Trust but Verify

Trust Establishment Mechanisms in Infrastructure Clouds

Paladi, Nicolae

2017

Document Version:
Publisher's PDF, also known as Version of record

Trust but Verify

Trust but Verify
Trust Establishment Mechanisms in Infrastructure Clouds

by Nicolae Paladi

LUNDS UNIVERSITET
Lunds Tekniska Högskola

Thesis for the degree of Doctor of Engineering

Thesis advisors: Prof. Ben Smeets
Assoc.Prof. Christian Gehrmann

Faculty opponent: Dr. Stefan Saroiu

To be presented, with the permission of the Faculty of Engineering of Lund University, for public criticism in the E:1406 lecture hall at the Department of Electrical and Information Technology on September 29, 2017 at 13.00.

Organization
LUND UNIVERSITY

Department of Electrical and Information Technology
Box 118
SE–221 00 LUND
Sweden

Document name
DOCTORAL DISSERTATION

Date of disputation
2017-09-29

Sponsoring organization

Author(s)
Nicolae Paladi

Title and subtitle
Trust but Verify: Trust Establishment Mechanisms in Infrastructure Clouds

Abstract
In the cloud computing service model, users consume computation resources provided through the Internet, often without any awareness of the cloud service provider that owns and operates the supporting hardware infrastructure. This marks an important change compared to earlier models of computation, for example when such supporting hardware infrastructure was under the control of the user. Given the ever increasing importance of computing, the shift to cloud computing raises several challenging issues, which include protecting the computation and ancillary resources such as network communication and the stored or produced data.

While the potential risks for data isolation and confidentiality in cloud infrastructure are somewhat known, they are obscured by the convenience of the service model and claimed trustworthiness of cloud service providers, backed by reputation and contractual agreements. Ongoing research on cloud infrastructure has the potential to strengthen the security guarantees of computation, data and communication for users of cloud computing. This thesis is part of such research efforts, focusing on assessing the trustworthiness of components of the cloud network infrastructure and cloud computing infrastructure and controlling access to data and network resources and addresses select aspects of cloud computing security.

The contributions of the thesis include mechanisms to verify or enforce security in cloud infrastructure. Such mechanisms have the potential to both help cloud service providers strengthen the security of their deployments and empower users to obtain guarantees regarding security aspects of service level agreements. By leveraging functionality of components such as the Trusted Platform Module, the thesis presents mechanisms to provide user guarantees regarding integrity of the computing environment and geographic location of plaintext data, as well as to allow users maintain control over the cryptographic keys for integrity and confidentiality protection of data stored in remote infrastructure. Furthermore, the thesis leverages recent innovations for platform security such as Software Guard Extensions to introduce mechanisms to verify the integrity of the network infrastructure in the Software-Defined Networking model. A final contribution of the thesis is an access control mechanism for access control of resources in the Software-Defined Networking model.

Key words
cloud computing infrastructure, security, trust

Classification system and/or index terms (if any)

Supplementary bibliographical information

Language
English

ISSN and key title
1654-790X

ISBN
978-91-7753-329-0
978-91-7753-330-6 (pdf)

Recipient's notes

Number of pages
224

Price

Security classification

Trust but Verify

Trust Establishment Mechanisms in Infrastructure Clouds

by Nicolae Paladi

LUNDS UNIVERSITET
Lunds Tekniska Högskola

A doctoral thesis at a university in Sweden takes either the form of a single, cohesive research study (monograph) or a summary of research papers (compilation thesis), which the doctoral student has written alone or together with one or several other author(s).

In the latter case the thesis consists of two parts. An introductory text puts the research work into context and summarizes the main points of the papers. Then, the research publications themselves are reproduced, together with a description of the individual contributions of the authors. The research papers may either have been already published or are manuscripts at various stages (in press, submitted, or in draft).

Funding information: This thesis was financially supported by project *InfraCloud* funded by Vinnova under grant agreement No 2012-01519, project *5G-ENSURE* part of European Union's Horizon 2020 research and innovation framework programme under grant agreement No 671562, and a direct assignment from Ericsson AB.

Several research visits conducted during the work on this thesis were made possible with the generous support from the Ericsson Research Foundation. Participation at several OpenStack Summit events was made possible with the travel support from the OpenStack Foundation.

Nicolae Paladi
Department of Electrical and Information Technology
Electrical Engineering
Lund University
P.O. Box 118, 221 00 Lund, Sweden
ISSN: 1654-790X, No: 104
ISBN: 978-91-7753-329-0 (print)
ISBN: 978-91-7753-330-6 (pdf)

Security Lab
RISE SICS
P.O Box 1263
SE-164 29 Kista, Sweden
SICS Dissertation Series 77
ISSN: 1101-1335

Printed in Sweden by Tryckeriet i E-huset, Lund University, Lund 2017

Pour Justine.

Părinților mei.

Popular summary in English

The past two decades have witnessed a transformation of the status and role of computing: from a commodity supporting essential societal functions to a utility permeating *all* aspects of daily life. This transformation was accompanied by the emergence of so-called cloud computing – a service model that made computation infrastructure reliable, scalable and easily accessible. Increasingly, cloud computing displays the characteristics common to utility services, such as: necessity, reliability, usability, low utilization rates, scalability and (in some cases) service exclusivity.

In the cloud computing service model, users consume computation resources provided through the Internet, often without any awareness of the cloud service provider that owns and operates the supporting hardware infrastructure. This marks an important change compared to earlier models of computation, for example when such supporting hardware infrastructure was under the control of the user. Given the ever increasing importance of computing, the shift to cloud computing introduces several challenging issues, which include ensuring the integrity and confidentiality of the computation itself, along with integrity and confidentiality of ancillary resources such as network communication and the stored or produced data.

While the potential risks for data isolation and confidentiality in cloud infrastructure are somewhat known, they are obscured by the convenience of the service model and claimed trustworthiness of cloud service providers, backed by reputation and contractual agreements. Ongoing research on cloud infrastructure has the potential to strengthen the security guarantees of computation, data and communication for users of cloud computing. This thesis is part of such research efforts, focusing on assessing the trustworthiness of components of the cloud network infrastructure and cloud computing infrastructure and controlling access to data and network resources.

The seven papers included in this thesis present a collection of contributions addressing select aspects of the focus areas above. The contributions include mechanisms to verify or enforce security in cloud infrastructure. Such mechanisms have the potential to both help cloud service providers strengthen the security of their deployments, and empower users to obtain guarantees regarding security aspects of service level agreements. By leveraging functionality of components such as the Trusted Platform Module, we describe mechanisms to provide user guarantees regarding integrity of the computing environment and geographic location of plaintext data, as well as to allow users maintain control over the cryptographic keys for integrity and confidentiality protection of data stored in remote infrastructure. Next, by leveraging recent innovations for platform security such as Software Guard Extensions, we describe mechanisms to verify the integrity of the network infrastructure in the Software-Defined Networking model. Finally, we propose an innovative scheme for access control of resources in Software-Defined Networking deployments.

Populärvetenskaplig sammanfattning på svenska

De senaste två decennierna har förändrat databehandlingens status och roll: från en nyttighet som stödjer viktiga samhällsfunktioner till något som är en naturlig del av väldigt många funktioner i det dagliga livet. Denna omvandling åtföljdes av framväxten av så kallad molnlösningar – en servicemodell som gjort datorresurser tillförlitliga, skalbara och lätt tillgängliga. Molnlösningar visar i ökande utsträckning på de egenskaper som är viktiga för moderna IT-tjänster, såsom: tillförlitlighet, användbarhet, utnyttjandegrad, skalbarhet och (i vissa fall) tjänsteexklusivitet.

I molnmodellen använder flera användare databehandlingsresurser som tillhandahålls via Internet, ofta utan att vara medvetna om molntjänstleverantören som äger och driver själva hårdvaruinfrastrukturen. Detta markerar en viktig förändring jämfört med tidigare modeller för databehandling, till exempel modeller där sådan stödjande hårdvaruinfrastruktur var under direkt kontroll av användaren eller användarens organisation. Med tanke på den allt större betydelsen av databehandling medför övergången till molnlösningar flera problem – till exempel gällande integritet och konfidentialitet i själva databehandlingen, tillsammans med integritet och konfidentialitet av tillhörande resurser, såsom lagrad eller producerad data och nätverkskommunikation.

De potentiella riskerna för brister i fråga om dataisolering och konfidentialitet i molninfrastrukturen är tämligen välkända men har hamnat lite i skymundan på grund av servicemodellens bekvämlighet och inte minst på grund av att den av molntjänstleverantörer hävdade pålitligheten hos tjänsterna samt genom garantier i service avtal. Forskningsresultat kring säkerhet för molninfrastrukturer kan på sikt stärka säkerhetsgarantierna för databehandling, data och kommunikation för användare av molntjänster. Denna avhandling är en del av sådana forskningsinsatser, med särskild inriktning på: (i) bedömning av komponenternas tillförlitlighet i molnnätverket och molninfrastrukturen samt (ii) kontroll av tillgången till data och nätverksresurser.

De sju papper som ingår i denna avhandling är en samling av forskningsbidrag som adresserar valda aspekter av ovanstående säkerhetsproblem. Forskningsbidragen innehåller nya mekanismer för att verifiera samt upprätthålla säkerheten i en molninfrastruktur. Sådana mekanismer har potential att både hjälpa molntjänstleverantörer att stärka säkerheten för deras installationer och att hjälpa slutanvändare att få säkerhetsgarantier motsvarande den nivå som serviceavtalen anger. Genom att utnyttja funktionalitet hos komponenter såsom “Trusted Platform Module” beskriver vi mekanismer för att ge slutanvändargarantier avseende integriteten i databehandlingsmiljön och geografisk placering av klartextdata, samt att tillåta användare att behålla kontrollen över de kryptografiska nycklar som används för att skydda integriteten och konfidentialiteten av data lagrad i molninfrastruktur. På samma sätt beskriver vi mekanismer för att verifiera nätverksinfrastrukturens integritet i den mjukvarudefinierade nätverksmodellen genom att utnyttja nya plattformssäkerhetsteknologier såsom “Software Guard Extensions”. Vi föreslår dessutom ett innovativt system för åtkomstkontroll av resurser i mjukvarudefinierade nätverksinstallationer.

List of included publications

This thesis is based on the following publications, referred to by letters of the Latin alphabet:

A **Trusted Launch of Virtual Machine Instances in Public IaaS Environments**

Nicolae Paladi, Christian Gehrmann, Mudassar Aslam, Fredric Morenius
ICISC'12, Proc. 15th International Conference on Information Security and Cryptology, 2012

B **Providing User Security Guarantees in Public Infrastructure Clouds**

Nicolae Paladi, Christian Gehrmann, Antonis Michalas
IEEE Transactions on Cloud Computing, 2016

C **Domain-Based Storage Protection with Secure Access Control**

Nicolae Paladi, Antonis Michalas, Christian Gehrmann
SCC'14, Proc. 2014 International Workshop on Security in Cloud Computing, 2014

D **Trusted Geolocation-Aware Data Placement in Infrastructure Clouds**

Nicolae Paladi, Mudassar Aslam, Christian Gehrmann
TrustCom'14, Proc. 13th International Conference on Trust, Security and Privacy in Computing and Communications, 2014

E **Towards Secure Multi-Tenant Virtualized Networks**

Nicolae Paladi, Christian Gehrmann
TrustCom'15, Proc. 14th International Conference on Trust, Security and Privacy in Computing and Communications, 2015

F **TruSDN: Bootstrapping Trust in Cloud Network Infrastructure**

Nicolae Paladi, Christian Gehrmann
SecureComm'16, Proc. 12th International Conference on Security and Privacy in Communication Networks, 2016

G **SDN Access Control for the Masses**

Nicolae Paladi, Christian Gehrmann
submitted for review

Acknowledgements

This work is the product of a very interesting and interdisciplinary collaboration in a series of national and international projects. I would like to thank my adviser at SICS and Lund University, Christian Gehrmann, for guiding me throughout the past years. I have greatly enjoyed the journey and learned a lot on the way. Likewise, I would like to thank Ben Smeets, my adviser at Lund University, in particular for his interest in transferring the TPM-related work to industry and for supporting my interest in SGX.

