). Therefore, an active replica logs all protocol messages sent to other replicas (omitted in Figures 7.5 and 7.6). To prevent a replica from running out of memory, CheapTiny makes use of periodic protocol checkpoints that allow a replica to truncate its message log.

A non-faulty active replica creates a new checkpoint after the execution of every kth request; k is a system-wide constant (e. g., 200). Having distributed the UPDATE for a request q that triggered a checkpoint, the replica first creates an application snapshot. Next, the replica sends a $\langle \text{CHECKPOINT}, ash_q, cc_{ag}, cc_{up} \rangle$ message to all (active and passive) replicas, which includes a digest of the application snapshot ash_q and two checkpoint certificates, cc_{ag} and cc_{up}, issued under the two CASH counters ag and up.

Upon receiving a CHECKPOINT, a replica verifies that its certificates are correct and that the counter values assigned are both in line with expectations. A checkpoint becomes stable

as soon as a replica has obtained matching checkpoints from all $f + 1$ active replicas. In this case, an active replica discards all requests up to request q as well as all corresponding PREPARE, COMMIT, and UPDATE messages.

Optimizations

CheapTiny allows to apply most of the standard optimizations used in Byzantine fault-tolerant protocols related to PBFT [CL02]. In particular, this includes batching, which makes it possible to agree on multiple requests (combined in a batch) within a single round of agreement. In the following, we want to emphasize two additional optimizations to reduce communication costs.

Implicit Leader COMMIT In the protocol description in Figure 7.4, the leader sends a COMMIT to all active replicas after having received its own (internal) PREPARE. As this COMMIT carries no additional information, the leader's PREPARE and COMMIT can be merged into a single message that is distributed upon receiving a request; that is, all replicas treat a PREPARE from the leader as an implicit COMMIT.

Use of Hashes PBFT reduces communication costs by selecting one replica for each request to send a full reply. All other replicas only provide a hash of the reply that allows the client to prove the result correct. The same approach can be implemented in CheapTiny. Furthermore, only a single active replica in CheapTiny needs to include a full state update in its UPDATE for the passive replicas.

7.5 Transition Protocol: CheapSwitch

CheapTiny is optimized to save resources during normal-case operation. However, the subprotocol is not able to make progress in the presence of suspected or detected faulty behavior of replicas. In such cases, CheapBFT falls back to the MinBFT protocol, which relies on $2f + 1$ active replicas and can therefore tolerate up to f faults. In this section, we present the CheapSwitch transition protocol responsible for the safe protocol switch.

7.5.1 Initiating a Protocol Switch

In CheapBFT, all nodes are eligible to request the abortion of the CheapTiny protocol. There are two scenarios that trigger a protocol switch:

- A client asks for a protocol switch in case the active replicas fail to provide $f + 1$ matching replies to a request within a certain period of time.
- A replica demands to abort CheapTiny if it suspects or detects that another replica does not behave according to the protocol specification, for example, by sending a false message certificate, or by not providing a valid checkpoint or state update in a timely manner.

In these cases, the node requesting the protocol switch sends a $\langle$PANIC$\rangle$ message to all (active and passive) replicas (see Figure 7.8). The replicas react by rebroadcasting the message to ensure that all replicas are notified (omitted in Figure 7.8). Furthermore, upon receiving a PANIC, a non-faulty active replica stops to send CheapTiny protocol messages and waits for the leader of the new CheapSwitch protocol instance to distribute an abort history. The CheapSwitch leader

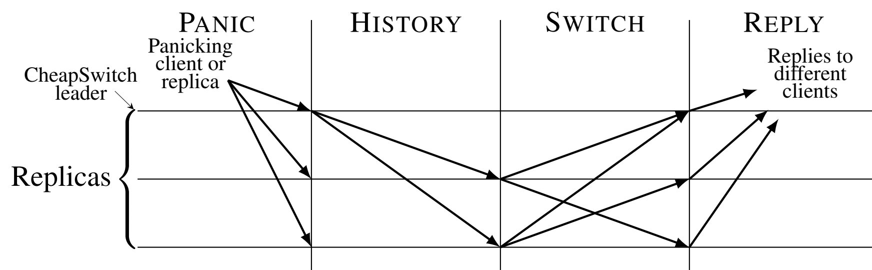


Figure 7.8: CheapSwitch protocol messages exchanged between clients and replicas during protocol switch ($f = 1$).

is chosen deterministically as the active replica with the lowest id apart from the leader of the previous CheapTiny protocol.

7.5.2 Creating an Abort History

An abort history is used by non-faulty replicas to safely end the active CheapTiny instance during a protocol switch. It comprises the CHECKPOINTs of all active replicas proving that the latest checkpoint has become stable, as well as a set of CheapTiny protocol messages that provide replicas with information about the status of pending requests. We distinguish three status categories:

- **Decided**: The request has been committed prior to the protocol abort. The leader proves this by including the corresponding UPDATE (which comprises the set of $f + 1$ COMMITs from all active replicas) in the history.
- **Potentially decided**: The request has not been committed, but prior to the protocol abort, the leader has received a valid PREPARE for the request and has therefore sent out a corresponding COMMIT. Accordingly, the request may have been committed on some active replicas. In this case, the leader includes its own COMMIT in the history.
- **Undecided**: The leader has received a request and/or a PREPARE for a request, but has not yet sent a COMMIT. As a result, the request cannot have been committed on any non-faulty replica. In this case, the leader includes the request in the abort history.

When creating the abort history, the leader of the CheapSwitch protocol instance has to consider the status of all requests that are not covered by the latest stable checkpoint. When a history h is complete, the leader asks the CASH subsystem for two history certificates $hc_{L,ag}$ and $hc_{L,up}$, authenticated by *both* counters. Then it sends a $\langle \text{HISTORY}, h, hc_{L,ag}, hc_{L,up} \rangle$ message to all replicas.

7.5.3 Validating an Abort History

When a replica receives an abort history from the leader of the CheapSwitch instance, it verifies that the history is correct. An abort history is deemed to be correct by a correct replica when all of the following four criteria hold:

- Both history certificates verify correctly.

- The CHECKPOINTs contained in the abort history prove that the latest checkpoint has become stable.
- Using only information contained in the abort history, the replica can reconstruct *the complete sequence* of authenticated protocol messages that the CheapSwitch leader has sent in CheapTiny since the latest checkpoint.
- The reconstructed sequence of messages does not violate the CheapTiny protocol specification.

Note that although an abort history is issued by only a single replica (i. e., the new leader), all other replicas are able to verify its correctness independently: each UPDATE contains the $f + 1$ COMMIT certificates that prove a request to be decided; each COMMIT in turn comprises a certificate that proves that the old leader has sent a PREPARE for the request (see Section 7.4.2). As replicas verify that all these certificates are valid and that the sequence of messages sent by the leader has no gaps, a malicious leader cannot modify or invent authenticated protocol messages and include them in the history without being detected. As a result, it is safe to use a correct abort history to get replicas into a consistent state (see Section 7.5.4).

Figure 7.9 shows an example of an abort history deemed to be correct, containing the proof CHK that the latest checkpoint has become stable, UPDATEs for three decided requests a, b, and c, and a COMMIT for a potentially decided request d. After verifying that all certificates are correct, a replica ensures that the messages in the history do not violate the protocol specification (e. g., the UPDATE for request a must comprise $f + 1$ matching COMMITs for a). Finally, a replica checks that the abort history proves the complete sequence of messages sent by the leader since the latest checkpoint; that is, the history must contain an authenticated message for every counter value of both the agreement-stage counter ag as well as the execution-stage counter up, starting from the counter values assigned to the last checkpoint and ending with the counter values assigned to the abort history.

The requirement to report a complete sequence of messages prevents equivocation by a malicious leader. In particular, a malicious leader cannot send inconsistent authenticated abort histories to different replicas without being detected: in order to create diverging histories that are both deemed to be correct, the leader would be forced to include the first authenticated history into all other histories. Furthermore, the complete message sequence ensures that all decided or potentially decided requests are included in the history: if a malicious leader, for example, sends a COMMIT for a request e after having created the history, all non-faulty replicas will detect the gap in the sequence of agreement counter values (caused by the history) and ignore the COMMIT. As a result, it is impossible for e to have been decided in the old CheapTiny instance. This property depends critically on the trusted counter.

7.5.4 Processing an Abort History

Having concluded that an abort history is correct, a replica sends a $\langle \text{SWITCH}, hh, hc_{L,ag}, hc_{L,up}, hc_{P,ag}, hc_{P,up} \rangle$ message to all other replicas (see Figure 7.8); hh is a hash of the abort history, $hc_{L,ag}$ and $hc_{L,up}$ are the leader's history certificates, and $hc_{P,ag}$ and $hc_{P,up}$ are history certificates issued by the replica and generated with the agreement-stage counter and the update-stage counter, respectively. Note that a SWITCH is to a HISTORY what a COMMIT is to a PREPARE. When a replica has obtained a correct history and f matching SWITCH messages from different replicas, the history becomes stable. In this case, a replica processes the abort history, taking into account its local state.

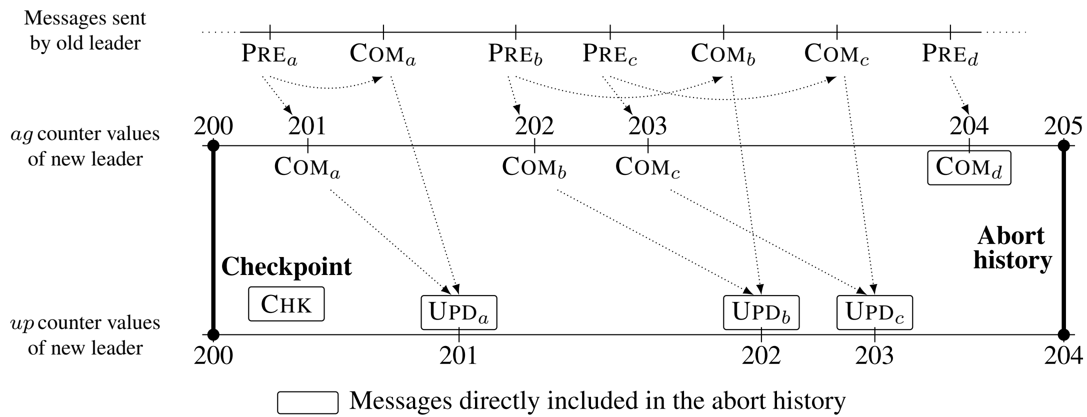


Figure 7.9: Dependencies of UPDATE (UPD$_*$) and COMMIT (COM$_*$) messages contained in a correct CheapTiny abort history for four requests a, b, c, and d ($f = 1$).

First, a replica executes all decided requests that have not yet been processed locally, retaining the order determined by the history, and sends the replies back to the respective clients. Former passive replicas only execute a decided request if they have not yet processed the corresponding state update. Next, a replica executes all unprocessed potentially decided requests as well as all undecided requests from the history. This is safe, as both categories of requests have been implicitly decided by $f + 1$ replicas accepting the abort history. Having processed the history, all non-faulty replicas are in a consistent state and therefore able to safely switch to the new MinBFT protocol instance.

7.5.5 Handling Faults

If an abort history does not become stable within a certain period of time after having received a PANIC, a replica suspects the leader of the CheapSwitch protocol to be faulty. As a consequence, a new instance of the CheapSwitch protocol is started, whose leader is chosen deterministically as the active replica with the smallest id that has not already been leader in an immediately preceding CheapSwitch instance. If these options have all been exploited the leader of the last CheapTiny protocol instance is chosen. To this end, the suspecting replica sends a $\langle \text{SKIP}, p_{NL}, sc_{P,ag}, sc_{P,up} \rangle$ message to all replicas, where p_{NL} denotes the replica that will now become the leader; $sc_{P,ag}$ and $sc_{P,up}$ are two skip certificates authenticated by both trusted counters ag and up, respectively. Upon obtaining $f + 1$ matching SKIPs with correct certificates, p_{NL} becomes the new leader and reacts by creating and distributing its own abort history.

The abort history provided by the new leader may differ from the old leader's abort history. However, as non-faulty replicas only accept an abort history from a new leader after having received at least $f + 1$ SKIPs proving a leader change, it is impossible that a non-faulty replica has already processed the abort history of the old leader.

Consider two abort histories h_0 and h_1 that are both deemed to be correct, but are provided by different replicas P_0 and P_1. Note that the extent to which they can differ is limited. Making use of the trusted CASH subsystem guarantees that the order (as indicated by the counter values assigned) of authenticated messages that are included in both h_0 and h_1 is identical across both histories. However, h_0 may contain messages that are not in h_1, and vice versa, for example, because one of the replicas has already received $f + 1$ COMMITs for a request, but the other replica has not yet done so. As a result, both histories may report a slightly different status for each pending request: In h_0, for example, a request may have already been decided, whereas in

h_1 its is reported to be potentially decided. Also, a request may be potentially decided in one history and undecided in the other.