Throughout the past several years, SICS has become my main residence, the place where I spent most of my days, evenings and very early mornings. I would like to thank the SICS administration for excellent support and for making SICS such a welcoming workplace. This project would not have been the same without my colleagues at the Security Lab. Here I would like to thank once again Christian Gehrmann for his leadership and thoughtful management of the Security Lab earlier on. I will carry with me the lessons of organization, work ethics and team management that I have learned while being part of the lab. Likewise, I would like to thank Oliver Schwarz for the intellectually challenging philosophical and technical discussions, and productive late-night brainstorming sessions. Thank you for reviewing my papers, finding new arguments to challenge my ideas and expanding my horizons; I wish you good luck in your new career. I have had a particularly dynamic collaboration with Antonis Michalas, both earlier as a colleague at the Security Lab and currently in his new role at Westminster University. I am thankful for all the lessons learned from working together and I look forward to continuing our collaboration.

I have received good advice and helpful guidance from many colleagues on the way. I would like to thank Lars Rasmusson putting me in contact with the Security Lab and helping out with the intricacies of software integrity measurement implementation in the Linux Kernel. Likewise, I would like to thank Rolf Blom, Ian Marsh, Fatemeh Rahimian and Amir H. Payberah for their friendly support, helpful advice and stimulating discussions. I have greatly enjoyed the challenging and enlightening discussions at the reading group at KTH and would like to thank Mads Dam, Roberto Guanciale, Christoph Baumann, Oliver Schwarz, Hamed Nemati and Andreas Lindner for this experience. I would also like to thank Fredric Morenius and Mudassar Aslam for the fruitful collaboration in the early days of the InfraCloud project as well as Felix Klaedtke for his guidance and collaboration in the 5G-ENSURE project.

Realizing demanding work projects would have not been possible without the solid moral support of my family and friends. Dear friends in Sweden and around the world, thank you for your support, interest and inspiration throughout the years. Thank you for sharing your energy in our common projects - whether organizing documentary screenings in Stockholm, finding our way through the towns and villages of Galicia or kayaking in Lappland, Wielikopolska and Brandenburg. I will thank you in person. Finally, I would like to express my deepest gratitude to my family for their kind and ceaseless support. *Ați fost mereu aproape de mine, în special atunci cînd am avut nevoie de susținerea și ajutorul vostru. Vă mulțumesc mult!*

Acronyms

This list contains the acronyms and abbreviations used throughout the thesis.

ABAC Attribute-Based Access Control

ACM Authenticated Code Module

AES Advanced Encryption Standard

AIK Attestation Identity Key

AMD Advanced Micro Devices

AMQP advanced message queuing protocol

API Application Programming Interface

APT Advanced Persistent Threat

ARP Address Resolution Protocol

ASE Asymmetric Searchable Encryption

BIOS Basic Input/Output System

CA Certificate Authority

CBA Capability Based Access

CBC Cipher Block Chaining

CH Compute Host

COTS Commercial off-the-shelf

CP Cloud Platform

CPU Central Processing Unit

CRTM Core Root of Trust for Measurement

CSP Cloud Service Provider

DBSP Domain-Based Storage Protection

DM Deployment Manifest

DRAM Dynamic Random Access Memory

DRTM Dynamic Root of Trust for Measurement

DTLS Datagram Transport Layer Security

EC Endorsement Credential

ECDH Elliptic-Curve Diffie Hellman

EK Endorsement Key

EPC Enclave Page Cache

EPCM Enclave Page Cache Map

EPID Enhanced Privacy ID

ETSI European Telecommunications Standards Institute

FIB Forwarding Information Base

GB Gigabyte

GPS Global Positioning System

GRE Generic Routing Encapsulation

GUI Graphical User Interface

HTTP Hypertext Transfer Protocol

HTTPS HTTP over Transport Layer Security (TLS)

I/O Input/Output

IaaS Infrastructure-as-a-Service

IEC International Electrotechnical Commission

IEEE Institute of Electrical and Electronics Engineers

IETF Internet Engineering Task Force

IP Internet Protocol

IPC Inter-Process Communication

ISO International Organization for Standardization

IT Information Technology

KVM Kernel Virtual Machine

LAN Local Area Network

LE Launch Enclave

LXC Linux Containers

MAC Message Authentication Code

MANO Management and Orchestration

MLE Measured Launch Environment

NACA North-bound Access Control API

NBI north-bound interface

NC Network Controller

NIB Network Information Base

NIC Network Interface Card

NIST US National Institute of Standards and Technology

NMA Network Management Application

NOS Network Operating System

OASIS Organization for the Advancement of Structured Information Standards

OEM original equipment manufacturer

OP Operator Policy

OS operating system

OVSDB Open vSwitch Database Management Protocol

PaaS Platform-as-a-Service

PBAC Policy-Based Access Control

PCIM Policy Core Information Model

PCR Platform Configuration Register

PDP Policy Decision Point

PII Personally Identifiable Information

PKI Public Key Infrastructure

PRF pseudo-random function

PRM Processor Reserved Memory

PSK Pre-Shared Key

QE Quoting Enclave

QEMU Quick Emulator

RAID redundant array of independent disks

RAM Random Access Memory

RBA Role-Based Authorization

REE Rich Execution Environment

REST REpresenational State Transfer

RM Reference Monitor

ROM Read-Only Memory

RoT Root of Trust

RSA Rivest-Shamir-Adleman (algorithm)

RT Request Tagger

RTM Root of Trust for Measurement

RTT Round-trip time

SaaS Software-as-a-Service

SC Secure Component

SDK Software Developing Kit

SDN Software-Defined Networking

SEV Secure Encrypted Virtualization

SGX Software Guard Extensions

SGX2 Second Generation of Software Guard Extensions

SHA Secure Hashing Algorithm

SIGMA Sign and Message Authentication (protocol)

SINIT Measured Launch Initialization

SLA Service Level Agreement

SQL Structured Query Language

SRK Storage Root Key

SRTM Static Root of Trust for Measurement

SSE Symmetric Searchable Encryption

TCB trusted computing base

TCG Trusted Computing Group

TCP Transport Communication Protocol

TEE Trusted Execution Environment

TLS Transport Layer Security

TPM Trusted Platform Module

TTP trusted third party

TXT Trusted eXecution Technology

URL Uniform Resource Locator

VA virtual appliance

VLAN Virtual Local Area Network

VM virtual machine

VNF Virtual Network Function

VT-x Intel Virtualization Technology

XACML eXtensible Access Control Markup Language

XML eXtensible Markup Language

Contents

Part I

Thesis

Chapter 1

Introduction

"SSL added and removed here :-)"

"Current Efforts - Google",
Author Unknown
US National Security Agency

Like every important technology trend, *cloud computing* has generated the widest possible spectrum of reactions - from enthusiastically embracing (sometimes "free") services based on the new paradigm to conservatively dismissing it as either something that has been seen before, or as a menace to data security and user control over data. Paradoxically, more than a decade since the early "Infrastructure-as-a-Service" offerings have been made publicly available, many of the predictions and reactions from across the spectrum have become true. In fact, the adherents of either of the extremes were both right and wrong.

Indeed, nowadays "X-as-a-Service" almost pervasively powers the core operations of both major corporations and ephemeral start-ups. Skeptics missed once-in-a-generation business opportunities that fueled a staggering growth of cloud-based services. However, cloud providers have become targets of choice for a clique of adversaries: intelligence services [1, 2], serious organized crime and skillful organized groups [3], motivated insiders [4] and endless armies of mechanical turk-like script kiddies [5].

Cloud providers are often *significantly* more capable (and motivated) to dedicate resources for securing their services compared to many organizations that are sometimes oblivious to the importance of protecting their data and networks. However, organizations that outsource their data and trust to cloud providers may have little control over the protection level implemented by the cloud provider. Moreover, such organizations often become victims of collateral damage - while not being targets themselves, they may have their data exposed as a consequence of attacks on the cloud infrastructure.

Amid this disconcerting reality, users are served a dull palette of options: avoid cloud services and rely on in-house computing infrastructure; apply a security policy for confidentiality [6] or integrity [7] to segregate data stored in-house and stored remotely and invest significant effort to ensure that data can be shared without breaking the policies [8]; or trust the security claims of cloud service providers, potentially also requiring that they are certified by a third party according the relevant audit or certification schemes [9].

1 Trust, but Verify

"Trust, but verify" goes an old adage. Indeed, relinquishing control over data and network security to cloud vendors without due diligence used to be common practice, especially in the early stages of cloud services. At first glance, arguments were compelling: drastic cost reduction for infrastructure maintenance and IT personnel on payroll; no high up-front fees when purchasing enterprise server and network equipment; flexible on-demand service scalability - all backed by a simple agreement and the market reputation of the cloud provider. Even if one were to follow the above adage, there was little support to do so in practice except relying on third-party certification. However, the certification bodies themselves took their time to develop specialized cloud security standards beyond the already available IT security audit schemes [9, 10]. Lately, major cloud service providers have adopted comprehensive cloud-specific security standards. For example, ISO/IEC 27017 *Cloud Security* clarifies the division of responsibilities for protecting data in the cloud environment between the cloud service provider and cloud users; it describes controls regarding sharing information security roles, management of customer assets in case of service termination, isolation of virtual computing and network environments, monitoring, etc. ISO/IEC 27018 *Cloud Privacy* contains controls for protection of Personally Identifiable Information (PII) in cloud environments, aimed towards customer control over PII, transparency with regard to data collection, PII transfer to third parties and data breach disclosure procedures. Other audit frameworks or certification schemes for auditing security measures could also be applicable to cloud services [9]. Beyond that however, users of cloud services - whether Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) or Infrastructure-as-a-Service (IaaS) - had no mechanisms to verify the implementation and application of security parameters of cloud deployments.

In fact, *post-factum* verification may be a legitimate strategy when a breach of trust can be mitigated through corrective action (for example through financial compensation) or simply accepted along with its consequences. However, this approach may not always be acceptable to either the users or the cloud service providers themselves.

2 Verify, then Trust

A better approach would be to establish a trust relationship immediately *prior* to transferring data or computation to a remote deployment. Considering that scalability and resource elasticity are among the strongest reasons for relying on cloud infrastructure and other similar forms of remote network and computation resources, any additional security mechanisms must have minimal impact on the performance and availability of such remote infrastructure. To this end, this thesis addresses the following challenges:

- *Providing user security guarantees in infrastructure clouds*: operating a restricted set of up-to-date hypervisors and encrypting data at rest within the deployment is a reasonable approach; however, this approach can be further improved by enabling deployment-time trust decisions about a given platform configuration and transferring control over data protection keys to the tenants. Better still if cloud tenants can obtain a proof that the chain of trust has been followed throughout the virtual infrastructure deployment process, and if tenants can limit the availability of data to a restricted set of authorized users or to a certain data center or geographical region.
- *Strengthening the security of Software-Defined Networking (SDN) deployments* - the SDN model has all but revolutionized large-scale network deployment and management; however, such changes have obsoleted many of the decades-old best practices of network security and introduced new security risks. These changes include a shift from embedded software on specialized hardware to applications executing on commodity hardware and operating systems, logical centralization of networks, as well as the wide adoption of virtual network functions. Newly introduced security risks can be addressed by improving the security of Software-Defined Networking infrastructure using approaches adapted from other fields, such as platform security and operating system security.

Establishing the trustworthiness of a remote cloud deployment is a complex procedure. Depending on the threat model, it may involve verifying a wide range of aspects, such as physical security, trustworthiness and access privileges of the support staff, system configuration and integrity of the software image on the remote platform. However, even assuming physical security and trustworthy support staff, effective appraisal of cloud deployment trustworthiness by tenants is difficult in practice, due to several reasons. First, the cloud model is built on an *asymmetrical* trust model (Chapter 2) and cloud service providers are not incentivized to disclose deployment internals to its tenants, as this may erode their competitive advantage or help adversaries find new attack vectors. Second, continuously assessing the trustworthiness of the entire deployment requires a detailed understanding of the trustworthiness of both the different software versions of cloud infrastructure components and of the consequences of their interaction and periodic upgrades. Even assuming that such level of access is granted, for tenants the cost of undertaking this effort voids the benefits of using cloud services. In some cases, cloud providers set up dedicated cloud deployments, certified according to a government

security assessment, authorization, and *continuous* monitoring program (for example FedRAMP [11]). However, access to such deployments is restricted to government agencies [12], as multi-tenancy is regarded as a security risk [13]. Other tenants have the option to use cloud offerings reserved for the general public, certified according to one of the programs which mostly conduct *periodic*, point-in-time assessment of a provider or a service. However, as pointed out in [9], the effectiveness of such programs is severely limited considering the rapid change of technology and products.

System configuration and software integrity can be verified remotely - both periodically and continuously - through remote attestation of the platform state. This the first focus area of the thesis. In Paper A we describe mechanisms that provide evidence of platform verification to tenants and end users of virtual appliances. We extend and describe this approach in Paper B by: (a) formally defining a set of attacks on the trusted virtual machine launch process and proving the security of the described scheme through a theoretical analysis; (b) describing a mechanism to protect the data stored beyond the perimeter of the executing host - this allows the use of cheap, commodity remote storage while maintaining the integrity and confidentiality of stored data. This approach further reduces the cost of using cloud computing without compromising security or burdening the users with overly complex procedures. In Paper C we describe the storage protection mechanisms extended with multi-tenancy support. Finally in Paper D we describe mechanisms to control access to plaintext data based on geographical location.

Security mechanisms for network infrastructure in cloud deployments are the second focus area of the thesis (Papers E, F, G). The swarms of physical server deployments required to power major Internet services have been dwarfed by a rapid growth in the number of virtual machine (VM) deployments. VMs are used to host web services, are leased entirely to tenants, or are further multiplexed using OS level virtualization. This has increased network complexity to a point where traditional network management evolved throughout the previous decades became inadequate. An unscalable approach became obsolete: rigid - yet brittle and intricate - network connectivity was configured by distributed algorithms with significant assistance from human operators who painstakingly wrote configuration policies. SDN addressed these new challenges by decoupling the forwarding and control layers of network infrastructure: increasingly well-performing commodity hardware forwards the packets, while the control layer operates on centralized network views to monitor and improve network management. However, this paradigm change in network management also implies that earlier best-practices became less relevant as new security risks were introduced. Such security risks to SDN must be identified and addressed to enable service providers to ensure the security of the virtual network infrastructure; this is a prerequisite for gaining user trust in - and adoption of - software applications that rely on network communication. This thesis aims to describe a suitable threat model and identify security threats to SDN (Paper E). It also addresses some of the threats to increase the trustworthiness of virtual network infrastructure, through integrity verification of the forwarding plane (Paper F) and access control for network management applications (Paper G).

The two focus areas of the thesis reflect the underlying research projects, namely *InfraCloud* and *5G-ENSURE*. InfraCloud was financed by Vinnova, the Swedish Innov-

ation Agency and aimed at strengthening the security expertise of public agencies in the area of cloud security; *5G-ENSURE* is financed by the European Union within the Horizon 2020 Framework Programme for Research and Innovation and aims to develop security mechanisms and technologies for the next generation of mobile communication infrastructure. Considering the ongoing convergence of the two fields, some of the results can be applied in both of them.

3 Thesis Outline

Following this introduction, Part I continues with a background chapter (Chapter 2) that describes core platform and network infrastructure security concepts to help readers better understand the papers in Part II. The background section contains an overview of the Infrastructure-as-a-Service cloud computing model, followed by an overview of the Software-Defined Networking model. It outlines the main challenges, recent developments and current security research efforts in both areas, and aims to place into a common context the papers included in the thesis. Chapter 3 presents the contributions to the thesis: first as an overview, followed by a description on a per-publication basis. It also contains an enumeration of the author's individual contributions and content updates to the original version.

> Throughout Sections 1 and 2 of Chapter 2, information boxes such as this briefly clarify how the introduced content relates to the papers included in the thesis. This is done to improve readability and help the reader to relate the background material to the included papers.

Part II contains the peer-reviewed publications (Papers A, B, C, D, E, F) and one manuscript under review (Paper G) included in this thesis.

Chapter 2

Background

> "There is no cloud just other people's computers."
>
> Postcard,
> Free Software Foundation Europe

Alice is the Chief Technology Officer in an organization that operates with large amounts of data as part of its business. The board of directors has been discussing cost reductions and someone mentioned migrating the enterprise systems to a public cloud infrastructure as a possible way to improve the balance sheet. Alice is well aware of the cloud computing concept and has even used a virtual machine from a public cloud service in her spare time. Beyond on-going hand-wave talk of cloud computing, Alice considers the most likely scenario would be to adopt the Infrastructure-as-a-Service (IaaS) model and deploy the plethora of enterprise systems on a cluster of virtual machines. Choosing a long-term infrastructure cloud service is a complex task. Beyond the maze of contractual and service level agreements, one issue persists - choosing a trustworthy provider to provide the best possible data security.

What is *trustworthiness* when it comes to a provider of computing, network and storage?

1 Trust, Attestation and Isolation

We adopt the definition from [14], according to which *trust is defined as confidence in the integrity of an entity for reliance on that entity to fulfill specific responsibilities.* Computational trustworthiness is described in assurance levels, based on specific measures that define the conditional and temporal requirements that must be fulfilled in order to rely on a relationship or transaction. As it is highly dynamic, trustworthiness may increase or decline over a certain dimension, the most common being *time* and *number of transactions*. For example, a service provider that has followed the Service Level Agreement (SLA) in the past *may* be increasingly trusted to maintain a high standard in the future (or vice-versa). On the other hand, consider a service provider that has at some point put in place certain strong security mechanisms (such as a strong hashing algorithm) but has not upgraded them over time: in this case the trustworthiness of the service provider *may* decline, since advances in cryptography are likely to render the respective hashing algorithm vulnerable to preimage-, collision- or other attacks.

A trust relation is not necessarily binary [14], as Alice may trust the service provider *to a certain extent* (for example for storing test or production data, but not the state-of-the-art research plans). Likewise, a trust relation does not need to be absolute [15] - Alice may trust the service provider A *more* than service provider B, but not trust either of them to reliably store business-critical data. On the other hand, trust is not necessarily symmetric - the fact that Alice trusts to a certain degree service providers A and B does not imply that the service providers trust Alice (as it is in the public cloud model). Finally, trust *may* be revocable: a data breach or audit report revealing the cloud service provider's poor security practices could precipitate Alice's complete loss of trust towards this service provider.

Prior to discussing several approaches to managing trust, we introduce informal definitions of users and tenants used throughout this thesis. A *user* is a human individual, computer, or an organization interacting with a service, a platform, or a virtual appliance without any control over its deployment and underlying resource allocation; by *end-users* we refer exclusively to human users. A *tenant*, according to the definition from [16] adapted to the context of this thesis, is the entity (which may include the user) responsible for the configuration, management and operation (including availability and security) of a cloud resource (service, platform, or computing, storage and communication infrastructure).

Both users and tenants have in practice few - if any at all - means to establish a *direct* trust relationship with the service provider. The reasons for this are both the stringent security routines claimed by the providers of public cloud services and (often) the complexity of computing deployments, incomprehensible to the vast majority of users and tenants.

1.1 Trust Delegation

Considering the situation described above, tenants can recur to *trust delegation* in one of the forms discussed below to establish a trust relationship with the service provider.

Transitive Trust

Tenant Alice may decide to trust the service provider because she knows that tenant Bob trusts the same service provider. Bob need not be aware of the implications - for Alice - of his trust relationship with the service provider. This is a form of *transitive trust*, where the trust relation between one tenant and the service provider is conditioned on the trust relation of a different tenant (or multiple tenants) with the same service provider (see Figure 2.1). However, transitive trust can be misleading. First, Alice

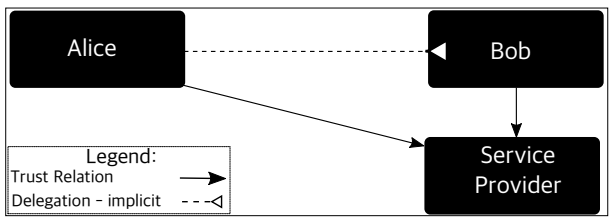


Figure 2.1: Transitive delegation of trust: Bob trusts the service provider, hence Alice chooses to trust the service provider as well.

cannot know whether the trust context or the parameters of the trust relation between Bob and the service provider are aligned with her own trust context and parameters. Second, unless a mechanism is available to notify Alice about the evolution of the trust relation between Bob and the service provider, she might be left unaware when Bob decides to discontinue the trust relation.

Reputational Trust

Another option for Alice is to rely on the (publicly or privately) available ratings, reviews and experiences documented by her peers (Bob, Carol and others) in order to establish a trust relation with the service provider. Such *reputational* trust bears similarities to transitive trust below, with one essential difference - Alice's peers are likely aware about their participation in this reputational trust scheme, making such scheme explicit and giving the peers certain privileges to influence it.

Transitive trust and reputational trust have been the de-facto approaches to trust establishment employed since the early days of cloud computing. However, this is currently transitioning to new forms of trust establishment, described below.

Direct delegation

The revelations of the whistle-blower Edward Snowden have significantly increased awareness about the magnitude of data collection and of activities aimed to subvert the security of major communication and cloud service providers [1, 2]. Furthermore, such revelations have contributed to adjusting - closer to reality - the threat models considered in security research. The flurry of reported vulnerabilities in cloud infrastructure [17–21] further de-legitimized reliance on transitive and reputational trust approaches for trust establishment and accelerated the development of dedicated security standards, such as ISO/IEC 27017 *Cloud Security* and ISO/IEC 27018 *Cloud Privacy* (introduced in Section 1), in addition to the earlier established ones [9].

This evolution allows tenants to complement transitive and reputational trust with *direct delegation* of trust (see Figure 2.2). In this case Alice - unable herself to assess the trustworthiness of the service provider - delegates the decision to Bob, which is more apt to make such a decision. Similar to transitive trust above, Bob might not be aware that Alice has delegated her trust decision to him. However, in this case an explicit certification mechanism is available - Bob verifies that the service provider fulfills a certain well-known set of conditions and provides a *time-limited* certificate to endorse the service provider. Note that this does not necessarily imply that Bob trusts the service provider for his own operational purposes, since Bob may have his own, stricter or entirely different requirements. In other cases, Bob may have an ex-

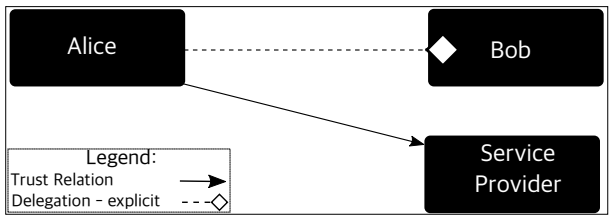


Figure 2.2: Direct delegation of trust: Bob can evaluate the trustworthiness of the service provider and issues time-limited certificates; Alice can rely on the certificates and thus explicitly delegate her trust decision to Bob.

plicit agreement with Alice and verify that the service provider fulfills certain specific conditions required by Alice. Examples of such additional conditions include explicit requirements regarding the software stack of the compute host infrastructure where data is processed, or support for hardware-enabled execution isolation features. While such user-specific evaluations can be performed on a much finer time-scale and provide additional assurance to the user, they incur significant costs and are rarely supported by cloud service providers. For example, the *Federal Risk and Authorization Management Program* (FedRamp) aims to achieve "near real-time monitoring" [22] of cloud service providers. However, access to cloud resources monitored under FedRamp is subject to restrictions (for example accessible only to government organizations) and deployed on dedicated infrastructure [23]. Isolation - discussed below - is essential to ensuring that the execution of a process cannot be affected by other, potentially malicious processes.

1.2 Attestation

Simply providing isolation of the execution environment is (in most cases) insufficient to establish a trust relationship with the target platform, as remote users cannot know whether they are communicating with the intended software or a maliciously modified instance. Therefore, *attestation* (explained below) is of central importance for trust establishment in a remote system. Attestation of a target can be performed either by a dedicated *appraiser*, or directly by the remote user. Users delegate trust (either directly or transitively) to an appraiser, which is an entity - generally a computer or a network - making a decision about one or more other targets. A *target* is a party (for example a computer system) about which an appraiser needs to make such a decision [24]. Alternatively, a remote user can itself assume the appraiser role and attest the target using components that are part of it or under its control.