However, if both histories are deemed to be correct, h_0 will never report a request to be decided that is undecided in h_1. This is based on the fact that for the request to become decided on P_0, P_1 must have provided an authenticated COMMIT for the request. Therefore, P_1 is forced to include this COMMIT in h_1 to create a correct history, which upgrades the status of the request to potentially decided (see Section 7.5.2). In consequence, it is safe to complete the CheapSwitch protocol by processing any correct abort history available, as long as all replicas process the same history, because all correct histories contain all requests that have become decided on at least one non-faulty replica.

It is possible that the abort history eventually processed does not contain all undecided requests, for example, because the CheapSwitch leader may not have seen all PREPAREs distributed by the CheapTiny leader. Therefore, a client retransmits its request if it is not able to obtain a stable result after having demanded a protocol switch. All requests that are not executed prior to or during the CheapSwitch run are handled by the following MinBFT instance.

7.6 Fall-back Protocol: MinBFT

After completing CheapSwitch, a replica is properly initialized to run the MinBFT protocol [VCB+11]. In contrast to CheapTiny, all $2f + 1$ replicas in MinBFT are active, which allows the protocol to tolerate up to f faults. However, as we expect permanent replica faults to be rare [GKQV10, DK11, WSV+11], the protocol switch to MinBFT will in most cases be performed to make progress in the presence of temporary faults or periods of asynchrony. To address this issue, CheapBFT executes MinBFT for only a limited period of time and then switches back to CheapTiny, similarly to the approach proposed by Guerraoui et al. in [GKQV10].

7.6.1 Protocol

In MinBFT, all replicas actively participate in the agreement of requests. Apart from that, the protocol steps are similar to CheapTiny: when the leader receives a client request, it sends a PREPARE to all other replicas, which in turn respond by multicasting COMMITs, including the PREPARE certificate. Upon receiving $f + 1$ matching COMMITs, a replica processes the request and sends a reply back to the client. Similar to CheapTiny, replicas in MinBFT authenticate all agreement-stage messages using the CASH subsystem and only accept message sequences that contain no gaps and are verified to be correct. Furthermore, MinBFT also relies on stable checkpoints to garbage collect message logs.

7.6.2 Protocol Switch

In CheapBFT, an instance of the MinBFT protocol runs only a predefined number of agreement rounds x. When the xth request becomes committed, a non-faulty replica switches back to the CheapTiny protocol and handles all subsequent requests. Note that if the problem that led to the start of MinBFT has not yet been removed, the CheapTiny fault-handling mechanism ensures that the CheapSwitch transition protocol will be triggered once again, eventually initializing a new instance of MinBFT. This new instance uses a higher value for x to account for the prolonged period of asynchrony or faults.

7.7 Evaluation

In this section, we evaluate the performance and resource consumption of CheapBFT. Our test setting comprises a replica cluster of 8-core machines (2.3 GHz, 8 GB RAM) and a client cluster of 12-core machines (2.4 GHz, 24 GB RAM) that are all connected with switched Gigabit Ethernet.

We have implemented CheapBFT by adapting the BFT-SMaRt library [BS]. Our CheapBFT implementation reuses BFT-SMaRt's communication layer but provides its own composite agreement protocol. Furthermore, CheapBFT relies on the CASH subsystem to authenticate and verify messages. In addition to CheapBFT and BFT-SMaRt, we evaluate an implementation of plain MinBFT [VCB+11]; note that to enable a protocol comparison the MinBFT implementation also uses our CASH subsystem. All of the following experiments are conducted with system configurations that are able to tolerate a single Byzantine fault (i. e., BFT-SMaRt: four replicas, MinBFT and CheapBFT: three replicas). In all cases, the maximum request-batch size is set to 20.

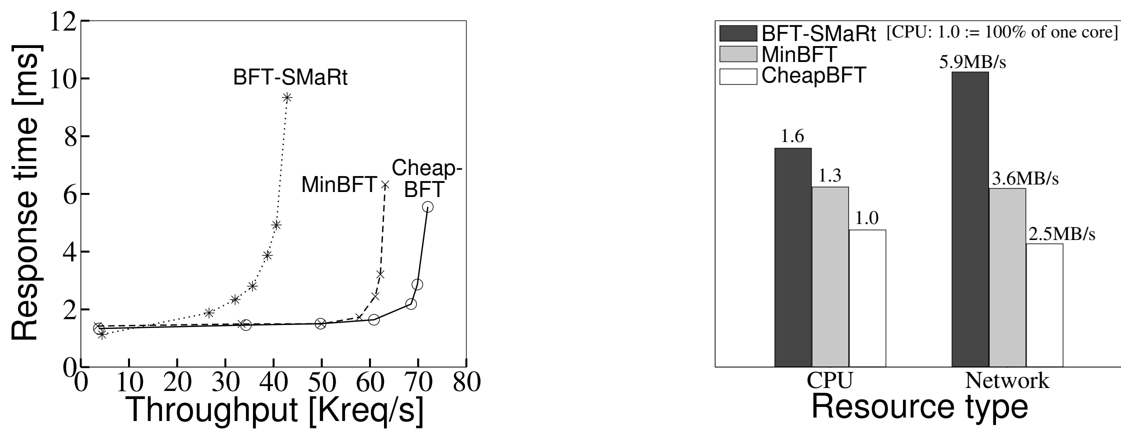
7.7.1 Normal-case Operation

We evaluate BFT-SMaRt, MinBFT, and CheapBFT during normal-case operation using a micro benchmark in which clients continuously send empty requests to replicas; each client waits to receive an empty reply before sending a subsequent request. In the CheapBFT configuration, each client request triggers an empty update. Between test runs, we vary the number of clients from 5 to 400 to increase load and measure the average response time of an operation. With no execution overhead and only small messages to be sent, the focus of the benchmark lies on the throughput of the agreement protocols inside BFT-SMaRt, MinBFT, and CheapBFT.

The performance results in Figure 7.10(a) show that requiring only four instead of five communication steps and only $2f + 1$ instead of $3f + 1$ agreement replicas, MinBFT achieves a significantly higher throughput than BFT-SMaRt. With only the $f + 1$ active replicas taking part in the agreement of requests, a CheapBFT replica needs to handle fewer protocol messages than a MinBFT replica. As a result, CheapBFT is able to process more than 72,000 requests per second, an increase of 14% over MinBFT.

Besides performance, we evaluate the CPU and network usage of BFT-SMaRt, MinBFT, and CheapBFT. In order to be able to directly compare the three systems, we aggregate the resource consumption of all replicas in a system and normalize the respective value at maximum throughput to a throughput of 10,000 requests per second (see Figure 7.10(b)). Compared to MinBFT, CheapBFT requires 24% less CPU, which is mainly due to the fact that a passive replica does not participate in the agreement protocol and neither processes client requests nor sends replies. CheapBFT replicas also send 31% less data than MinBFT replicas over the network, as the simplified agreement protocol of CheapBFT results in a reduced number of messages. Compared to BFT-SMaRt, the resource savings of CheapBFT add up to 37% (CPU) and 58% (network).

We also evaluate the three BFT systems in an experiment in which clients send empty requests and receive replies of 4 kilobyte size. Note that in this scenario, as discussed in Section 7.4.2, only a single replica responds with the actual full reply while all other replicas only provide a reply hash to the client. Figure 7.11 shows the results for performance and resource usage for this experiment. In contrast to the previous benchmark, this benchmark is dominated by the overhead for reply transmission: as full replies constitute the majority of network traffic, CheapBFT replicas only send 2% less data than MinBFT replicas and 8% less data than BFT-SMaRt replicas



(a) Throughput vs. response time for an increasing number of clients.

(b) Average resource usage per 10 Kreq/s normalized by throughput.

Figure 7.10: Performance and resource-usage results for a micro benchmark with empty requests and empty replies.

over the network. Furthermore, the need to provide a passive replica with reply hashes reduces the CPU savings of CheapBFT to 7% compared to MinBFT and 20% compared to BFT-SMaRt.

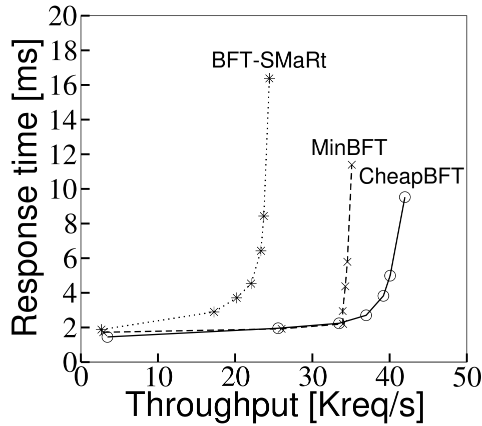
In our third micro-benchmark experiment, clients send requests of 4 kilobyte size and receive empty replies; Figure 7.12 reports the corresponding performance and resource-usage results for this experiment. For such a workload, transmitting requests to active replicas is the decisive factor influencing both performance and resource consumption. With the size of requests being much larger than the size of other protocol messages exchanged between replicas, compared to BFT-SMaRt, CheapBFT replicas need to send 67% less data over the network (50% less data compared to MinBFT). In addition, CheapBFT consumes 54% less CPU than BFT-SMaRt and 37% less CPU than MinBFT.

7.7.2 Protocol Switch

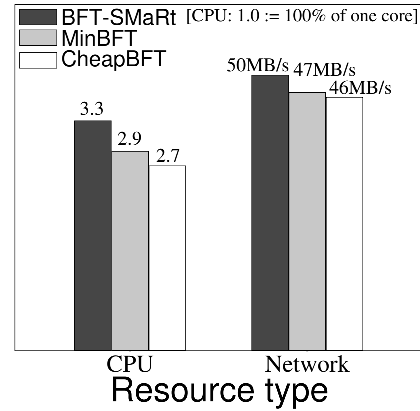
To evaluate the impact of a fault on the performance of CheapBFT, we execute a protocol switch from CheapTiny to MinBFT during a micro benchmark run with 100 clients; the checkpoint interval is set to 200 requests. In this experiment, we trigger the protocol switch shortly before a checkpoint becomes stable in CheapTiny to evaluate the worst-case overhead caused by an abort history of maximum size. Figure 7.13 shows the response times of 1,000 requests handled by CheapBFT around the time the replicas run the CheapSwitch transition protocol. While verifying and processing the abort history, replicas are not able to execute requests, which leads to a temporary service disruption of max. 254 milliseconds. After the protocol switch is complete, the response times drop back to the normal level for MinBFT.

7.7.3 ZooKeeper Use Case

ZooKeeper [HKJR10] is a crash-tolerant coordination service used in large-scale distributed systems for crucial tasks like leader election, synchronization, and failure detection. This section presents an evaluation of a ZooKeeper-like BFT service that rely on BFT-SMaRt, MinBFT, and CheapBFT for fault-tolerant request dissemination, respectively.

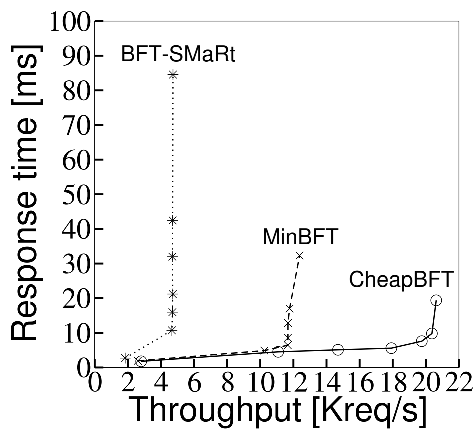


(a) Throughput vs. response time for an increasing number of clients.

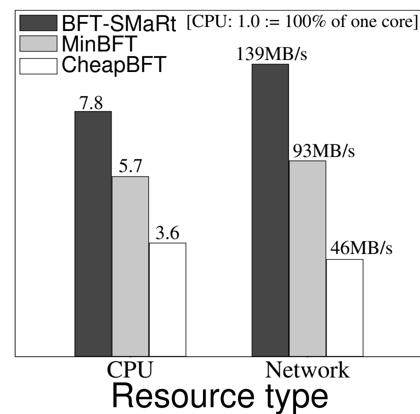


(b) Average resource usage per 10 Kreq/s normalized by throughput.

Figure 7.11: Performance and resource-usage results for a micro benchmark with empty requests and 4 kilobyte replies.



(a) Throughput vs. response time for an increasing number of clients.