Attestation is the activity of making a claim to an appraiser about the properties of a target by supplying evidence which supports that claim. An *attester* is a party performing this activity. An appraiser's decision-making process based on attested information is *appraisal* [24].

The goal of an appraisal is to take a decision regarding the expected behavior of the target prior to establishing a trust relationship. This is done by collecting enough information about the target - such as hardware, software, and configuration data - in order to establish that the target is in an acceptable state or will not transfer to an unacceptable state after a trust relationship is established. Given that the collected information can be very thorough, the target may restrict access to such information about itself in order to avoid disclosing certain business-specific configuration or protect the privacy of the humans owning or operating it. Furthermore, the target may provide distinct information to different appraisers depending on the existing trust relationships between them. This reflects a potential contradiction between the interests of the human organizations behind the appraiser and target: the owner of the appraiser requires as much information as possible to establish a trust relationship, while the owner of the target is interested in revealing the least possible amount of information in order to establish a trust relationship. The possibility of establishing a trust relationship between a target and an appraiser ultimately depends on the trust relationship between the humans or human organizations owning or operating the target and appraiser. Communication between an appraiser and a target is conducted in the form of an *attestation protocol*, as defined in [24]:

Definition 1.1. *An attestation protocol is a cryptographic protocol involving a target, an attester, an appraiser, and possibly other principals serving as trust proxies. The purpose of an attestation protocol is to supply evidence that will be considered authoritative by the appraiser, while respecting the privacy goals of the target (or its owner).*

Note that attestation is different from target measurement, which has a narrower scope. Multiple measurements of a target can be reported in an attestation protocol to serve as the basis for appraisal. Coker et al. [24] define target measurement as collecting evidence about the target through direct and local observation of it.

For the purposes of establishing a trust relationship, the collection of evidence about the target cannot be fully deferred to the target itself for obvious reasons - a malicious target may fabricate the expected measurements and report them. Instead, the appraiser is assisted by a *Root of Trust (RoT)* placed on the target. A RoT is an immutable computation engine with known behavior, which a certificate asserts to be present on a particular platform. The Trusted Computing Group (TCG) has defined several types of roots of trust [25]:

- A *root of trust for measurement* is a computation engine (or some functionality provided by hardware) that can reliably prepare certain measurements on the software state of a device.
- A *root of trust for reporting* is a computation engine (or some functionality provided by hardware) that can reliably attest to the result of a measurement.
- A *root of trust for storage* is a computation engine (or some functionality provided by hardware) that ensures that certain data such as cryptographic keys will be stored in a way that will always preserve their secrecy.

Coker et al. outline five central principles for attestation architectures [24]. To date, no attestation mechanisms correspond to all of the principles of such an "ideal" attestation architecture. Instead, the extent of support for the different principles vary across the available attestation mechanisms.

1. *Fresh information*: Assertions about the target should reflect the running system rather than disk images of the target.
2. *Comprehensive information*: Attestation mechanisms should be capable of delivering comprehensive information about the target; its internal state should be accessible to local measurement tools.
3. *Constraint disclosure*: A target should be able to enforce policies governing which measurements are sent to each appraiser. Hence, an attestation architecture must allow the appraiser to be identified to the target. Policies may distinguish the kinds of information to be delivered to different appraisers. The policy may be dynamic and rely on current run-time information for individual disclosure decisions. For instance, a target may require that the appraiser provides an attestation of its own state, before the target discloses its state.
4. *Semantic explicitness*: The semantic content of attestations should be explicitly presented in logical form. The identity of the target should be determined by these semantics, so an appraiser can collect attestations about it. The appraiser should be able to infer consequences from several attestations, for example when different measurements of the target jointly imply a prediction about its behavior. Hence, attestations should have uniform semantics, and be composable using valid logical inferences.

5. *Trustworthy mechanism:* Appraisers should receive evidence of the trustworthiness of the attestation mechanisms on which they rely. In particular, the attestation architecture in use should be identified to both appraiser and target.

Attestation mechanisms may follow the principles described above selectively, or to varying degrees depending on their implementation and the use cases they aim to address. For example, some targets may not require constrained disclosure of measurements, while "freshness" of the information may vary depending on the trade-off between the appraiser's requirements and the performance overhead induced on the target by the attestation. Differences in supporting the principles can be caused by the inherent limitations of the attestation architectures (static variation) or by providing different degrees of evidence depending on certain contextual factors, such as whether the appraiser is known to the target (dynamic variation).

1.3 Isolation

Isolation is an essential concept in trust establishment, as well as for platform and network security. In platform security, the goal is to protect certain assets from malicious software executing on the platform. In the context of cloud computing, this aspect is crucial for separation of collocated assets belonging to distinct tenants. In network security, the goal is to prevent unauthorized parties from intercepting traffic from adjacent network domains. In the context of cloud computing, this aspect is essential for preventing distinct tenants collocated in the same infrastructure from intercepting the traffic from each other's or third party network domains. This trivial taxonomy blurs in the case of virtual networks or infrastructure in the Software-Defined Networking (SDN) model: network components on the forwarding plane (for example virtual switches) are deployed on the same platform as the virtual appliances (including virtual machines, containers [26] and unikernels [27, 28]) that belong to different tenants. In this case network and platform isolation become intertwined - the adversary can compromise a virtual switch to break network isolation; likewise, an adversary having access to a network domain can craft malicious packets in order to compromise the virtual switch and even break the isolation between guest VMs in case of a hypervisor breakout. Aspects of isolation in SDN are discussed in Section 3.3.

Trusted Computing Base Isolation of code and data as well as support for security policies in a computer system are enabled by a set of hardware and software components comprising the trusted computing base (TCB). According to the definition in [29], the TCB includes hardware, firmware, and software critical to system security and must be designed and implemented such that system elements excluded from it need not be trusted to maintain protection. A TCB should be as simple as possible consistent with the functions it has to perform, since as the size and complexity of the TCB increases, it is more likely to contain exploitable vulnerabilities. The TCB often operates alongside a much larger collection of hardware and software components which are not critical to the security of the system. When such components misbehave, the TCB can make

sure that the system is "fail-secure", for example by restricting functionality or access to data until the system is in a correct state [30].

Trusted Boot and Secure Boot Gasser et al. introduced two approaches to platform booting which are important in the process of TCB isolation, namely the *trusted boot* and *secure boot* [31]. The difference between trusted boot and *secure boot* is illustrated in Figure 2.3. In a trusted boot, the chain of trust is initiated by a hardware-

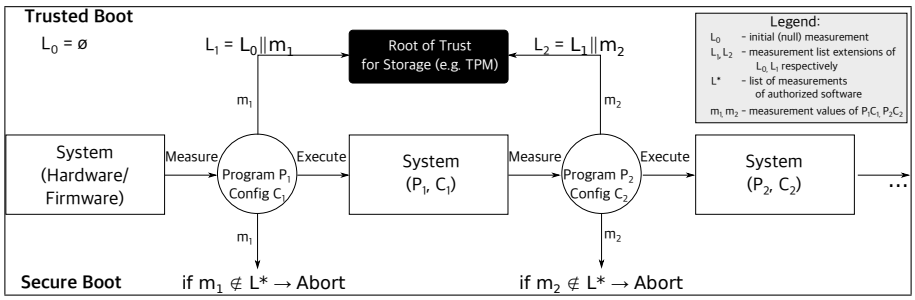


Figure 2.3: Conceptual illustration of *secure boot* and *trusted boot*, based on [32].

based RoT that measures the initial Basic Input/Output System (BIOS) code, which subsequently measures and executes the bootloader, after which the bootloader measures and executes the operating system. Note that one cannot conclude that a system that has booted following the trusted boot procedure is necessarily trustworthy, merely that it must be trusted if the platform itself is to be trusted [32]. Furthermore, a failed appraisal does not prevent execution of the operating system (OS) on the platform; however, it can prevent access to certain code and data, only made available when the platform is in a trustworthy state.

In a secure boot, each system component on the platform - starting with the boot Read-Only Memory (ROM) - measures the code to be loaded and compares it to a list of measurements for authorized software. Software is typically authorized by a signature from a trusted authority; however, this requires the authority's public key to be available to the platform firmware. The boot process is halted in case an attempt to load unauthorized code is detected [31,33]. By completing the boot process successfully, secure boot provides assurance that the platform is in an approved state [32].

Trusted Execution Environments An isolated execution environment can be created in several ways. One approach to constructing an isolated execution environment is by validating the TCB using secure boot, as the TCB is by definition isolated from the rest of the system [34]. However, this approach is progressively less suitable beyond a very compact TCB, such as a hypervisor that can be formally verified.

A different approach is required when confidential information - such as cryptographic keys - is persistently maintained on the platform. To address this, some platform manufacturers have introduced support for hardware-based Trusted Execution Environments

(TEEs). TEEs often include storage for a (statistically) unique device key and an execution environment in which small pieces of code can be executed is isolation from the rest of the system [34]. Combined with the secure boot or trusted boot procedures, trust roots and identities described above, TEEs can become a minimal TCB for platform software. The TCB can in turn be leveraged by the booted OS, as well as by software installed on the device or external appraisers that aim to assess the platform's trustworthiness. A TEE is a secure, integrity-protected processing environment, with processing, memory and storage capabilities, isolated from an untrusted, Rich Execution Environment that comprises the operating system and installed applications [35].

A sequence of three implementations of hardware-assisted Trusted Execution Environments is presented next. This introduction aims to facilitate the understanding of their application in the papers included in this thesis. While the technologies described below currently coexist, the presented sequence can be seen as steps on the evolutionary path of hardware-assisted isolated execution environments.

1.4 Trusted Platform Module

Trusted Platform Modules (TPMs) are hardware components providing secure non-volatile storage, cryptographic key generation and use, sealed storage and (remote) attestation, according to the specifications defined by the TCG [36]. TPMs assume platform integrity by identifying and reporting the platform state, which comprises the hardware and software components on the platform [37]. In this context, *trust* is based on the conjecture that a certain behaviour can be expected based on the reported platform state. Being a discrete component on the platform motherboard, the state of the TPM is distinct from the state of the platform. The TPM interacts with the platform through an interface defined in the TPM specifications [36, 38]. Furthermore, a TPM can prove its association between a cryptographically verifiable identity and the host platform though *platform binding*. The first widely deployed TPM - version 1.1b - was released in 2003. To correct incompatibilities on the hardware level, vulnerabilities to dictionary attacks and many other issues, TPM specification version 1.2 was developed in the following years. The TPM 2.0 specification was released in 2014 [39]. Along with hardware TPMs produced by multiple vendors, there are also software TPM implementations for both TPM 1.2 [40] and TPM 2.0 [41]. A high-level overview of select TPM 1.2 features and changes in TPM 2.0 follows below. For a complete overview, see the specifications [36, 42] and other relevant literature [39]. Other secure co-processors, such as the IBM 4758 co-processor [43] offer similar functionality.

TPM 1.2

Storage According to the specification [36], a TPM 1.2 carries 24 Platform Configuration Registers (PCRs). PCRs are integrity-protected registers used to store measurements reflecting the state of the platform or selected files. Each PCR can hold one digest value. A PCR value can be modified either by extending it or by resetting the

PCR to an initial value. The PCRs can be reset using the TPM functionality (for resettable PCRs) or through power cycling. Extending the PCR allows to store multiple digest values as one cumulative hash. In the process of extending a value into a PCR, the incoming digest is appended to the existing PCR value and fed into a hash function, as follows: $\mathsf{PCR}_{\mathsf{new}} = \mathsf{H}(\mathsf{PCR}_{\mathsf{old}}||\mathsf{digest})$ where $\mathsf{PCR}_{\mathsf{new}}$ is the new digest value stored in the PCR, $\mathsf{PCR}_{\mathsf{old}}$ is the digest value previously in the PCR, $\mathsf{H}()$ is the hash function associated with the PCR and digest is the measurement extended into the PCR. This process is illustrated in Figure 2.4. PCRs are of central importance to the functionality enabled by TPMs, as they are used both in the *attestation* process for platform appraisal, as well as for *sealing* data to a given platform state.