(b) Average resource usage per 10 Kreq/s normalized by throughput.

Figure 7.12: Performance and resource-usage results for a micro benchmark with 4 kilobyte requests and empty replies.

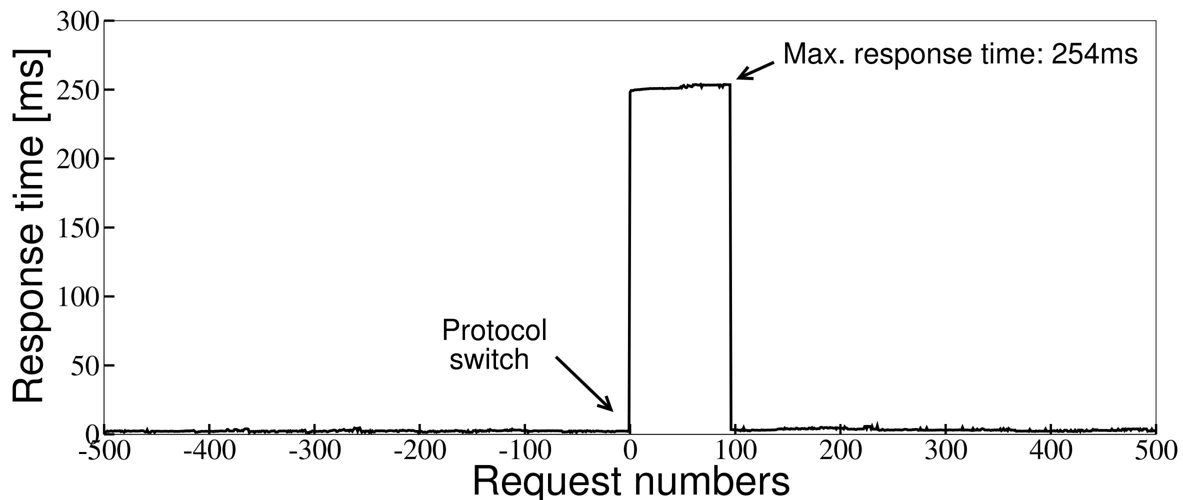


Figure 7.13: Response time development of CheapBFT during a protocol switch from CheapTiny to MinBFT.

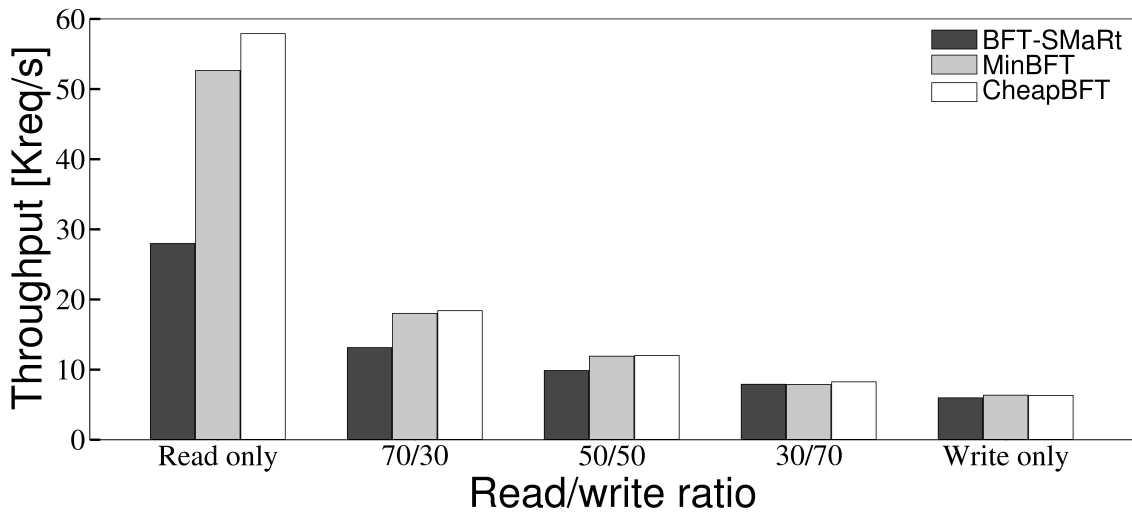
ZooKeeper allows clients to store and retrieve (usually small) chunks of information in data nodes, which are managed in a hierarchical tree structure. We evaluate the three implementations for different mixes of read and write operations. In all cases, 1,000 clients repeatedly access different data nodes, reading and writing data chunks of random sizes between one byte and two kilobytes. Figure 7.14 presents the performance and resource-usage results for this experiment.

The results show that with the execution stage (i. e., the ZooKeeper application) performing actual work (and not just sending replies as in the micro-benchmark experiments of Section 7.7.1), the impact of the agreement protocol on system performance is reduced. In consequence, all three ZooKeeper implementations provide similar throughput for write-heavy workloads. However, the resource footprints significantly differ between variants: in comparison to the MinBFT-based ZooKeeper, the replicas in the CheapBFT-based variant save 7-12% CPU and send 12-20% less data over the network. Compared to the BFT-SMaRt implementation, the resource savings of the CheapBFT-based ZooKeeper add up to 23-42% (CPU) and 27-43% (network).

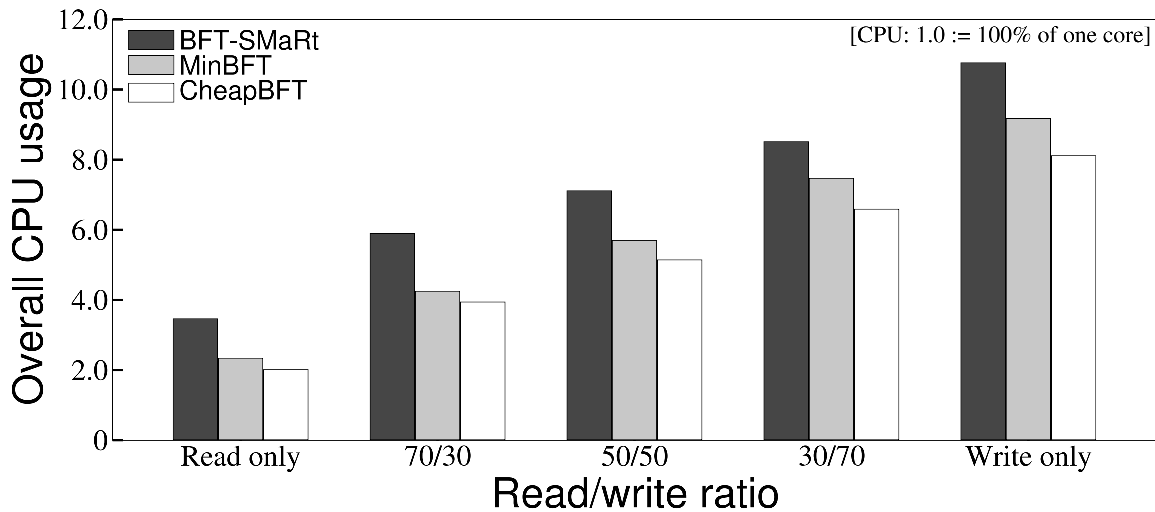
7.8 Discussion

As described in Section 7.5.1, the first PANIC received by a replica triggers the abort of the CheapTiny protocol. In consequence, a single faulty client is able to force a protocol switch, even if all replicas are correct and the network delivers messages in time. In general, we expect such faulty behavior to be rare, as only authenticated clients get access to the system (see Section 7.3.1). Nevertheless, if an authenticated client repeatedly panics, human intervention may be necessary to revoke the access permissions of the client. However, even if it takes some time to remove the client from the system, unnecessary switches to the MinBFT protocol only increase the resource consumption of CheapBFT but do not compromise safety.

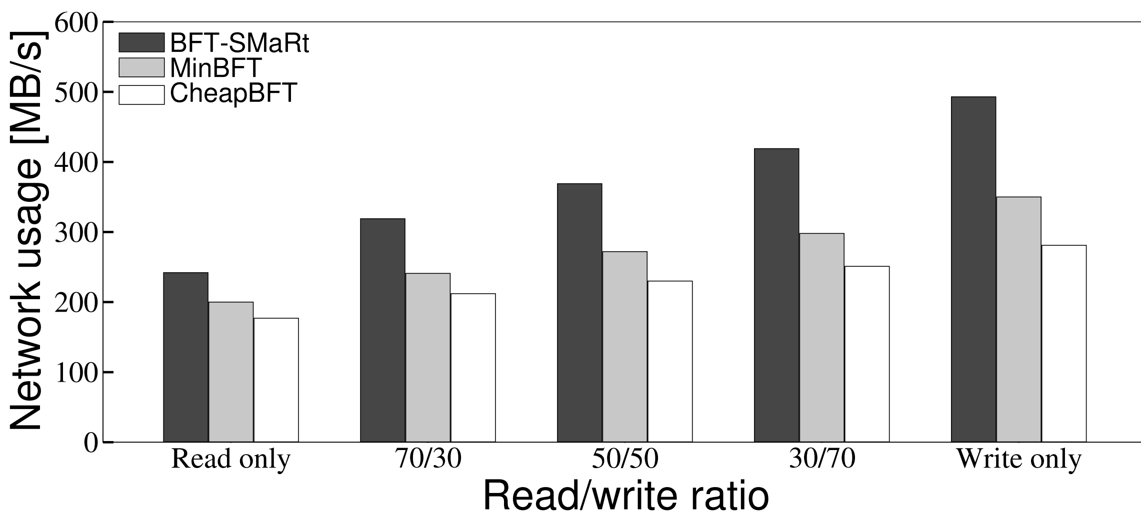
Having completed the CheapSwitch transition protocol, all non-faulty replicas are in a consistent state. Following this, the default procedure in CheapBFT is to run the MinBFT protocol for a certain number of requests before switching back to CheapTiny (see Section 7.6.2). The rationale of this approach is to handle temporary faults and/or short periods of asynchrony which usually affect only a number of subsequent requests. Note that in case such situations



(a) Realized throughput for 1,000 clients.



(b) CPU usage per 10 Kreq/s normalized by throughput.



(c) Network transfer volume per 10 Kreq/s normalized by throughput.

Figure 7.14: Performance and resource-usage results for different BFT variants of our ZooKeeper service for workloads comprising different mixes of read and write operations.

are not characteristic for the particular use-case scenario, different strategies of how to remedy them may be applied. In fact, if faults are typically limited to single requests, for example, it might even make sense to directly start a new instance of CheapTiny after CheapSwitch has been completed.

CheapTiny has a low resource footprint, however, the resource usage is asymmetrically distributed over active and passive replicas. Accordingly, the active replicas, especially the leader, can turn into a bottleneck under high load. This issue can be solved by dynamically alternating the leader role between the active replicas similar to Aardvark [CWA+09] and Spinning [VCBL09]. Furthermore, one could dynamically assign the role of passive and active replicas thereby distributing the load of agreement and execution over all nodes.

7.9 Related Work

Reducing the overhead is a key step to make BFT systems applicable to real-world use cases. Most optimized BFT systems introduced so far have focused on improving time and communication delays, however, and still need $3f + 1$ nodes that actually run agreement as well as execution stage [KAD+09, GKQV10]. Note that this is the same as in the pioneering work of Castro and Liskov [CL02]. The high resource demand of BFT was first addressed by Yin et al. [YMV+03] with their separation of agreement and execution that enables the system to run on only $2f + 1$ execution nodes. In a next step, systems were subdivided in trusted and untrusted components for preventing equivocation; based on a trusted subsystem, these protocols need only $2f + 1$ replicas during the agreement and execution stages [CNV04, CMSK07, RK07]. The trusted subsystems may become as large as a complete virtual machine and its virtualization layer [CNV04, RK07], or may be as small as the trusted counter abstraction [VCB+11, VCBL10].

Subsequently, Wood et al. [WSV+11] presented ZZ, a system that constrains the execution component to $f + 1$ nodes and starts new replicas on demand. However, it requires $3f + 1$ nodes for the agreement task and relies on a trusted hypervisor and a machine-management system. In a previous work, we increased throughput by partitioning request execution among replicas [DK11]. Here, a system relies on a selector component that is co-located with each replica, and no additional trust assumptions are imposed. Moreover, we introduced passive execution nodes in SPARE [DKP+11]; these nodes passively obtain state updates and can be activated rapidly. The system uses a trusted group communication, a virtualization layer, and reliable means to detect node crashes. Of all these works, CheapBFT is the first BFT system that limits the execution *and* agreement components for all requests to only $f + 1$ replicas, whereas only f passive replicas witness progress during normal-case operation. Furthermore, it relies only on a lightweight trusted counter abstraction.

The idea of witnesses has mainly been explored in the context of the fail-stop fault model so far [Par86]. In this regard, CheapBFT is conceptually related to the Cheap Paxos protocol [LM04], in which $f + 1$ main processors perform agreement and can invoke the services of up to f auxiliary processors. In case of processor crashes, the auxiliary processors take part in the agreement protocol and support the reconfiguration of the main processor set.

Related to our approach, Guerraoui et al. [GKQV10] have proposed to optimistically employ a very efficient but less robust protocol and to resort to a more resilient algorithm if needed. CheapBFT builds on this work and is the first to exploit this approach for changing the number of nodes actively involved (rather than only for changing the protocol), with the goal of reducing the system's resource demand.

PeerReview [HKD07] omits replication at all by enabling accountability. It needs a sufficient

number of witnesses for discovering actions of faulty nodes and, more importantly, may detect faults only *after* they have occurred. This is an interesting and orthogonal approach to ours, which aims at tolerating faults. Several other recent works aim at verifying services and computations provided by a single, potentially faulty entity, ranging from database executions [WSS09] and storage integrity [SCC+10] to group collaboration [FZFF10].

7.10 Conclusion

CheapBFT is the first Byzantine fault-tolerant system to use $f + 1$ active replicas for both agreement and execution during normal-case operation. As a result, it offers resource savings compared with traditional BFT systems. In case of suspected or detected faults, replicas run a transition protocol that safely brings all non-faulty replicas into a consistent state and allows the system to switch to a more resilient agreement protocol. CheapBFT relies on the CASH subsystem for message authentication and verification, which advances the state of the art by achieving high performance while comprising a small trusted computing base.

Chapter 8

Tailored Memcached

Chapter Authors:
Klaus Stengel (TUBS)

8.1 Introduction

Many new applications running on top of current cloud systems are designed according to the principles of service oriented architecture (SOA). This means that application software no longer just comprises a single program running in a virtual machine at the cloud provider, but assumes the form of many concurrently running instances. These have to cooperate with each other and they also draw on a large number of simpler low-level services in a distributed environment. Due to this structure, the application becomes more scalable and fault-tolerant, as service instances can be added and removed according to demand.

Unfortunately, those simple services are often implemented on top of comparatively large general-purpose operating systems (e.g. Linux, Windows) and run-time systems (e.g. Java, .NET). While general-purpose systems are very convenient to use, because they offer many features, most of these are unnecessary or even harmful if the goal is to offer a secure and efficient implementation of such simple services. Thus we present an approach to implement this kind of services with minimized runtime overhead and improved security by applying techniques known from tha area of Aspect-Oriented Programming (AOP) to the type-safe, functional programming language Haskell. We demonstrate the approach using a simple in-memory key/value storage system called memcached[Mema], which is commonly used in cloud environments to to cache all sorts of dynamically generated data.

The rest of this chapter is structured as follows: We begin in section 8.2 introducing the memcached service and its protocol and look at some applications to determine how this service is typically used in different scenarios. This allows us to split the features offered by the service into independent subsets that will be enabled only if it's actually required by the application. In the following section 8.3, we will go into more detail regarding the chosen implementation language Haskell and how the implementation can made highly configurable with little effort.

8.2 Memcached Feature Sets

The first step in creating a reconfigurable service is to determine which features of a service are required in which situation. Thus, we will have a short look at the functionality of a full memcached implementation has to offer and what the typical use-cases are. The complete documentation of the protocol is available at their code repository [Memb].

8.2.1 Feature description

At the protocol level, i.e., the way client applications can access the cache, are two supported models. One method is based on human readable text strings, while the other one uses binary codes. Both protocols run over a standard internet TCP/IP connection and support the same set of commands, only the encoding is different.

The memcached uses a simple Key/Value model to store information, where a user-specified key is associated with a value that is also provided by the user. In order to retrieve the stored value, the cache service can be asked for the value associated with the given key. The response will either contain the previously stored value or inform the inquirer that there is no value available for this key. Keys can be any text string, while the values consist of two parts: The first one is just a 32-bit word for keeping status information and the second can contain binary data of arbitrary length.

Besides the usual `put` and `get` operations to store and retrieve information from the cache, the services also support some additional operations. Namely, the put action can be made dependent on the current presence of an value for a given key. Using the `replace` command instead of `put` will exclusively update an existing entry and refuses to create a new one. The opposite behaviour is possible with the `add` operation, which only saves the value if the given key is not assigned yet.

In some use cases it is desirable to update a value only if it has n't changed since we retrived it last time, for example to implement atomic updates. Memcached provides the Compare-and-Swap operation (`cas`) for these situations, which allows wait-free implementation of consensus algorithms [Her91].

While the previous operations only dealt with updates of entire values, it is also possible to extend existing values by adding more data in the beginning or end using `prepend` and `append`. Another for of value manipulation allows the user to interpret the value as a decimal number perform increments or decrement operations (`incr`, `decr`).

As the primary purpose of the memcached service is to provide temporary storage, there are also certain mechanisms available to remove entries from the cache. If the application wants to control the storage duration manually, it can issue a `delete` command, which will cause the specified entry to be removed instantly. Sometimes the cached information is only useful for a certain amount of time (e.g., weather forecasts) and it does not make sense to keep it indefinitely. In this case, the application can delegate the timely deletion to the memcached service itself by specfiying a point in time when the data should disappear when creating an entry. It is also possible to update this expiration date without modifying the data by using the `touch` command. Thus there is no need for the application to track the entries and perform `delete` operations at appropriate times. Additionally, cached data can automatically expire if the server runs out of free memory and reclaims cache entries that weren't used for longer time periods.

8.2.2 Feature groups

Now that we presented an overview of the features the memcached service usually supports, we will now have a look at some common use cases and determine which of these can be made configurable. The results of this analysis are summarized in Table 8.1 and lists and described in the following paragraphs.

First of all, there is the question of which protocol the application actually uses to talk with the memcached service. It makes sense to make the protocol representation configurable, as it is quite unlikely that the same application would want to talk with the caching service using both

Feature set	Supported functions/commands
Binary protocol	Support for all active commands via binary protocol
Text protocol	Support for all active commands via text protocol
Core	get, put
Timed expiration	Cached values are removed depending on system time, touch
Least recently used	Entries will be deleted automatically when memory becomes scarce
Manual expire	delete
Conditional put	add, replace
Atomic update	cas (also enables version counters)
Extending values	append, prepend
Counting	incr, decr

Table 8.1: List of configurable features

ASCII and binary in parallel. However, it should not be forbidden to enable both at once, as some use-cases, like the memcached client module included in the Nginx web server [Ngi], expect the cache to be filled from an external data source. In that particular case, multiple programs access the same cache and might prefer different protocols to do so.

Regarding the actually useful, the minimum core functionality consists of the obligatory get and put operations. These do not specify any mechanism to remove entries from the cache yet, so the cache would just fill up and it would no longer be possible to add more entries after a certain time. Thus, depending on which expiration mode is suitable for the application at hand, it is also necessary to enable at least either the timed, least-recently-used (LRU) or manual delete features.

The remaining features we have to discuss are concerned with the atomic update and manipulation of existing values. These are all entirely optional and only matter if the application requires certain consistency guarantees for data stored in the cache. From an algorithmic perspective, the append and prepend operations are very similar and only differ in the position where the new data has to be inserted. As a result, it makes sense to group both operations into one feature set. The same principle applies to the incr and decr operations, which differ only on the calculation that has to be performed. Unlike the functions to merely attach data to existing values, however, these require program code to process numbers formatted as decimal strings. The cas operation finally deserves its own feature, as it requires additional maintenance of an update counter for each entry.

Now that we presented an overview and categorization of the features a memcached implementation has to offer, we'll discuss various options to implement these in an configurable way in the following section.

8.3 Implementing Variability

The programming language used has a large impact on the techniques that are readily available to design reconfigurable programs. As a result, this section is divided in two parts: The first one will discuss the merits of using the Haskell programming language, while the second part will present a short survey on current research regarding aspect-oriented programming in Haskell.

8.3.1 The Haskell programming language

In the previous deliverable D2.1.1 we already evaluated several different programming languages for their safety properties. While they all have different approaches and reach this goal to certain degrees, this often comes with a lack of flexibility. Language extensions like Frama-C[BCF+10] for the C programming language or the RavenSPARK profile for Ada[CA] improve safety by limiting the expressiveness. Most of the restrictions apply to dynamic allocation of memory, which is either completely forbidden or very hard to make accessible to the static verification these languages provide. This is a reasonable approach for embedded systems, where most computations are signal processing functions that work in datasets of predetermined size. For many cloud services, however, like the memcached we use in our example, such restrictions are too severe. Most data structures the service needs internally need to grow dynamically. Therefore we chose to avoid this problem by looking at the higher level languages that makes explicit memory management unnecessary.

With Haskell, the lower level resource allocation is automatically managed by the run-time system, so the verification can concentrate on the application-specific algorithms. Together with the properties of the Haskell language, purity and laziness, this allows one to write programs that is both configurable and verifyable. Layziness means that any computation is delayed until the actual result of the computation is required to proceed. Purity ensures that each function, given the same parameters and context, will always yield the same result. The laziness allows one to define arbitrary control flow operators within the language, while the purity property enforces the proper encapsulation of side-effects.

Basic control structures, like a if-then-else-statement, can be expressed as basic Lambda-expressions in a lazy environment: `True` can be defined as $\lambda ab. \rightarrow a$ and `False` as $\lambda ab. \rightarrow b$. This definition provides functions that takes two arguments, where the first is the function that should be returned if the value is true (if-part) and the second parameter if the value is false (else-part). The laziness of the language then prevents the program code in the path that is not taken from actually executing. By varying the function binding and employing more advanced concepts like Monads or Arrows, this allows one to create flexible control structures. Such structures will be reviewed in the following section, where we discuss current approches how to write to highly configurable programs.

8.3.2 Aspect-Oriented Programming in Haskell

The basic idea of reconfigurable programs is realized in many widespread software systems and probably the most prominent one today is the Linux kernel. The parts which will be compiled into the kernel and some parameters are derived from a configuration file given at compile time. In Haskell, a similar approach is also taken by the Xmonad[SS07] window manager. It provides the user a configuration file that can be used to set certain paramaters and is just compiled into the actual program. The compiler can then detect that the disabled functions are never called and thus removes them from the program.

Unfortunately, these simple approches often have problems in places where many features have to interact, or if an aspect requires changes spread over large portions of the code base. Each possible combination has either to be implemented manually or the program code contains many conditionally activated code sections. This creates maintenance problems, as it becomes very hard to keep all parts in a consistent state when a change becomes necessary.

As a result, the idea of Aspect Oriented Programming (AOP)[KLM+97] was created to focus exactly on this issue. The general idea is to separate the software into different aspects that arise

from different views on the subject. The interrelationship between those aspects is then described explicitely and used to derive possible configurations for the system. On an implementation level, each aspect forms its own unit with information how they can interact with other ones if they are present.

The approach taken by the popular language extensions, like AspectJ[KHH+01] for Java, is to write special program blocks and a set of rules where they should be weaved into regular program code. Surprisingly, there is not very much literature available on how to achieve similar results using the Haskell programming language. An mostly equivalent concept is available in Haskell with so called *Attribute Grammars*[VSS09]. As shown in the paper, it is even possible to implement these on top of the Haskell language with the help of some commonly available language extensions and libraries. Unfortunately, it has to manage control flow information using dynamic list, which is still hard to optimize for the currently available compilers. As a result, this way of achieveing reconfigurable software is not particularily efficient. Moreover, this style of providing aspect oriented programming was only demonstrated to work on compilers or other programs that mostly operate on tree-like data structures. Therefore it is not always clear how to translate these ideas to a long running network service, like our memcached example, or other, more general applications.

A different way to look at the problem is presented by EffectiveAdvice[OSC10]. The general idea is to start from a translation of object oriented programming into Haskell, where the *this* reference is implemented as a fix-point function. The extension to this model now consists of a special *proceed* function, which can be used to weave in additional functionality. Although the points where these functions can be attached to are no longer invisible (as this is the case with the Attribute Grammars), this approach is easier to understand and more efficient. It also addresses concerns regarding the interaction of features when internal state is involved, so that it becomes possible to restrict the ways state transitions can be made by other aspects.

Comparing the two presented approaches, EffectiveAdvice and Attribute Grammars, the techniques presented by EffectiveAdvice seem to be more practical and also better suited for our tailored memcached service.

8.4 Conclusion

In this chapter we presented the memcached service, a simple Key/Value store and categorized its protocol functions into different feature groups. As evidenced by the table 8.1, even such a relatively simple service can be divided into many different aspects, of which most applications only need a few.