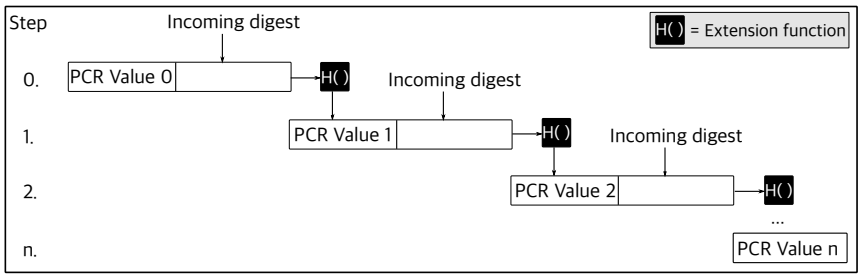


Figure 2.4: PCR extension mechanism in TPMs. *Extension function* is an implementation-specific hash function.

Key types Three non-migrateable types of keys are essential for implementing the *Root of Trust for Reporting* and *Root of Trust for Storage* abstractions in TPM 1.2.

1. Endorsement Key (EK) is an RSA signing key permanently embedded in the TPM at manufacturing time. it uniquely identifies and validates a TPM (and transitively the host platform) and is used in the process of issuing attestation identity credentials to establish platform ownership.
2. Attestation Identity Key (AIK) is a 2048-bit RSA key, alias of the EK, used to sign quotes of values contained in TPM PCRs multiple AIKs can be generated for each TPM.
3. Storage Root Key (SRK) is an RSA storage key generated within a TPM for every new owner; the SRK serves as a root key for its hierarchy that can contain both migrateable and non-migrateable keys (Figure 2.5).

Several other types of keys - migratable or non-migratable - are used to support the data protection functionality offered by TPM modules:

- Storage key is a 2048-bit RSA keys used for encrypting and decrypting other keys or sealed data with their security attributes external to the TPM.

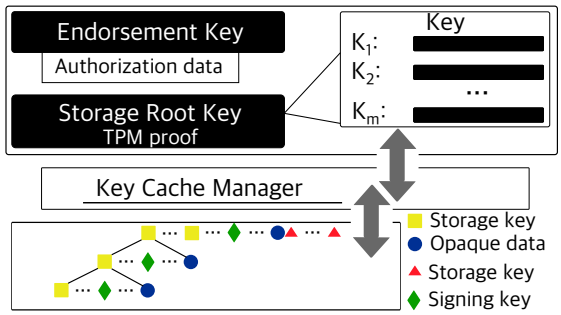


Figure 2.5: Root of Trust for Storage Architecture, from [36].

- Signing key is an RSA key part of the SRK hierarchy, used for signing data.
- Binding key is an RSA key used to decrypt - using the `TPM_Unbind` operation - generic data sets externally encrypted and associated with the identification and authentication data of the TPM.
- Legacy keys are a combination of binding and sealing keys usable in a limited set of commands discouraged from usage.

At the time of provisioning of the EK to the TPM, a TPM entity (normally the manufacturer) provides an Endorsement Credential (EC) - a certificate of the EK that binds the public part of the EK to a TPM description [44]. Credential binding is done through a signature computed with the signing key of the manufacturer and vouches that a TPM is genuine [45]. A Privacy Certificate Authority (CA) uses the EC to verify that a TPM is genuine prior to creating an *identity credential* by binding the public part of the AIK to the identity label and generic information about the platform [46]. The identity credential can thereafter be used to attest the authenticity of a platform configuration without unambiguously identifying the TPM.

Remote attestation TPMs can be used to establish the trustworthyness of the host platform by obtaining a signed quote of the platform state through *remote attestation*. During the attestation process, the TPM produces a quote containing the values of selected PCRs. The quote is next signed using the locally generated AIK (certified by EKs) The appraiser uses the quote received from the target host to decide on the trustworthyness of the host based on information about the software that has been loaded and measured.

Data protection The TPM provides several operations for data protection. *Binding* encrypts data using migratable or non-migratable asymmetric storage keys; in the latter case the encrypted data is not associated with a particular platform and the ciphertext can be decrypted on other platforms with the appropiate private key. *Sealing* is an extension of binding, where only non-migratable storage keys can be used to encrypt

data (hence, data is always bound to a specific platform). Sealing also allows to specify one or more PCRs to include a subset of the platform state in the ciphertext. The resulting ciphertext can only be decrypted ("unsealed") if the platform is in the same state as reflected by the values of the PCRs included at sealing time. Such functionality allows to prevent access to the sealed data if the host is compromised or is running an unexpected software stack; likewise, it allows to persist data in encrypted form until the execution enters a trusted state (for example, as implemented in Intel Trusted eXecution Technology [47] described below).

TPMs provide *signing* functionality to protect the integrity and determine the authenticity of data; TPM owners can use AIKs to sign audit data, quoted data, or tick-stamped binary objects. Finally, *sealed signing* includes - as part of the computation of the signed message digest - the values of a set of PCRs mandated by the user. This enables the appraiser to inspect the PCR values supplied in the signed message and obtain information about the platform configuration at the time when the signature was generated.

TPM 1.2 functionality was used in the mechanisms and prototype implementations described in Papers A, B, C, D included in this thesis.

Changes in TPM 2.0 The latest TPM specification at the time of writing - the TPM 2.0 library specification - has been developed to address two major concerns in the TPM 1.2 main specification: inadequate cryptographic algorithms in TPM 1.2 [36] and lack of universally accepted reference implementations. The first issue has been addressed by introducing *algorithm agility*, i.e., changing the algorithms as needed, without revisiting the specification. The second issue has been addressed by making the specification the same as the reference implementation [39, 41]. Introduction of algorithm agility has led to several additional improvements [39]:

- *Enhanced authorization* unifies the approach to authorizing TPM entities. Along with additional management functions, this enables authorization policies that allow for multi-factor and multi-user authentication.
- *Quick key loading* using symmetric encryption, rather than asymmetric encryption as previously done.
- *Non-brittle PCRs*, which were introduced to address management problems when locking keys to device states on platforms that must undergo state changes.
- *Flexible management* allows to separate the different types of authorization, to help management of TPM resources.
- *Resource identification* was been modified to use cryptographically secure names for all TPM resources.

Multiple key hierarchies are another addition to TPM 2.0. A *hierarchy* is a collection of entities (hierarchy handles, primary objects at the root of a tree, keys in the tree) that are related and managed as a group [39]. While TPM 1.2 has one hierarchy -

represented by the owner authorization and SRK - TPM 2.0 features three persistent hierarchies, namely *platform*, *storage* and *endorsement* hierarchies. Each hierarchy has an authorization value and a policy, an enable flag, a persistent seed for key and data object derivation and potentially a primary key from which descendants can be created.

According to the TPM 2.0 specification, the platform hierarchy is intended to be controlled by the platform manufacturer, represented by the early boot code shipped with the platform.

The storage hierarchy (similar to TPM 1.2) is intended to be controlled by the platform owner.

The endorsement hierarchy is used for privacy-sensitive operations and to certify the authenticity of the TPM. TPM manufacturers generate primary keys at the root of the endorsement hierarchy using a *seed* and a *template*. The seed is generated when the TPM is first powered on and thereafter resides in the TPM; the template describes the key algorithm and size, and optionally supplies additional entropy. The use of seeds and templates enables support for flexible key algorithms and key sizes without consuming non-volatile memory (compared to TPM 1.2 which directly generated one 2048-bit RSA endorsement key). The TPM manufacturer uses the public part of the generated primary key to create a certificate asserting that the public key belongs to a genuine vendor TPM. For privacy-sensitive operations, primary keys in the endorsement hierarchy are used to derive descendant *encryption* keys through a credential activation protocol with a privacy CA.

Finally, along with three persistent hierarchies, TPM 2.0 implements an *ephemeral* NULL hierarchy. The seed of the NULL hierarchy changes on every reboot, making this hierarchy suitable for the implementation of *ephemeral* key hierarchies (including primary keys and storage keys) and sessions. Ephemeral key hierarchies are cryptographically erased upon reboot - while keys may be present on disk, they cannot be loaded into the TPM. Ephemeral cryptographic objects can be created using trivial (zero-length password) authorization, in the cases when the TPM is used as a cryptographic co-processor, since the NULL hierarchy is always enabled and has an empty (unsatisfiable) policy.

The TPM is often used in trusted boot implementations to store measurement values, computed using another mechanism defined by the TCG, namely the Root of Trust for Measurement (RTM). Two common types of RTM implementations defined by the TCG are the Static Root of Trust for Measurement (SRTM) and the Dynamic Root of Trust for Measurement (DRTM). Trusted boot implementation typically uses an SRTM as follows: at system boot time the SRTM measures itself as well as other parts of the BIOS and the master boot record and stores the measurements in the TPM PCRs. A *Core Root of Trust for Measurement (CRTM)* is the component performing the self-measurement. Appraisers obtain - through an attestation protocol - a copy of the PCRs signed by the TPM in order evaluate the boot measurements and establish the trustworthiness of the platform. An attacker can exploit vulnerabilities in the implementation of the CRTM to modify the CRTM without the self-measurement detecting the change. In turn, this allows to corrupt without detection all subsequent elements in the chain

- thus fundamentally breaking the chain of trust [48]. Privileged software can use a DRTM to instatiate a trusted environment at a later point in time, even if the system booted in an untrusted state [49], as discussed next.

Intel TXT

Intel Trusted eXecution Technology is an implementation of the TCG Dynamic Root of Trust for Measurement [47]. Its primary purpose is to *detect* the potential presence of certain types of attacks, *notify* system owners about the detected attacks and prevent the creation of an Measured Launch Environment in the event of a compromise [39]. This is done by combining the SRTM and DRTM capabilities, along with additional support in software and in the *instruction set architecture*. At power-on, SRTM is used to establish and extend a chain of trust from the Intel processor (and chipset) to and including the BIOS. Once booted, the operating system or an application executing on the operating system can initiate a measured launch sequence by invoking the `GETSEC(SENTER)` instruction, which triggers the loading of the Measured Launch Initialization (SINIT) Authenticated Code Module (ACM). The SINIT ACM, which verifies the `TXT.ERRORCODE` register to ensure no security issues have occurred, triggers relevant policy checks and performs the measured launch, bringing the operating system into *secure mode*. The MLE is terminated by the operating system either explicitly when exiting the MLE (with the possibility to re-enter following the procedure described above) or implicitly at platform power-off or restart.

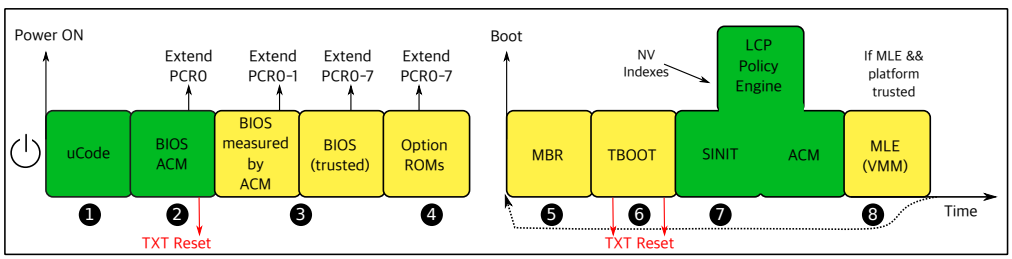


Figure 2.6: Intel TXT Boot timeline, based on [39].

Figure 2.6 illustrates the Intel TXT boot sequence, which uses the SRTM in order to detect BIOS and reset attacks [39]. First, the microcode verifies the BIOS ACM and starts the ACM in the Central Processing Unit (CPU) internal memory ❶. The BIOS ACM measures certain portions of the BIOS - specified in the firmware interface table configured by the BIOS original equipment manufacturer (OEM) - necessary for system integrity guarantees and extends the computed measurements into PCR `0` ❷. In case of an error, the *Startup* ACM sets an error code in the `TXT.ERRORCODE` register and interrupts the verified launch and resets the CPU (marked by 'TXT reset' in Figure 2.6). Next, BIOS extends *all* remaining BIOS blocks to PCR `0` ❸. Any other code in the BIOS trust boundary is measured to PCRs `1-7` ❹, for example option ROMs into PCR `3` and option ROM configuration into PCR `4`. Finally, the system boots the

operating system loaded on the system ❺ - at this point the operating system is ready to perform DRTM booting.

DRTM can be started by either the operating system or an application running in the operating system by triggering ❻ the `GETSEC(SENTER)` instruction. This invokes a microcode flow that verifies, loads and executes the SINIT ACM. In turn, the SINIT ACM verifies that no errors have been reported to the `TXT.ERRORCODE` register and measures the trusted operating system code and invokes ❼ a *Launch Control Policy* engine to perform policy checks and verify the measured OS code and PCR values against a list of known good values. Measurements performed in this step are extended into PCRs `17-18`. If any of the checks fail at this stage, the platform is reset (marked by 'TXT reset' in Figure 2.6). Upon a successful verification the operating system enters the *trusted mode* ❽, referred to as a Measured Launch Environment. In the MLE the operating system acquires ❾ TPM Locality 2 access, i.e. ability to extend PCRs `18-22` with measurements of other operating system components and configurations, which can also be used by local applications. The DRTM is terminated by either the initiating process itself or platform restart.

Intel Software Guard Extensions

Software developers can use Intel Software Guard Extensions (SGX) enclaves (introduced in [50–53]) to create TEEs during operating system execution. Such enclaves rely for their security on a trusted computing base of code and data loaded at initialization creation time, processor firmware and processor hardware. Program execution within an enclave is transparent to both the underlying operating system and other enclaves. Multiple mutually distrusting enclaves can operate on the platform.

Enclaves operate in a dedicated memory area called the Enclave Page Cache (EPC) which in turn is a subset of Processor Reserved Memory (PRM). The PRM is a range of Dynamic Random Access Memory (DRAM) reserved by BIOS which cannot be accessed by system software or peripherals [51, 54]. Furthermore, Enclave Page Cache Map (EPCM) is a data structure that contains a mapping between the enclave identities and the EPC pages that belong to them. This mapping is used by the CPU to verify enclave access to memory pages and prevent unauthorized access attempts. The CPU firmware and hardware are the Root of Trust of an enclave. It prevents access to the memory segment of the enclave by either the platform operating system, other enclaves, or other external agents.

The life cycle of an SGX enclave starts with a *creation* stage, when the `ECREATE` instruction invoked by the system software allocates a memory page for the *SGX Enclave control structure* and populates it with data about the memory size and layout of the enclave, made available by the system software. Once the enclave is created, system software uses the `EADD` instruction to load code and data into the enclave using the `EEXTEND` instruction to update the measurement of the enclave. Finally, the system software obtains an intialization token (`EINITTOKEN`) from a dedicated Launch Enclave (LE) and initializes the enclave (using the `EINIT` instruction). Once the enclave is ini-

tialized, the enclave application software can execute the code in the enclave. Note that the LE must sign the produced `EINITTOKEN` using one of they keys supported by Intel SGX (the Intel signing key is the default option [54]). The CPU saves the measurement throughout the lifetime of the enclave to later assert the integrity of the enclave contents. In the first implementation of SGX, no additional changes could be done to the memory of the initialized enclave [50]. The next generation of SGX (denoted SGX2), introduces support for additional instructions for manipulation of enclave memory [54] (described below).

Remote attestation allows an enclave to provide integrity guarantees of its contents [50] (see Figure 2.7). For this, the platform produces an attestation report with information about the identity of the enclave and details of its internal state (such as the mode of the software environment, associated data, and a cryptographic binding to the platform trusted computing base producing the assertion). For *intra-platform attestation* (i.e. between enclaves on the same platform), the reporting enclave (*reporter*) invokes the `EREPORT` instruction to create a `REPORT` structure with the assertion and calculate a MAC, using a *report key*, known only to the target enclave (*target*) and the CPU. The structure contains a `user data` field, where the reporter can store a hash of the auxiliary data provided. The target recomputes the MAC with its report key to verify the authenticity of the structure and compares the hash in the `user data` with the hash of the auxiliary data to verify its integrity. Enclaves then use the auxiliary data to establish a secure communication channel.

For *inter-platform attestation* (see Figure 2.7) the appraiser[1] sends a challenge ❶ to the target enclave application, which complements the challenge with the identity of a Quoting Enclave (QE) and sends it ❷ to the target enclave. The target enclave computes an integrity `REPORT` which contains its identity and internal state and sends it ❸ to the Quoting Enclave ❹; The QE verifies the `REPORT`, computes an attestation `QUOTE` and signs it with a platform-specific key using the Enhanced Privacy ID (EPID) [55], and returns it ❺ (using the enclave application) to the appraiser ❻. Finally, the appraiser checks ❼ the authenticity of the signature and the report itself [50] and assesses the reported state of the enclave.

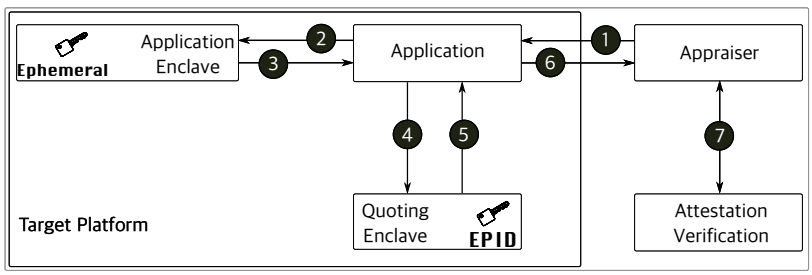


Figure 2.7: Intel SGX external remote attestation overview.

Despite the contribution of SGX towards protection of data on remote hosts, it must be noted that the first published specification contains vulnerabilities to practical at-

[1] In this example the appraiser itself performs the attestation.

tacks, described in [56–59]. Some of the vulnerabilities have been addressed through improvements implemented in SGX2. One example are the *Iago attacks*, where a malicious operating system subverts a protected application by exploiting the application's reliance on correct results of system calls [60]. Baumann et al. in ported a library operating system into an SGX enclave to handle page faults inside the enclave [61] and thus prevent Iago attacks. This work has contributed to the subsequent implementation of secure exception handling in SGX2 [53] and dynamic memory allocation in enclaves [52].

Changes is SGX2 Novel instructions introduced in SGX2 allow adding memory resources and new threads to an enclave after initialization [54]:

- `EUG` adds a page to an initialized enclave.
- `EMODPR` restricts the access rights associated with an EPC page in an initialized enclave.
- `EMODT` changes the type of an existing EPC page.
- `EACCEPT` accepts changes made by system software to an EPC page in the running enclave.
- `ECCEPTCOPY` initializes a dynamically allocated EPC page from another page in the EPC.
- `EMODPE` extends the access rights of an existing EPC page.

See [54] for a complete description of the instructions supported by SGX2.

SGX - similar to other trusted computing solutions - is vulnerable to *cuckoo attacks* [62], which is made possible because Intel SGX is not resistant to hardware attacks [51]. Thus, the adversary can acquire an SGX-enabled platform and launch a long-term physical attack to extract the key necessary to sign the intialization token (`EINITTOKEN`) and impersonate other SGX enclaves. For platforms running in a cloud environment, Schuster et al. addresses this by introducing an additional component - namely a *Cloud QE*, created by the cloud provider for each provisioned SGX-enabled platform; the *Cloud QE* complements quotes by the QE with quotes asserting platform ownership by the cloud provider [63].

Following this introduction on trust, attestation and execution isolation, Section 2 introduces the cloud infrastructure system model.

2 The Cloud Infrastructure Model

As defined in [64], "cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (such as networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." Cloud computing has evolved into a complex system for delivering various computing and communication resources, while abstracting aspects such as physical resource ownership, operation and geographic location.

The OpenStack project has created a popular open-source cloud management platform that allows to set up, operate and maintain large-scale cloud computing deployments [65, 66]. Since its first release in 2010, OpenStack has had a rapid community-driven evolution and is at the time of writing at its sixteenth release; the collaborative and open-source structure of this initiative facilitated the adoption of OpenStack in research projects. In this thesis, the OpenStack platform was both a foundation for the cloud infrastructure system model and the platform of choice for prototype implementation and evaluation.

This section starts with a brief discussion of the cloud operation and service models, followed by a review of the logical components of a cloud infrastructure deployment based on the OpenStack project.

2.1 Operation and Service Model

Two important aspects of the cloud computing paradigm are its service models and deployment models. There are three widely adopted service models for cloud computing:

- *Service provisioning* (SaaS) allows users to access the cloud service provider's applications deployed on a cloud infrastructure. The underlying implementation and deployment is normally abstracted from the user and only a limited set of configuration controls are made available. Similarly, data created or collected by SaaS applications - which are a type of service provisioning - is transparently stored in the cloud infrastructure. Examples include applications such as word processing [67], enterprise resource planning software [68], image recognition for robotics [69] and various network functions such as firewalls [70] under the umbrella term *network function virtualization* [71].
- *Platform provisioning* (PaaS) allows the users a wider range of capabilities, including software development tools, middleware, Software Developing Kits (SDKs), and Application Programming Interfaces (APIs). Platform providers commonly support run-time environments, such as content delivery networks, mobile applications, and large-scale data processing platforms [72].
- *Infrastructure provisioning* (IaaS) allows tenants to remotely access processing power, disk storage, random access memory and network capabilities. Tenants

can develop, deploy and run arbitrary software and networked services. In the IaaS model, tenants access a sandboxed environment, determined by the leased resource *quota* enforced by the cloud provider. Within the limits of the quota, tenants have full control over the provisioned virtual resources. However, tenants cannot access the underlying cloud management infrastructure (such as the cloud platform management agents deployed on the host servers) or configuration of the physical hosts (such as BIOS settings).

Four cloud deployment models are commonly known: private, public, community, and hybrid clouds [64]. In private deployments, all components of a cloud deployment are potentially under the full control of the tenants, such that tenants can configure the hardware, network and software components. In public, community, and hybrid deployments, the cloud deployment infrastructure is either partially or fully placed on the premises of other entities, hence limiting the tenant's capabilities to monitor and control the infrastructure. In the context of the current thesis, we focus on the distinction between private deployments and other types of cloud deployments.

2.2 OpenStack Architectural Overview

OpenStack is a collection of independent components that intercommunicate through public APIs and collectively form a robust cloud computing platform. The logical architecture of OpenStack is illustrated in Figure 2.8. Below, we review the main components of the OpenStack infrastructure cloud model; similar components are present in most cloud infrastructure platforms and deployments.

Compute Service

Compute is a core component responsible for provisioning and management of compute hosts. Compute service instances run on the hosts in a cloud deployment to support virtualization management tasks. *Hardware virtualization* [73] allows to multiplex physical compute hosts among virtual machines or unikernels. Configuration of compute hosts can have far-reaching security implications. For example, a high instance per server density (over-commit) is beneficial from a resource utilization perspective when guests do not perform CPU or network-intensive tasks [74]. However, a higher over-commit ratio leads to increased interference from collocated virtual machines [74] and higher potential for side-channel information leakage [20, 75].

The choice of CPU architecture and vendor, along with server OEM determines the availability of - and support for - hardware-enabled security features. Security features such as AMD Secure Encrypted Virtualization [76, 77], TrustZone [78, 79], Intel SGX [50, 51] or SecureBlue++ [80] can protect computation and data both directly on the physical hosts and in virtualized environments deployed on such hosts [81]. However, such features differ wildly in their security models, firmware support and functionality.