In order to generate minimal instances of such a service, tailored to the application, we need an approch to make our software configurable. Therefore we first had a look at the Haskell programming language and some of its characteristics. While the language is generally very flexible and allows for high levels of abstraction, there doesn't seem to be any generally accepted way how to implement fine-grained features in Haskell. From the area of Aspect Oriented Programming, the approach taken by EffectiveAdvice[OSC10] looks like a promising candidate to be applied to our prototype.

Chapter 9

Log Service

Chapter Authors:
Paolo Smiraglia, Gianluca Ramunno (POL)

In this chapter we present an important component of the cloud infrastructure, the Log Service. The main focus of this component is to log and track events generated at multiple cloud layers (infrastructure, platform, software) with the purpose of increasing the trustworthiness of the whole cloud infrastructure. Particularly it will be presented a draft design and implementation of the service defined in the deliverables D2.1.1 and D2.4.1, that is mainly based on the scheme for secure logging proposed by Schneier and Kelsey in [SK99].

9.1 Background

In this section we present the concepts necessary to understand how the Log Service works. Since this aspect has been already treated in deep in D2.1.1, Chapter 6 and in D2.4.1, Chapter 7, the description will be short in this document.

Schneier and Kelsey propose a scheme for secure logging in a remote environment [SK99]. For brevity, in the sequel we refer to such scheme as "SK".

In SK they identify a trusted server $\mathcal{T}$, a logging machine $\mathcal{U}$ and moderately-trusted person or machine called $\mathcal{V}$ that wants to access and verify the logs. Moreover, they define a log entry creation procedure which is depicted in Figure 9.1.

Such procedure makes possible the immediate identification of log tampering because all log entries are linked in an hash-chain through the element Y_j. Moreover, the integrity of the logs is ensured by including a MAC field (Z_j) in each log entry and the confidentiality of the logged data (D_j) is guaranteed thanks to the usage of a symmetric cryptography mechanism ($E_{K_j}(D_j)$). In SK the log entries are grouped by logging sessions. Each of these is identified by a unique identifier (e.g. UUID) and by a randomly generated authentication key (A_0).

The usage of SK ensures for the generated logs the presence of the *forward integrity* property. Such property implies that, if an attacker succeeds in compromising the log system, he can not modify the log entries collected before his attack without being noticed [BY97].

A subset of key definitions relevant for the comprehension of the current Log Service implementation is listed in the following. For a complete list, refer to D2.1.1, Sections 6.2.1 and 6.2.3.

Terminology

- *Log entry:* a record containing information about one event.
- *Log:* a set of log entries.

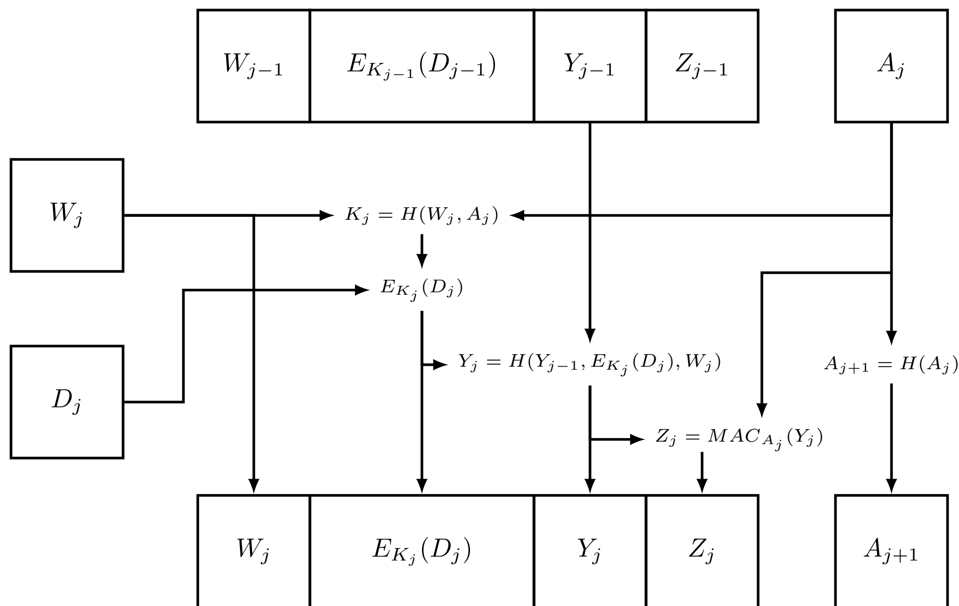


Figure 9.1: Schneier and Kelsey's log entry creation scheme.

- *Log file:* a portion of a log, usually defined to specify a usage/security policy. A Log file could be considered a SK logging session.

Actors

- *Cloud Component:* a component of the cloud infrastructure, intended as a service that can be either provided to the User or internally used by the cloud itself.
- *Log Reviewer:* an external authority with enough privileges to read (part of) the logs. Depending on its privileges, the reviewer will have access to different subsets of the logs.

Functionality

- *Create log:* a Cloud Component creates a log entry and stores it in the log. This operation requires a previous initialization of the log file (logging session).
- *Read logs:* read stands for accessing all logs, possibly subject to a privacy-preserving policy, but usually not restricted to a view on a specific resource
- *Retrieve logs:* the User accesses the logs related to his resources. This functionality can be seen as the composition of a *read* and a filter.
- *Verify logs:* during this step, the forward integrity of the log entries is verified and, therefore, attacks to the logs are detected.

9.2 Design

In this section we present the design of the Log Service. The description will be focused on the building blocks and on the data exchange. The Figure 9.2 depicts an high level view of the Log Service that is focused on the interactions among building blocks.

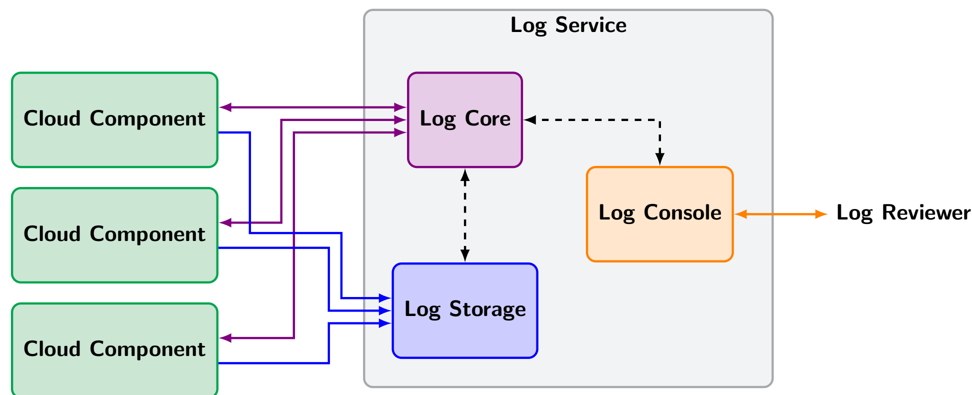


Figure 9.2: Log Service high level view.

9.2.1 Building blocks

Log Core

With the name Log Core we identify the core component of the Log Service. Referring to SK, the Log Core may be considered the $\mathcal{T}$ entity. The Log Core is an application running in a secure location (e.g. on a trusted server) that exports a RESTful service. Through these services, the Cloud Components will be able to initialize a new logging session and the Log Reviewer, via the Log Console, will be able to verify the already opened logging sessions.

Log Storage

The Log Storage is the component of the Log Service that manages the storage of the log entries. Such as sub-component is an application that exports as RESTful service a storage technology (e.g. databse system, replicated file system, etc.). In Figure 9.3 is depicted a Log Storage high level view.

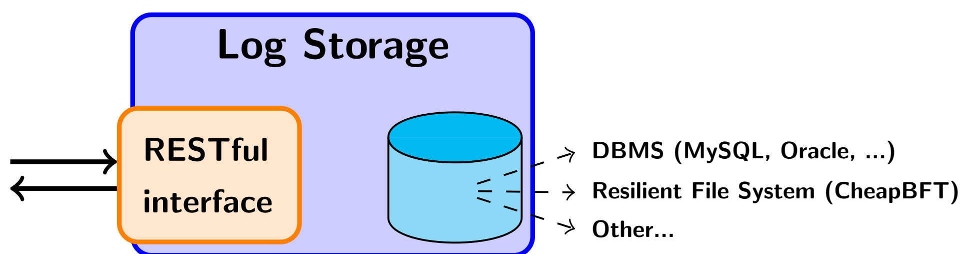


Figure 9.3: Log Storage high level view.

Log Console

The Log Console is the interface used by Log Reviewers to access the logs managed by the Log Service. The operations provided by the Log Console are the retrieve of a list that contains the already opened logging sessions and the possibility to request the verification for one of these. About the verification process, in case of positive result the Log Console provides the Log Reviewers with a temporary URL that can be used to download a *dump* of the just verified logging session. We use the term dump to identify a Log File copy that has been verified.

Cloud Component

A Cloud Component is a generic entity of the cloud that generates log entries. To take advantage from the features provided by the Log Service, each Cloud Component has to include an additional module that makes possible the interaction with the Log Storage and the Log Core (Figure 9.4). An example of integration of the Log Service in a Cloud Component will be presented in Section 9.3.4.

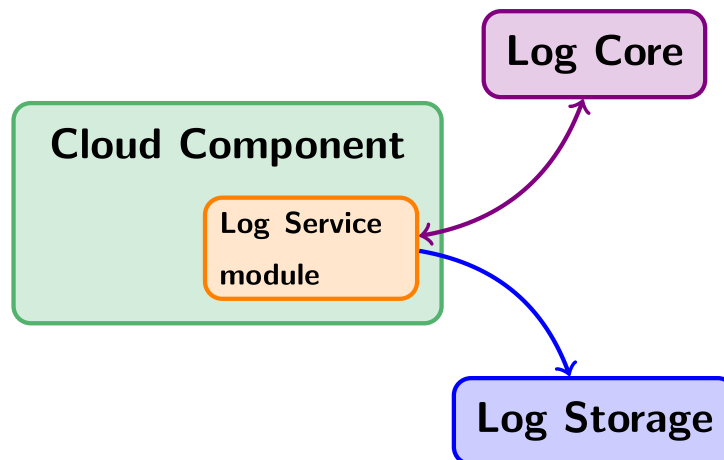


Figure 9.4: Cloud Component with additional Log Service module.

9.2.2 Data exchange

Communications and data exchange among the Log Service building blocks are based on the REpresentation State Transfer (REST) protocol. A RESTful approach makes possible the execution of CRUD operations (Create, Read, Update, Delete) on resources that can be addressed as Universal Resource Locators (URLs). Moreover, a RESTful framework leverages the HyperText Transfer Protocol (HTTP) infrastructure, including caching, referrals, authentication, version control, and secure transport (HTTPS) [NPFS11].

In Log Service data are exchanged using JavaScript Object Notation (JSON) format. We use it because JSON has several properties (language independent, easy for humans to read and write, easy for machines to parse and generate) that make it an ideal data-interchange language [Cro12] and because JSON is the format used for data exchange by the OpenStack API [OC12].

An example of data transferred during the Log file verification process is shown in the following. More in details, in Listing 9.1 it is included an HTTP POST request to trigger the verification of a certain Log file that is generated by the Log Reviewer via the Log Console. In Listing 9.2 instead, it is shown the HTTP response to the previously mentioned HTTP POST request that is generated by the Log Core.

```
POST / logservice HTTP/1.1
Host: localhost:9000
Accept-Encoding: identity
Content-Length: 94
Content-Type: application/json

{
```

```
  "operation": "verifyLogfile",
  "data": {
    "logfile_id": "44d3ae82-c83d-11e1-a555-0025b345ca14"
  }
}
```

Listing 9.1: Example of HTTP POST request to verify a Log file.

```
HTTP/1.0 200 OK
Date: Tue, 10 July 2012 09:57:09 GMT
Server: WSGIServer/0.1 Python/2.7.3rc2
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Content-Length: 104
Content-Type: application/json

{
  "operation": "verifyLogfile",
  "logfile_id": "44d3ae82-c83d-11e1-a555-0025b345ca14",
  "result": "success",
  "dumpurl": "http://www.example.com:9000/logservice/dumps/44d3ae82-c83d-11
     e1-a555-0025b345ca14.txt"
}
```

Listing 9.2: Example of HTTP response to a Log file verification request.

9.3 Implementation

This section is focused on the Log Service implementation details. In the following it will be presented the *state of the art* about the implementation of each building block, the core library written in C and its Python bindings. Moreover, it will be presented also an example of Log Service integration in the open source framework for cloud computing OpenStack.

9.3.1 Core library

The core element of the Log Service is the library for C language called **`libsklog`** [Smi12]. Such library, entirely developed by Politecnico di Torino (POL), has as main objective to provide the application developers with the functionality for secure logging proposed by Schneier and Kelsey in [SK99]. In addition to the log entry generation scheme as depicted in Figure 9.1, the **`libsklog`** library includes into each log entry some additional information in order to follow the *Common Event Expression* (CEE) directives [Cor12]. This feature is provided thanks to the usage of the library **`libumberlog`** [Nag12]. The Figure 9.5 depicts how the original log entries creation scheme has been modified in order to add the CEE directives.

Following the SK roles definition ($\mathcal{U}$, $\mathcal{T}$, $\mathcal{V}$), **`libsklog`** provides functions to implement each role. To differentiate the functions per role, we used a specific naming convention. In particular, functions that have the prefix `SKLOG_U` in their name can be used to implement a $\mathcal{U}$ role. The same way, `SKLOG_T` to implement $\mathcal{T}$ role and `SKLOG_V` to implement $\mathcal{V}$ role.

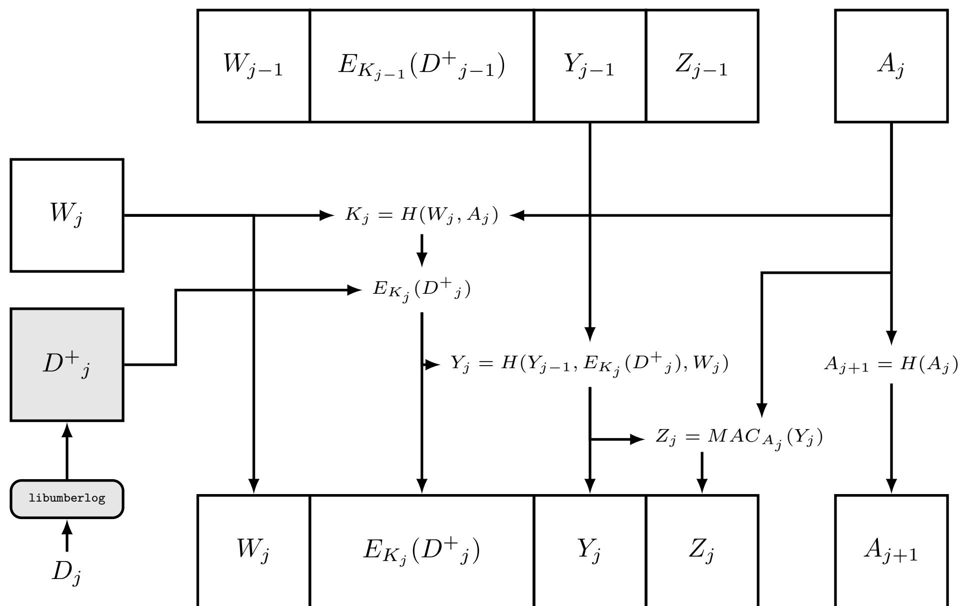


Figure 9.5: Schneier and Kelsey's modified log entry creation scheme.

9.3.2 Bindings

The **libsklog** library is currently provided with bindings for the Python language. Since it is the language used to implement OpenStack, we chose it as binding language for the **libsklog** library in order to simplify the future integration of the Log Service in OpenStack.

Bindings are provided as a Python module called **PySklog**. Following the C library approach, **PySklog** includes the definition of three Python classes, one for each SK role: `Sklog_U` for $\mathcal{U}$ role, `Sklog_T` for $\mathcal{T}$ role and finally `Sklog_V` for $\mathcal{V}$ role. While **libsklog** implements only the SK capabilities, the module **PySklog** implements also the REST communication protocol of the Log Service.

9.3.3 Building blocks implementation

Log Storage

Despite the definition of the Subsection 9.2.1, actually the Log Storage is not implemented as RESTful service. In the current version, to store the log entries the Cloud Components execute a MySQL query on a remote database. This approach could be considered the best approximation to the real behaviour of the Log Storage that will be implemented as REST service as next step.

Log Core

The Log Core is actually a Python script that implements a standalone REST service (Figure 9.6). It is implemented using the Python module **PySklog** to provide the Log Service capabilities and the module **bottlepy** [Hel12] to build the REST service.

Log Console

Actually the Log Console is implemented as Python script that uses **PySklog**. This script can be used by the Log Reviewer to request the verification of a certain Log File. Such a simple

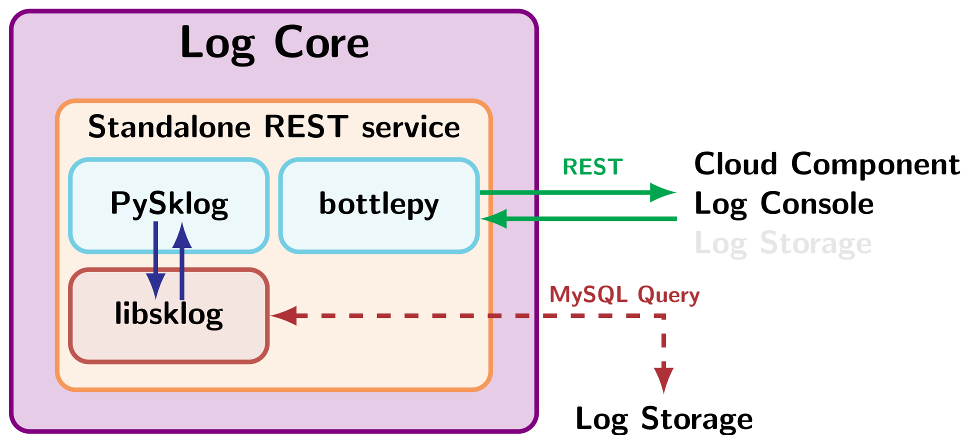


Figure 9.6: Log Core low level architecture.

console, converts the Log Reviewers' requests in REST requests and send them to the Log Core without performing any check about the Log Reviewer permissions.

9.3.4 Integration in OpenStack "Essex"

OpenStack [Ope12] is an open source framework for cloud computing that is implemented in Python. It is the composition of five services: `nova` (computing service), `glance` (imaging service), `swift` (object storage service), `keystone` (identity service) and finally `horizon` (web based cloud management dashboard). The OpenStack logging system (`log.py`) is based on the Python module called **`logging`** [Pyt12a]. Such module makes possible the definition of multiple logging handlers [Pyt12b].

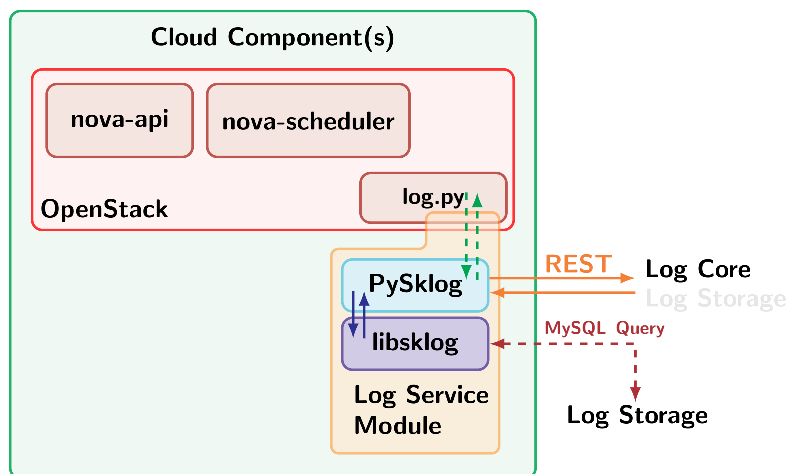


Figure 9.7: Integration of Log Service in OpenStack.

The integration of the Log Service in OpenStack (Figure 9.7) includes the definition of a new logging handler called `SecureLoggingHandler` and the definition of a new group of `nova` configuration flags that is called `securelog`. Such group includes the flags listed in Table 9.1. The Listing 9.3 shows an example of the usage of the `securelog` flags.

```
...
```

Flag	Values	Description
use_secure_log	Boolean	Enable or disable the securelog logging handler
logcore_address	String	Address where Log Core is listening
logcore_port	Integer	Port where Log Core is bound
securelog_logfile	String	Path to the file used to log sessions opening and closure
securelog_services	String	Services that have to log using secure logging

Table 9.1: securelog configuration flags.

```
[securelog]
use_secure_log = True
logcore_address = logcore.example.com
logcore_port = 9000
securelog_logfile = /var/log/nova/securelog.log
securelog_services = nova-api, nova-scheduler

...
```

Listing 9.3: Example of nova.conf using securelog flags group.

Bibliography

[AA08a] Imad M. Abbadi and Muntaha Alawneh. Preventing insider information leakage for enterprises. In *Proceedings of the Second International Conference on Emerging Security Information, Systems and Technologies*. IEEE, 2008.

[AA08b] Muntaha Alawneh and Imad M. Abbadi. Preventing information leakage between collaborating organisations. In *ICEC '08: Proceedings of the tenth international conference on Electronic commerce*, pages 185–194. ACM Press, NY, August 2008.

[AA08c] Muntaha Alawneh and Imad M. Abbadi. Sharing but protecting content against internal leakage for organisations. In *DAS 2008*, volume 5094 of *LNCS*, pages 238–253. Springer-Verlag, Berlin, 2008.

[AA11] Muntaha Alawneh and Imad M. Abbadi. Defining and analyzing insiders and their threats in organizations. In *The 2011 IEEE International Workshop on Security and Privacy in Internet of Things (IEEE SPIoT 2011)*. IEEE, Nov 2011.

[Abb11] Imad M. Abbadi. Clouds infrastructure taxonomy, properties, and management services. In Ajith Abraham, Jaime Lloret Mauri, John F. Buford, Junichi Suzuki, and Sabu M. Thampi, editors, *Advances in Computing and Communications*, volume 193 of *Communications in Computer and Information Science*, pages 406–420. Springer Berlin Heidelberg, 2011.

[ABCS06] Ross Anderson, Mike Bond, Jolyon Clulow, and Sergei Skorobogatov. Cryptographic processors — a survey. *Proceedings of the IEEE*, 94(2):357–369, February 2006.

[ANM11] Imad M. Abbadi, Cornelius Namiluko, and Andrew Martin. Insiders analysis in cloud computing focusing on home healthcare system. In *The 6th International Conference for Internet Technology and Secured Transactions (ICITST-2011)*, pages 350–357. IEEE, December 2011.

[ARM] ARM. TrustZone technology overview. http://www.arm.com/products/security/trustzone/index.html.

[aws] AWS Marketplace: Server Software for Amazon Web Services. https://aws.amazon.com/marketplace/.

[AZN+08] Masoom Alam, Xinwen Zhang, Mohammad Nauman, Tamleek Ali, and Jean-Pierre Seifert. Model-based behavioral attestation. pages 175–184, 2008.

[BBD+10] Ken Barr, Prashanth Bungale, Stephen Deasy, Viktor Gyuris, Perry Hung, Craig Newell, Harvey Tuch, and Bruno Zoppis. The VMware mobile virtualization platform: is that a hypervisor in your pocket? *SIGOPS Operating Systems Review*, 2010.

[BCF+10] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filiatre, Claude Marche, Benjamin Monate, Yannick Moy, and Virgile Prevosto. Acsl: Ansi/iso c specification langauge version 1.5. http://frama-c.com/download/acsl_1.5.pdf, 2010.

[BCG+06] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM: Virtualizing the trusted platform module. In *Proceedings of the 15th USENIX Security Symposium*, pages 305–320, 2006.

[BDD+11a] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. XManDroid: A new Android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Universität Darmstadt, 2011.

[BDD+11b] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry. Practical and lightweight domain isolation on android. In *1st ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM'11)*. ACM, Oct 2011.

[BDF+01] Tom Berson, Drew Dean, Matt Franklin, Diana Smetters, and Michael Spreitzer. Cryptography as a Network Service. In *Internet Society Network and Distributed Systems Security Symposium (NDSS'01)*, 2001.

[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 *19th ACM symposium on Operating systems principles (SOSP'03)*. ACM, 2003.

[BDNP08] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In *15th ACM conference on Computer and communications security (CCS'08)*. ACM, 2008.

[BGHP08] Matt Bishop, Dieter Gollmann, Jeffrey Hunker, and Christian W. Probst. Countering insider threats. In *Dagstuhl Seminar Proceedings 08302*, pages 1–18. RAND Corp., Santa Monica, California, 2008.

[Bit] Bit9. Identify unknown software with Bit9 Global Software Registry. http://www.bit9.com/products/bit9-global-software-registry.php, Accessed: Jun 2006.