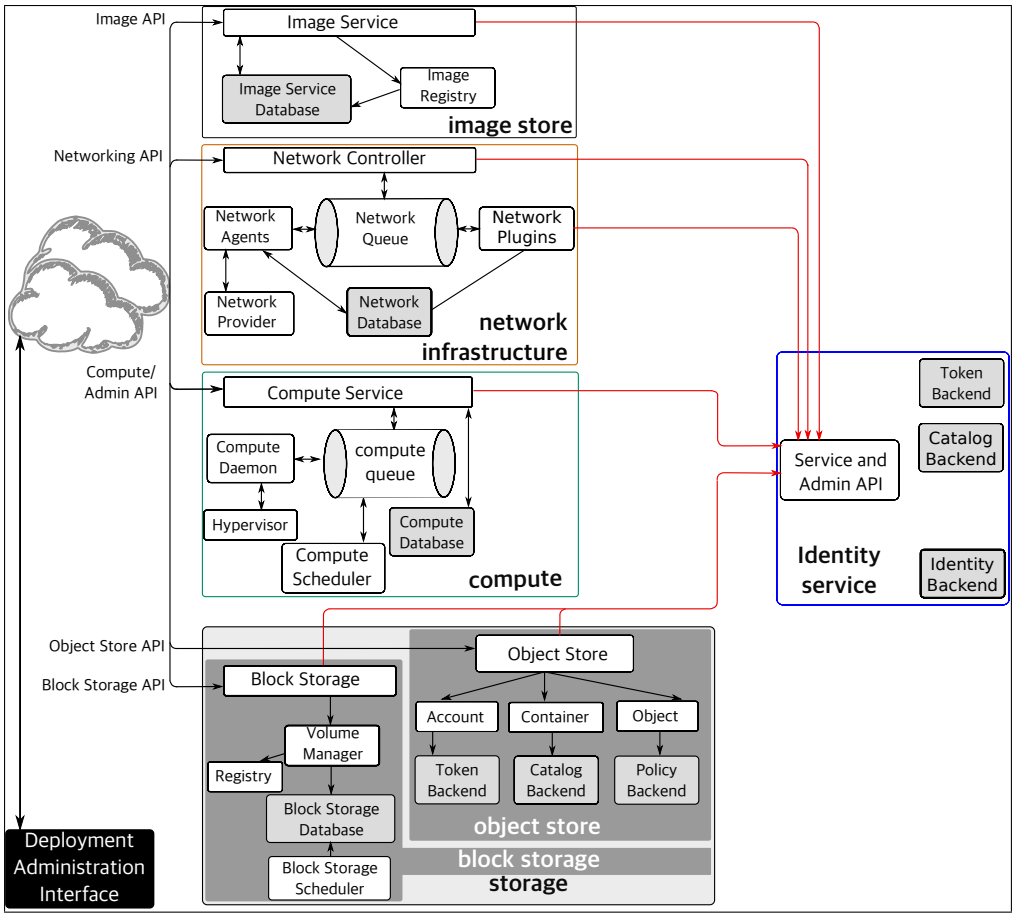


Figure 2.8: OpenStack Logical Architecture.

OpenStack compute component (Nova) was used in the prototype implementations described in Papers A, B included in this thesis.

Storage

Storage is essential for both infrastructure management and resource provisioning in enterprise deployments. From an IaaS management perspective, storage capacity is necessary for maintaining databases of tenant accounts, network topology information, authentication credentials, tokens and policies, along with multiple other critical data, i.e. *management data*. From a resource provisioning perspective, storage capacity is required for VM operating system execution along with additional file systems for storing VM images, as well as any further persistent storage for VM instances, i.e. *tenant data*.

In terms of confidentiality, integrity and availability, the security concerns and requirements towards tenant data and management data differ depending on the threat model. In this context we distinguish between two data persistence categories:

- *Ephemeral storage* does not persist over virtual machine instantiation cycles. For example, such storage can be allocated to virtual machine instances used for data processing tasks that do not produce data that must be persisted. From a tenant point of view, data on ephemeral storage disappears when a virtual machine is terminated. Beyond RAM, this includes *ephemeral disks* [82] - virtual disks that can be mounted to and used by a virtual machine during its operation but are destroyed once the instance is terminated.
- *Persistent storage* outlives any other resource and is available regardless of the state of the virtual machine instance. Infrastructure providers use persistent storage to store infrastructure management data.

Figure 2.8 illustrates two types of storage in enterprise deployments, namely block storage and object storage, further discussed below.

Block Storage Tenants use block storage to add storage to a virtual machine and maintain access to it even after the instance has been terminated. Tenants use the block storage API to create and operate persistent block storage volumes on servers. Block storage is accessed through a block device that can be partitioned, formatted, and mounted. Such storage is appropriate for performance-sensitive scenarios - such as database storage, expandable file systems, access to raw block-level storage, etc. Block storage *volumes* are located either off compute host storage - on a shared file system; on compute host storage - on a shared file system; or on compute host storage - in a dedicated file system. As illustrated in Figure 2.8, the high-level structure of a block storage component in an enterprise deployment is as follows: based on requests received through the *block storage API*, a *volume manager* operates storage partitions enumerated in a *block storage database* and described by meta-data information in a *registry*. The physical location of volumes is determined by the block storage scheduler based on deployment architecture and relevant policies. Examples of block storage back-ends include Ceph [83] and GlusterFS [84].

OpenStack block storage (Cinder) was used in the prototype implementations described in Papers B, C included in this thesis.

Object Storage Tenants use object storage to store, expose and manage data as *objects* instead of files or blocks. Stored objects contain a variable amount of meta-data, to facilitate indexing and data management in large-scale data stores. The design of object stores prioritizes horizontal scalability across multiple hosts (in the order of hundreds or thousands) and high availability; it is often tuned towards read-intensive data access patterns. Object stores support larger namespaces and eliminate name collisions by enabling addressing and identification of individual objects by unique identifiers within a

bucket, or across the entire system. External entities access data stored in object stores through an API - such as the REpresenational State Transfer (REST) - that is limited to storage and retrieval of files and that does not support mounting directories (such as in the case of a file server). Figure 2.8 includes a high-level logical structure of an *object store*, which stores *object* data from multiple *accounts* in replicated *containers* located across a set of hosts in the deployment. Examples of object stores include Dynamo [85], PNUTS [86], Haystack [87], Azure (Blob storage) [88] and Ambry [89].

OpenStack object storage component (Swift) was used in the prototype implementation described in Paper D included in this thesis.

Image Service

In order to reduce data transfer and VM isntance launch time, enterprise deployments often maintain a set of virtual machine images offered to tenants for instantiation. The *image service* is a repository that stores and versions VM images available to tenants. Integrity of the VM images made available through the repository is critical for protecting tenant data and computation. Furthermore, images provided through the repository must be patched and maintained up to date [90].

Identity service

The *identity service* maintains and provides tenant account and identity information. This service communicates with all user-facing components of the deployment for authentication and authorization purposes. Security of the identity service is essential for preventing impersonation and privilege escalation attacks in cloud deployments [17].

Network infrastructure

Large-scale enterprise deployments implement a wide range of network topologies and architecture models. SDN is a popular architectural model in cloud deployments. Along with flat networks with Virtual Local Area Networks (VLANs) for tenant isolation, network control can take advantage of the SDN model and create massively scalable multi-tenant virtualized networks. The extension framework (*network plugins* in Figure 2.8) allows to deploy and manage software implementations of additional network services, such as load balancing, firewalls and virtual private networks. Figure 2.9 illustrates the types of networks present in cloud deployments:

- The *install* or *out-of-band network* enables deploying software images to compute hosts.
- The *internal* or *management network* enables communication between compute, storage and management hosts.

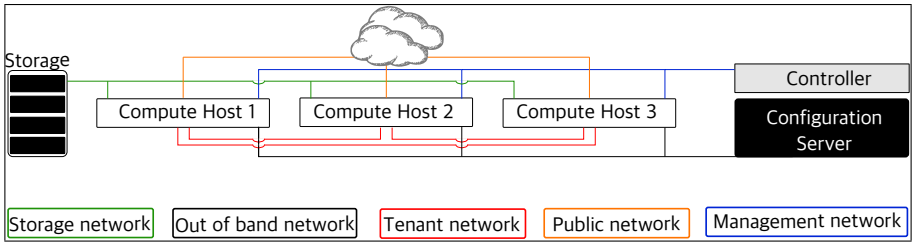


Figure 2.9: Types of networks typically present in a cloud infrastructure deployment.

- The *tenant network* enables communication among instances.
- The *storage network* enables communication between compute hosts and storage back-ends.
- The *public network* enables instance access to the public network space; public network addresses are handled by a deployment-specific network agent on network controller nodes.

Network isolation is important to prevent interference from other networks in case of data-intensive transfers. Section 3.3 presents further aspects of SDN, along with security considerations for network connectivity in infrastructure cloud deployments.

Management Components

Depending on the use case and usage model of the infrastructure cloud deployment, additional common but deployment-specific components may be present. Such components include the log aggregation subsystem, telemetry components collecting usage data for billing purposes, or a dashboard displaying deployment status information.

2.3 The Software Defined Networking Model

Cloud computing relies on hardware virtualization to operate many computing workloads deployed in self-contained, migratable virtual appliances [91]. Such atomic units were implemented as VMs at first and later complemented with various operating system virtualization approaches, collectively denoted as *virtual endpoints*. While the existing switching hardware and routing protocols can support connectivity between virtual endpoints (see "hairpin switching" in Figure 2.10), they lack the flexibility and scalability necessary for effectively managing connectivity at large [92]. To address this, the *Clean slate* initiative [92] proposed to decouple network forwarding from control and management logic. This initiative gained wide support and later evolved into the SDN model, with several key contributions playing a major role in its evolution.

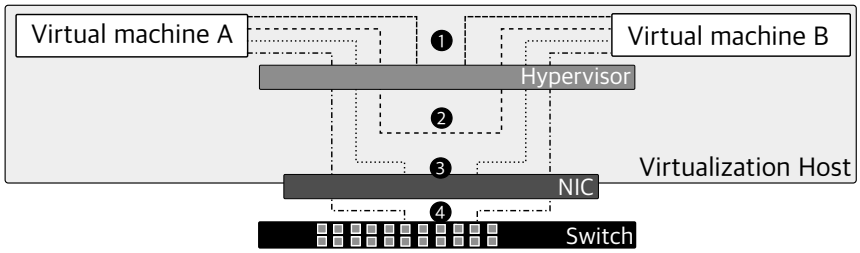


Figure 2.10: Inter-VM communication paths: (1) virtual switch; (2) host-local (native bridging); (3) virtual queues in the Network Interface Card (NIC); (4) external switch, i.e. "hairpin switching".

Figure 2.10 illustrates alternative software switching approaches for communication between virtual endpoints. Along with other approaches, virtual switching offered a scalable and flexible alternative to routing of both layer 2 and layer 3 communications.

In the SDN model, software or *virtual switches* route packets according to *flows* stored in the switch flow table. In turn, each flow entry consists of match fields, counters, and a set of instructions that the switch applies to matching packets. Flow entries are installed by a logically centralized controller, according to policies defined by human administrators and network management applications. While applications - also known as "middleboxes" - often appear as hardware components (such as firewalls and traffic shapers) in traditional networks, software implementations (so-called Virtual Network Functions) are better suited for dynamic SDN deployments.

Communication between virtual switches (on the logical *forwarding plane*) and the network controller (on the *control plane*) is done through a so-called "southbound API". However, there is no single widely adopted corresponding interface between applications (on the *management plane*) and network controllers (i.e. a "north-bound API"). Network controllers implement a variety of interfaces [93, 94]. The SDN system model is further described and analysed in Papers E, F, G included in this thesis.

Several implementations of SDN infrastructure components were put forth by both industry and academia in the years following the introduction of SDN. Two of the SDN projects - Open vSwitch [95] and OpenFlow [96] - have become *de-facto* standard implementations.

Open vSwitch is a popular virtual switch implementation, which has heavily influenced the evolution of Software-Defined Networking. It is designed to have *minimal* processing logic and to forward packets that match installed rules; unmatched packets are either sent to the network controller or discarded. However, results reported in [97] demonstrate that complementing the switch implementation with control logic improves forwarding performance and facilitates management tasks.

OpenFlow is a widely adopted southbound API. It specifies the format of messages that a network controller can use to add, update, and delete flow entries in flow tables.

OpenFlow is supported by popular *network operating systems*, such as NOX [98], Rosemary [99], ONOS [100]. In the context of SDN, network operating systems are a part of the network controller abstraction, logically placed on the control plane.

Following this introduction of the cloud infrastructure model, Section 3 introduces some security aspects of the cloud infrastructure system model.

3 Security in Cloud Infrastructure

Since 1996, when the term "cloud computing" was first coined [101], the technology behind this concept has progressed from a bold vision to massive deployments in multiple application domains. However, the complexity of the underlying technology introduces novel security risks and challenges. Earlier research on cloud security identified challenges across the stack: from cross-site scripting attacks allowing to take over the deployment [18]; to side-channel attacks causing information leakage from virtual machines [17, 20]; to attacks capable of extracting sensitive information from isolated execution environments specially designed from cloud environments [56, 57]. While none of the attacks mentioned above are "cloud-specific" and each of them belongs to an own narrow field of security research, malicious actors can apply them in the cloud computing context.