[BLCSG12] Shakeel Butt, H. Andres Lagar-Cavilla, Abhinav Srivastava, and Vinod Ganapathy. Self-service cloud computing. In *19th ACM Conference on Computer and Communications Security (CCS'12)*. ACM, October 2012.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *13th European Symposium on Research in Computer Security: Computer Security (ESORICS'08)*. Springer, 2008.

[BNP+11] Sven Bugiel, Stefan Nürnberger, Thomas Pöppelmann, Ahmad-Reza Sadeghi, and Thomas Schneider. AmazonIA: When Elasticity Snaps Back. In *18th ACM Conference on Computer and Communications Security (CCS'11)*. ACM, Oct 2011.

[Bra11] Tony Bradley. DroidDream becomes Android market nightmare. http://www.pcworld.com/businesscenter/article/221247/droiddream_becomes_android_market_nightmare.html, 2011.

[BS] BFT-SMaRt. http://code.google.com/p/bft-smart/.

[BY97] Mihir Bellare and Bennet S. Yee. Forward integrity for secure audit logs. Technical report, 1997.

[CA] Rod Chapman and Peter Amey. Spark 95 – the spade ada 95 kernel. http://www.altran-praxis.com/downloads/SPARK/technicalReferences/SPARK95_RavenSPARK.pdf.

[Cac01] Christian Cachin. Distributing trust on the Internet. In *Proceedings of the Conference on Dependable Systems and Networks*, pages 183–192, 2001.

[Car10] Paul Carton. New burst of momentum for Google Android OS. http://www.investorplace.com/18151/google-android-os-major-corporate-smart-phone-winner/, 2010.

[CDE+10] L. Catuogno, A. Dmitrienko, K. Eriksson, D. Kuhlmann, G. Ramunno, A.R. Sadeghi, S. Schulz, M. Schunter, M. Winandy, and J. Zhan. Trusted Virtual Domains–Design, Implementation and Lessons Learned. *Trusted Systems*, pages 156–179, 2010.

[CFH+05] C. Clark, K. Fraser, S. Hand, J.G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX, 2005.

[CL02] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[CLL+06] Liqun Chen, Rainer Landfermann, Hans Löhr, Markus Rohe, Ahmad-Reza Sadeghi, and Christian Stüble. A protocol for property-based attestation. pages 7–16, 2006.

[Clo10] Cloud Security Alliance (CSA). Top threats to cloud computing, version 1.0. http://www.cloudsecurityalliance.org/topthreats/csathreats.v1.0.pdf, March 2010.

[CMSK07] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of 21st Symposium on Operating Systems Principles*, pages 189–204, 2007.

[CMW+08] Allen Clement, Mirco Marchetti, Edmund Wong, Lorenzo Alvisi, and Mike Dahlin. BFT: The time is now. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, pages 1–4, 2008.

[CNC10] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. CRePE: Context-related policy enforcement for Android. In *13th Information Security Conference (ISC)*, 2010.

[CNV04] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd Symposium on Reliable Distributed Systems*, pages 174–183, 2004.

[Cor12] The MITRE Corporation. Common Event Expression: CEE, A Standard Log Language for Event Interoperability in Electronic Systems. Retrieved from: http://cee.mitre.org, June 2012.

[Cro12] Douglas Crockford. JavaScript Object Notation (JSON). Retrieved form: http://json.org, July 2012.

[CRS+11] Emanuele Cesena, Gianluca Ramunno, Roberto Sassu, Davide Vernizzi, and Antonio Lioy. On scalability of remote attestation. In *STC '11 Proceedings of the sixth ACM workshop on Scalable trusted computing*. ACM, Dec 2011.

[CS11] Yao Chen and Radu Sion. To cloud or not to cloud?: musings on costs and viability. In *2nd ACM Symposium on Cloud Computing (SOCC'11)*. ACM, 2011.

[CVEa] CVE-2007-4993. Bug in pygrub allows guests to execute commands in dom0.

[CVEb] CVE-2007-5497. Integer overflows in e2fsprogs allows remote attackers to execute arbitrary code.

[CVEc] CVE-2008-1943. Buffer overflow in xensource allows to execute arbitrary code.

[CWA+09] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation*, pages 153–168, 2009.

[DDKW11] Lucas Davi, Alexandra Dmitrienko, Christoph Kowalski, and Marcel Winandy. Trusted virtual domains on OKL4: Secure information sharing on smartphones. In *6th ACM Workshop on Scalable Trusted Computing (STC)*, 2011.

[DDSW10] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on Android. In *13th Information Security Conference (ISC)*, 2010.

[DEK+09] Alexandra Dmitrienko, Konrad Eriksson, Dirk Kuhlmann, Gianluca Ramunno, Ahmad-Reza Sadeghi, Steffen Schulz, Matthias Schunter, Marcel Winandy, Luigi Catuogno, and Jing Zhan. Trusted Virtual Domains – design, implementation and lessons learned. In *INTRUST*, 2009.

[DGLI10] Alessandro Distefano, Antonio Grillo, Alessandro Lentini, and Giuseppe F. Italiano. SecureMyDroid: enforcing security in the mobile devices lifecycle. In *ACM CSIIRW*, 2010.

[DK11] Tobias Distler and Rüdiger Kapitza. Increasing performance in Byzantine fault-tolerant systems with on-demand replica consistency. In *Proceedings of the 6th EuroSys Conference*, pages 91–105, 2011.

[DKP+11] T. Distler, R. Kapitza, I. Popov, H. P. Reiser, and W. Schröder-Preikschat. SPARE: Replicas on hold. In *Proceedings of the 18th Network and Distributed System Security Symposium*, pages 407–420, 2011.

[DLP+01] Joan G. Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert van Doorn, Sean W. Smith, and Steve Weingart. Building the IBM 4758 secure coprocessor. *IEEE Computer*, 34(10):57–66, October 2001.

[DMKC11] Boris Danev, Ramya Jayaram Masti, Ghassan O. Karame, and Srdjan Capkun. Enabling secure VM-vTPM migration in private clouds. In *27th Annual Computer Security Applications Conference (ACSAC'11)*. ACM, 2011.

[DT] Numaguchi Daisuke and Giuseppe La Tona. Tomoyo-android: TOMOYO Linux on Android. http://code.google.com/p/tomoyo-android/.

[DY83] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[EB09] Jan-Erik Ekberg and Sven Bugiel. Trust in a small package: Minimized MRTM software implementation for mobile secure environments. In *4th ACM Workshop on Scalable Trusted Computing (STC)*, 2009.

[EGC+10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[EGP+07] Thomas Eisenbarth, Tim Güneysu, Christof Paar, Ahmad-Reza Sadeghi, Dries Schellekens, and Marko Wolf. Reconfigurable trusted computing in hardware. In *Proceedings of the 2007 Workshop on Scalable Trusted Computing*, pages 15–20, 2007.

[EL08] Paul England and Jork Loeser. Para-virtualized TPM sharing. In *Proceedings of the 1st International Conference on Trusted Computing and Trust in Information Technologies*, pages 119–132, 2008.

[Eng08] Paul England. Practical techniques for operating system attestation. In Peter Lipp, Ahmad-Reza Sadeghi, and Klaus-Michael Koch, editors, *Trusted Computing - Challenges and Applications*, volume 4968 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin / Heidelberg, 2008.

[ent] enterproid. http://www.enterproid.com/.

[EOM09a] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *16th ACM Conference on Computer and Communications Security (CCS)*, 2009.

[EOM09b] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding Android security. *IEEE Security and Privacy Magazine*, 2009.

[FWM+11] Adrienne Porter Felt, Helen Wang, Alex Moshchuk, Steven Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011.

[FZFF10] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, pages 337–350, 2010.

[Gar11] Gartner Inc. http://www.gartner.com/it/page.jsp?id=1689814, 2011.

[Gen09] C. Gentry. Fully homomorphic encryption using ideal lattices. In *41st annual ACM symposium on Theory of Computing*, pages 169–178. ACM, 2009.

[GJP+05] John Linwood Griffin, Trent Jaeger, Ronald Perez, Reiner Sailer, Leendert Van Doorn, and RamÃşn CÃąceres. Trusted Virtual Domains: Toward secure distributed services. In *First Workshop on Hot Topics in System Dependability (Hotdep'05)*. IEEE, 2005.

[GKQV10] Rashid Guerraoui, Nikola Knezevic, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. In *Proceedings of the 5th EuroSys Conference*, pages 363–376, 2010.

[Goo] Google Inc. Google Android. http://www.android.com/.

[Goo10a] Dan Goodin. Android bugs let attackers install malware without warning. http://www.theregister.co.uk/2010/11/10/android_malware_attacks/, 2010.

[Goo10b] Google. The Android developer's guide - Android Manifest permissions. http://developer.android.com/reference/android/Manifest.permission.html, 2010.

[GPC+03] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *19th ACM symposium on Operating systems principles (SOSP'03)*. ACM, 2003.

[Hel12] Marcel Hellkamp. Bottle: Python Web Framework. Retrieved form: http://bottlepy.org/docs/dev/, July 2012.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, January 1991.

[HHT04] Toshiharu Harada, Takashi Horie, and Kazuo Tanaka. Task Oriented Management Obviates Your Onus on Linux. In *Linux Conference*, 2004.

[HKD07] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the 21st Symposium on Operating Systems Principles*, pages 175–188, 2007.

[HKJR10] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, pages 145–158, 2010.

[HSH+08] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones. In *IEEE CCNC*, January 2008.

[KAD+09] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Transactions on Computer Systems*, 27(4):1–39, 2009.

[KBC+12] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. CheapBFT: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*, EuroSys '12, pages 295–308, New York, NY, USA, April 2012. ACM.

[KD04] Ramakrishna Kotla and Mike Dahlin. High throughput Byzantine fault tolerance. In *Proceedings of the 2004 Conference on Dependable Systems and Networks*, pages 575–584, 2004.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of aspectj. In *ECOOP 2001 – Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354. Springer Berlin / Heidelberg, 2001.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97 – Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin / Heidelberg, 1997.

[KR09] Petr Kuznetsov and Rodrigo Rodrigues. BFTW3: Why? When? Where? Workshop on the theory and practice of Byzantine fault tolerance. *SIGACT News*, 40(4):82–86, 2009.

[KSS07] Ulrich Kühn, Marcel Selhorst, and Christian Stüble. Realizing property-based attestation and sealing with commonly available hard- and software. pages 50–57, 2007.

[LBOR09] Yung-Chuan Lee, Stephen Bishop, Hamed Okhravi, and Shahram Rahimi. Information leakage detection in distributed systems using software agents. In *Proceedings of the International Conference on Intelligent Agents*, pages 128–135. IEEE, 2009.

[LDLM09] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation*, pages 1–14, 2009.

[LM04] Leslie Lamport and Mike Massa. Cheap Paxos. In *Proceedings of the Conference on Dependable Systems and Networks*, pages 307–314, 2004.

[LM09] John Lyle and Andrew Martin. On the feasibility of remote attestation for web services. In *Proceedings of the 2009 International Conference on Computational Science and Engineering - Volume 03*, pages 283–288, Washington, DC, USA, 2009. IEEE Computer Society.

[Loo10] Lookout Mobile Security. Security alert: Geinimi, sophisticated new Android Trojan found in wild. http://blog.mylookout.com/2010/12/geinimi_trojan/, 2010.

[LRW10] Anthony Lineberry, David Luke Richardson, and Tim Wyatt. These aren't the permissions you're looking for. BlackHat USA 2010. http://dtors.files.wordpress.com/2010/08/blackhat-2010-slides.pdf, 2010.

[MC05] Microsoft Corporation. Microsoft Windows Rights Management Services, 2005. http://download.microsoft.com/download/8/d/9/8d9dbf4a3b0d4ea1905b-92c57086910b/RMSTechOverview.doc.

[Mema] Memcached contributors. A distributed memory object caching system. http://memcached.org/.

[Memb] Memcached contributors. Protocol documentation. http://github.com/memcached/memcached/blob/master/doc/protocol.txt.

[MG] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing.

[MLQ+10] J.M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy (SP'10)*. IEEE, 2010.

[MMH08] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. Improving xen security through disaggregation. In *4th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE'08)*. ACM, 2008.

[MPP+08] J.M. McCune, B.J. Parno, A. Perrig, M.K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*. ACM, 2008.

[Nag12] Gergely Nagy. Libumberlog GitHub repository. Retrieved form: https://github.com/algernon/libumberlog, July 2012.

[Nat] National Security Agency. Security-Enhanced Linux. http://www.nsa.gov/research/selinux.

[Ngi] Nginx contributors. ngx_http_memcached_module. http://nginx.org/en/docs/http/ngx_http_memcached_module.html.

[Nil10] Nils. Building Android sandcastles in Android's sandbox. BlackHat Abu Dhabi 2010. https://media.blackhat.com/bh-ad-10/Nils/Black-Hat-AD-2010-android-sandcastle-wp.pdf, 2010.

[NKZ10] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In *ACM ASIACCS*, 2010.

[NKZS10] Mohammad Nauman, Sohail Khan, Xinwen Zhang, and Jean-Pierre Seifert. Beyond kernel-level integrity measurement: Enabling remote attestation for the Android platform. In *TRUST*, 2010.

[NPFS11] Andrew Newton, Dave Piscitello, Benedetto Fiorelli, and Steve Sheng. A RESTful Web Service for Internet Name and Address Directory Services. In *USENIX;login:*, pages 23 – 32. 2011.

[Obe10] Jon Oberheide. Android Hax. SummerCon 2010. http://jon.oberheide.org/files/summercon10-androidhax-jonoberheide.pdf, 2010.

[OBM10] Machigar Ongtang, Kevin Butler, and Patrick McDaniel. Porscha: Policy oriented secure content handling in Android. In *ACSAC*, 2010.

[OBRL09] Hamed Okhravi, Stephen Bishop, Shahram Rahimi, and Yung-Chuan Lee. A MA-based system for information leakage detection in distributed systems. In *Emerging Technologies, Robotics and Control Systems*. 3rd edition, 2009.

[OC12] OpenStack Community. OpenStack API Documentation. Retrieved form: http://api.openstack.org/, July 2012.

[OHL+11] J. Ott, E. Hyytia, P. Lassila, T. Vaegs, and J. Kangasharju. Floating content: Information sharing in urban areas. In *2011 IEE International Conference on Pervasive Computing and Communications (PerCom'11)*, 2011.

[OMEM09] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in Android. In *ACSAC*, 2009.

[Ope] Open Kernel Labs. Ok:android. http://www.ok-labs.com/products/ok-android.

[Ope12] OpenStack Foundation. Openstack Open Source Cloud Computing Software. Retrieved form: http://openstack.org, July 2012.

[Ora08] Oracle. Information rights management — managing information everywhere it is stored and used, June 2008. http://www.oracle.com/technology/products/content-management/irm/IRMtechnicalwhitepaper.pdf.

[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.

[Pal05] Palm Source, Inc. Open Binder. Version 1. http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html, 2005.

[Par86] Jehan-Francois Paris. Voting with witnesses: A consistency scheme for replicated files. In *Proceedings of the 6th Int'l Conference on Distributed Computing Systems*, pages 606–612, 1986.

[PSHW04] Jonathan Poritz, Matthias Schunter, Els Van Herreweghen, and Michael Waidner. Property attestation—scalable and privacy-friendly security assessment of peer computers. Technical Paper RZ3548, IBM Research, 2004.

[Pyt12a] Python Software Foundation. 15.7. logging – Logging facility for Python – Python v2.7.3 documentation. Retrieved form: http://docs.python.org/library/logging.html, July 2012.

[Pyt12b] Python Software Foundation. 15.9. logging.handlers – Logging handlers – Python v2.7.3 documentation. Retrieved form: http://docs.python.org/library/logging.handlers.html, July 2012.

[Qum06] Qumranet. KVM: Kernel-based Virtualization Driver, 2006.

[RC11] Francisco Rocha and Miguel Correia. Lucy in the sky without diamonds: Stealing confidential data in the cloud. In *41st International Conference on Dependable Systems and Networks Workshops (DSNW'11)*. IEEE Computer Society, 2011.

[Ric07] Robert Richardson. The 12th annual computer crime and security survey, 2007. http://i.cmpnet.com/v2.gocsi.com/pdf/CSISurvey2007.pdf.

[RJ09] Vikhyath Rao and Trent Jaeger. Dynamic mandatory access control for multiple stakeholders. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2009.

[RK07] H. P. Reiser and R. Kapitza. Hypervisor-based efficient proactive recovery. In *Proceedings of the 26th Symposium on Reliable Distributed Systems*, pages 83–92, 2007.

[RSA04] RSA Laboratories. PKCS #11 v2.20: Cryptographic Token Interface Standard. Available from http://www.rsa.com/rsalabs/, 2004.

[RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[Rus81] J. M. Rushby. Design and verification of secure systems. In *8th ACM Symposium on Operating System Principles (SOSP)*, 1981.

[SA05] Jay Srage and Jerome Azema. M-Shield mobile security technology, 2005. TI White paper. http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf.

[SCC+10] Alexander Shraer, Christian Cachin, Asaf Cidon, Idit Keidar, Yan Michalevsky, and Dani Shaket. Venus: Verification for untrusted cloud storage. In *Proceedings of the 2010 Workshop on Cloud Computing Security*, pages 19–30, 2010.

[SCP+02] C.P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M.S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. *ACM SIGOPS Operating Systems Review*, 36(SI):377–390, 2002.

[SCSJM07] L. St. Clair, J. Schiffman, T. Jaeger, and P. McDaniel. Establishing and sustaining system integrity via root of trust installation. In *ACSAC 2007: 23rd Annual Computer Security Applications Conference*, pages 19–29, 2007.

[SFE10] Asaf Shabtai, Yuval Fledel, and Yuval Elovici. Securing Android-powered mobile devices using SELinux. *IEEE Security and Privacy Magazine*, 2010.

[SGR09] N. Santos, K.P. Gummadi, and R. Rodrigues. Towards trusted cloud computing. In *Hot topics in cloud computing (HotCloud'09)*. USENIX, 2009.

[SJV+05] Reiner Sailer, Trent Jaeger, Enriquillo Valdez, Ramon Caceres, Ronald Perez, Stefan Berger, John Linwood Griffin, and Leendert van Doorn. Building a mac-based security architecture for the xen open-source hypervisor. In *21st Annual Computer Security Applications Conference (ACSAC'05)*. IEEE, 2005.

[SJZvD04] Reiner Sailer, Trent Jaeger, Xiaolan Zhang, and Leendert van Doorn. Attestation-based policy enforcement for remote access. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 308–317, New York, NY, USA, 2004. ACM.

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

[SK10] Udo Steinberg and Bernhard Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th EuroSys Conference*, pages 209–222, 2010.

[SMB+10] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary CPU architectures. In *USENIX Security Symposium*, 2010.

[Smi12] Smiraglia. Libsklog GitHub repository. Retrieved form: https://github.com/psmiraglia/Libsklog, September 2012.

[SMV+10] Joshua Schiffman, Thomas Moyer, Hayawardh Vijayakumar, Trent Jaeger, and Patrick McDaniel. Seeding clouds with trust anchors. In *ACM workshop on Cloud computing security (CCSW'10)*. ACM, 2010.

[SRGS12] Nuno Santos, Rodrigo Rodrigues, Krishna P. Gummadi, and Stefan Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *21nd conference on USENIX Security Symposium*. USENIX, 2012.

[SS04] Ahmad-Reza Sadeghi and Christian Stüble. Property-based attestation for computing platforms: caring about properties, not mechanisms. In *Workshop on New security paradigms (NSPW'04)*. ACM, 2004.

[SS07] Don Stewart and Spencer Sjanssen. Xmonad. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, Haskell '07, pages 119–119, New York, NY, USA, 2007. ACM.

[SS08] Mario Strasser and Heiko Stamer. A software-based trusted platform module emulator. In *Proceedings of the 1st International Conference on Trusted Computing and Trust in Information Technologies*, pages 33–47, 2008.

[SSC10] Edward Stott, Pete Sedcole, and Peter Cheung. Fault tolerance and reliability in field-programmable gate arrays. *IET Computers & Digital Techniques*, 4(3):196–210, 2010.

[SSFG10] Marcel Selhorst, Christian Stüble, Florian Feldmann, and Utz Gnaida. Towards a trusted mobile desktop. In *TRUST*, 2010.

[SSW08] Ahmad-Reza Sadeghi, Christian Stüble, and Marcel Winandy. Property-based TPM virtualization. In *11th International Conference on Information Security (ISC'08)*, volume 5222. Springer, 2008.

[SWS+07] Ahmad-Reza Sadeghi, Marko Wolf, Christian Stüble, N. Asokan, and Jan-Erik Ekberg. Enabling fairer digital rights management with trusted computing. In *10th International Conference on Information Security (ISC'07)*. Springer, 2007.

[SZ05] Fred B. Schneider and Lidong Zhou. Implementing trustworthy services using replicated state machines. *IEEE Security & Privacy Magazine*, 3:34–43, 2005.

[SZJvD04] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based Integrity Measurement Architecture. pages 223–238, 2004.

[SZZ+11] Roman Schlegel, Kehuan Zhang, Xiaoyong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *18th Annual Network and Distributed System Security Conference (NDSS)*, 2011.

[the] The Cloud Market. http://thecloudmarket.com/.

[Thi10] Samuel Thibault. Stub domains: A step towards dom0 disaggregation. http://www.xen.org/files/xensummitboston08/SamThibault_XenSummit.pdf, 2010.

[Tru] Trusted Computing Group. Mobile Trusted Module Specification. Version 1.0 Revision 6, 26 June 2008.

[Tru07] Trusted Computing Group. TCG Specification Architecture Overview Revision 1.4. https://www.trustedcomputinggroup.org, 2007.

[Tru08] Trusted Computing Group (TCG). Trusted platform module specifications. Available from http://www.trustedcomputinggroup.org, 2008.

[Tru09] Trusted Computing Group (TCG). *TNC Architecture for Interoperability, Version 1.4, Revision 4*, 2009.

[VCB+11] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Veríssimo. Efficient Byzantine fault tolerance. *IEEE Transactions on Computers*, 2011.

[VCBL09] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *Proceedings of the 28th Symposium on Reliable Distributed Systems*, pages 135–144, 2009.

[VCBL10] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. EBAWA: Efficient Byzantine agreement for wide-area networks. In *Proceedings of the 12th Symposium on High-Assurance Systems Engineering*, pages 10–19, 2010.

[VDJ10] Marten Van Dijk and Ari Juels. On the impossibility of cryptography alone for privacy-preserving cloud computing. In *5th USENIX conference on Hot topics in security (HotSec'10)*. USENIX, 2010.

[vDRSD07] Marten van Dijk, Jonathan Rhodes, Luis F. G. Sarmenta, and Srinivas Devadas. Offline untrusted storage with immediate detection of forking and replay attacks. In *2007 ACM workshop on Scalable trusted computing (STC'07)*. ACM, 2007.

[VMw09] VMware. VMware ESX and VMware ESXi, 2009.

[VSS09] Marcos Viera, S. Doaitse Swierstra, and Wouter Swierstra. Attribute grammars fly first-class: how to do aspect oriented programming in haskell. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 245–256, New York, NY, USA, 2009. ACM.

[Win08] Johannes Winter. Trusted computing building blocks for embedded linux-based ARM trustzone platforms. In *3rd ACM Workshop on Scalable Trusted Computing (STC)*, 2008.

[WJ10] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *2010 IEEE Symposium on Security and Privacy (SP'10)*. IEEE, 2010.

[WJW12] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. The xen-blanket: virtualize once, run everywhere. In *7th ACM european conference on Computer Systems (EuroSys'12)*. ACM, 2012.

[WSS09] Peter Williams, Radu Sion, and Dennis Shasha. The blind stone tablet: Outsourcing durability to untrusted parties. In *Proceedings of the 16th Network and Distributed System Security Symposium*, 2009.

[WSV+11] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. ZZ and the art of practical BFT execution. In *Proceedings of the 6th EuroSys Conference*, pages 123–138, 2011.

[XS07] Shouhuai Xu and Ravi Sandhu. A Scalable and Secure Cryptographic Service. In *21st annual IFIP WG 11.3 working conference on Data and applications security*. Springer, 2007.

[YMV+03] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the 19th Symposium on Operating Systems Principles*, pages 253–267, 2003.

[ZAS07] Xinwen Zhang, Onur Acıiçmez, and Jean-Pierre Seifert. A trusted mobile phone reference architecture via secure kernel. In *2nd ACM Workshop on Scalable Trusted Computing (STC)*, 2007.

[ZCCZ11] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. ACM, 2011.

[ZMRG07] Xiaolan Zhang, Suzanne McIntosh, Pankaj Rohatgi, and John Griffin. XenSocket: A High-Throughput Interdomain Transport for Virtual Machines. In *Middleware 2007*, volume 4834 of *Lecture Notes in Computer Science*, pages 184–203. Springer, 2007.

[ZSA10] Xinwen Zhang, Jean-Pierre Seifert, and Onur Acıiçmez. SEIP: simple and efficient integrity protection for open mobile platforms. In *12th International Conference on Information and Communications Security (ICICS)*, 2010.