Having introduced the necessary background, we next provide an overview of research contributions across several aspects of cloud infrastructure security (see [102] for a more comprehensive overview). In this context, we also introduce the contributions of this thesis.

3.1 Confidentiality and Integrity of Computation

On-demand pooling of virtual resources in a multi-tenant model is one of the defining features of cloud computing [64]. As cloud computing relies on hardware virtualization to multiplex physical infrastructure, ensuring the security of VM instances[2] is an essential prerequisite for the trustworthiness of a cloud provider.

For private cloud deployments, monitoring the security of VMs is facilitated both by stricter control over tenant identity and access, as well as by VM introspection [103], which offers solutions such as VM-based intrusion detection, forensic memory analysis, and low-artifact malware analysis [104]. The task is more complex for public cloud providers: on the one hand, they must ensure that the anonymous, mutually mistrustful or competing tenants do not break the isolation of the tenant domain; on the other hand, they have very limited introspection capabilities due to both the security risks involved [105] and the privacy concerns of its tenant end-users [106]. Assuming that the physical security of the host systems is ensured [32, 107, 108], the adversary may attempt to break the isolation provided by commodity hypervisors in order to access the compute environment of a target VM instance. The adversary is in this case represented by either a peer tenant in a public cloud deployment, or an administrator of the cloud management platform. A broad range of approaches to virtualization security are available.

CloudVisor [107] is a minimal security hypervisor that uses nested virtualization and a dynamic RoT for measurement functionality to protect the memory and I/O access of virtual machine instances. Furthermore, the solution enforces memory isolation and

[2] This is equally relevant for the more recent virtual appliances such as containers and unikernels.

disk storage protection for virtual machine instances. In this approach, the TPM stores the integrity measurements of the TCB and enables remote attestation of the security hypervisor.

Credo [109] relies on a minimal and measurable TCB to protect a virtual machine guest's I/O, as well as its memory and CPU state. The Credo TCB includes a hypervisor, TPM and TXT firmware along with additional hardware and firmware components launched using a DRTM; following the launch procedure, integrity measurements of the TCB are stored in resetable TPM registers. The virtual machine guest's memory and CPU state are protected through a combination of scrubbing memory pages prior to release, limiting the number of intercepts forwarded to the management partition of the hypervisor[3] and enhancing access control of virtual registers accessible by intercepts. The solutions described in [110] and [61] reuse some of the approaches from Credo to create isolated application sandboxes in cloud environments.

Mutual isolation between users and computing infrastructure providers was also addressed by Butt et al. in [111]. The solution splits the control over the tenant and operator administrative domains and introduces in this approach *mutually-trusted service domains*, to resolve the tension between the introspection needs of the cloud service provider and the confidentiality and integrity requirements of the cloud tenants. Such service domains are based on introspection policies mutually agreed between the cloud infrastructure provider and tenants. Cloud infrastructure providers deploy workload management and introspection code that executes in compliance with the established introspection policies. Tenants inspect measurements of the code collected at bootstrap time and stored in the registers of tenant-specific virtual TPMs, in order to attest the integrity of the introspection code and evaluate its compliance with the established introspection policies.

While the approaches enumerated above - as well as many others [108,112–114] - address the security of the virtualization software or of the launched virtual machines, they do not enable tenants to verify that such mechanisms are actually used by cloud providers. A tenant following the *direct delegation of trust* approach may choose to rely on third party certification to confirm that the hypervisor security features described above are present on compute hosts. However, this leaves the tenant exposed to bugs and vulnerabilities in the cloud management platform or workload scheduling code, such that the virtual machine is launched on an arbitrary platform without the hypervisor security features requested by the tenant.

This issue is addressed by the *verifiable trusted launch* introduced in Paper A. This approach enables tenants to verify that the VM instance they communicate with was launched on a platform with a certain TCB, without exposing the details of the TCB itself. This is achieved through remote attestation of the platform (leveraging the functionality of a TPM), combined with sealing of a user-generated token unsealed only if the virtualization host maintains the expected TCB. The work described in Paper A was based on research conducted prior to, or in parallel with the emergence of hardware support for "cloud-native" TEEs such as Intel SGX, AMD Secure Pro-

[3]The approach is designed based on the Hyper-V hypervisor.

cessor, SecureBlue++ [80] and AMD Memory Encryption [76, 77] and others [115]. Such TEEs enable new security models for cloud computing, have remote integrity attestation functionality, and allow tenants to deploy rich applications [63], modified operating systems [61] or virtual appliances [116] in a protected environment with a minimal TCB. However, the performance of software placed in such TEEs reported by the authors is lower compared to native execution [61, 63, 116]. The authors of SCONE [116] highlight the trade-off between the size of the TCB and the execution performance.

3.2 Cloud Storage Protection

Storing data backups in remote infrastructure has a decades-long history [117]; early works primarily addressed reliability through disaster recovery [118–120], with little regard to the security of the remotely stored data. However, the dynamicity and multi-tenancy of cloud infrastructure, along with new uses of storage for internal cloud management operations have introduced new attack vectors.

Whether used for storing internal cloud infrastructure management data or allocated to VM instances, block storage is often represented by logical volumes assembled from multiple disk partitions that may be physically scattered throughout the deployment and replicated within or across datacenters. Beyond the obvious challenges of pinpointing the physical location of data at any moment in time, peer tenants, rogue system administrators and state-level adversaries may attempt to break the integrity and confidentiality of data stored on block storage allocated to VM instances.

Results from industry security research reported in 2011 a vulnerability where block storage allocated to a VM instance was later re-allocated to new VM instances while still containing old data [21]. Disk wiping [121] - or in case of encrypted volumes simply by reliable destruction of the encryption key - can effectively resolve such issues. Below follows a brief account of notable research efforts towards protection of data in remote infrastructure.

Kamara and Lauter describe a *cryptographic cloud storage* architecture in [122]. The architecture provides confidentiality and integrity of data stored on remote infrastructure, while providing availability, reliablity, efficient retrieval and data sharing. Its core workflow is as follows: customer data is transferred to the cloud storage through a *data processor*; once deployed, its integrity can be verified at any moment using a *data verifier* component; search and retrieval of customer data segments is done using search tokens issued by a *token generator*; finally, third parties can access and query the customer data using credentials issued by a *credential generator* according to a user-defined access control policy. The proposed architecture is enabled by several advances in cryptography supporting the requirements for cloud storage - namely searchable encryption [123, 124], attribute-based encryption [125] and proofs of storage [126].

With *CloudProof*, Popa et al. address the lack of security aspects in the SLAs of cloud service providers. The proposed system detects and proves instances of security prop-

erty violation under the threat model of an untrusted cloud provider. The addressed security properties are: *confidentiality*, achieved by client-side content encryption prior to deployment in the cloud; *integrity*, achieved through a combination of client-owned private signature keys used to sign updated data blocks and public verification keys used to verify integrity of data blocks at read time; write-serializability (i.e. reliable versioning of stored data), achieved using *attestation chains*, i.e. chains of hashes over the data block version number and content computed at each data block update; *freshness* (i.e. guarantee to provide the latest data block version), verified by checking the correctness of the attestation chain for a selected data block.

Approaches such as *FADE* [127] implement policy-based file assured deletion, in order to effectively prevent access to the stored files by the cloud service provider upon revocations of file access policies.

Excalibur combines policy-sealed data with Ciphertext Policy Attribute-Based Encryption [128] in order to protect data in cloud infrastructure [129]. Policy-sealed data is a trusted computing abstraction designed for cloud services that leverages TPM functionality. It allows data to be sealed (i.e., encrypted to a customer-defined policy) and then unsealed (i.e., decrypted) only by hosts whose configuration matches a given policy.

Mylar is a security framework that combines data protection in the cloud with end-to-end protection of client data access [130]. Mylar consists of four main components: (i) a server-side library implements keyword search over encrypted data on the server; (ii) a client-side library intercepts the data transfer with the server and manages data encryption and decryption, as well as client key management. (iii) a browser extension verifies the integrity of the client-side web application code; (iv) an optional identity provider verifies the link between user names and keys. This approach can benefit from both the continuous advances in searchable encryption for keyword search [123, 124] and from advances in hardware-supported isolated execution environments [50] that can help reduce the reliance on the identity provider component.

The contributions described above presented a variety of approaches to storing, searching through, and sharing confidentiality and integrity protected data in cloud deployments. In Paper B we present a complementary approach based on combining several previous results [131–133], that implements a comprehensive protection mechanism including virtualization host attestation prior to virtual machine instantiation (*verifiable trusted launch*) and user-controlled storage protection transparent for the VM instances (*domain-based storage protection*). Verifiable trusted launch, initially introduced in Paper A, provided a means for users to verify that the VM instance they communicate with has been launched following the trusted launch protocol on a platform with an attested TCB (see Section 3.2). Domain-based storage protection, initially described in [132] and the related patent [133] allowed encryption of persistent VM block storage by the hypervisor, transparent to the guest VM and independent of the implementation of the encryption libraries in the VMs.

By shifting disk encryption to the underlying hypervisor where it is managed by a dedicated *secure component*, the approach described in Paper B reduces the attack surface by maintaining all key material in the secure component rather than in the

virtual machine instances. In the same time, encrypting locally on the virtualization host the virtual disks mounted to VM instances allows to reduce the cost of cloud storage by storing data in other deployments with more relaxed security guarantees. Finally, maintaining control of the data encryption keys externally from the virtualization host allows tenants to seamlessly swap cloud services without the hurdle of secure data migration between infrastructure deployments.

Beyond using TPM functionality for key sealing, the approach described in Paper B relies on a *secure component* - a verifiable execution module performing confidentiality and integrity protection operations on VM instance data and key management. The use of a secure component in this work was inspired by approaches such as SecVisor [112] and CloudVisor [107], which relied on a verified software module executing at the highest privilege level. The solutions in [112] and [107] required placing the verified software module at the highest privilege level in order to protect it from a potentially malicious host operating system. In a novel implementation, this approach can leverage the increasing hardware support for TEEs on commodity platforms [115,134] by deploying the secure component in one such isolated execution environment. The solution is generic enough to be deployed with a variety of available or upcoming TEE implementations. We used the block storage (Cinder) to implement domain-based storage protection in both Paper B and in [132].

In Paper C we extended the solution with access control for multi-tenancy support. In particular, we introduce extensions that allow tenants to control, at instance launch time, its read and write access rights over a storage device.

Data geolocation

Concerns about the physical location of data and its availability in different jurisdictions [135] gained further importance as state regulations on data placement caught up with technological developments. Such regulations are referred to as *data localization* [136, 137], defined as "a policy whereby national governments compel Internet content hosts to store data about Internet users in their country on servers located within the jurisdiction of that national government" [137]. Several countries have adopted data localization regulations regarding storing, processing, or handling of certain types of data (the specific types of data differ depending on the country). Examples of such regulations include: *Personally Controlled Electronic Health Record Provision* in Australia [138], *Cybersecurity law* in China [139]; *Telecommunications Act* in Germany [140]; *National Data Sharing and Accessibility Policy* in India [141]; *Information and Electronic Transaction Law* in Indonesia [142]; *Federal Law No. 242-FZ* in Russia [143]; and *Defense Federal Acquisition Regulation Supplement: Network Penetration Reporting and Contracting for Cloud Services* in the United States of America [144] (see [136] for more details).

While data localization is a fairly recent term, a large body of research investigated the closely related aspect of data location, which remains an unsolved issue in the context of cloud computing. This is partly caused by the architecture of cloud computing de-

ployments, which rely extensively data replication and load balancing to ensure elastic scalability [64] and high availability at all times. Watson et al [145] showed that there are limits to the accuracy of verifying the location of data in a cloud storage. The authors demonstrated that when a malicious cloud service provider colludes with malicious hosts, it is unfeasible for a user to correctly verify the exact location of files. Furthermore, Watson et al. were the first to take into consideration cases where two or more malicious hosts collude and make copies of the stored files. This assumption led them to posit that the task of restricting the geographic location of data is impossible. The authors have suggested a proof of location scheme that can be used by a user to obtain the location of a stored file. However, the proposed scheme required significant supplementary infrastructure, as the solution relied on the existence of trusted landmarks responsible for verifying the existence of files on a host.

Similar to other solutions that rely on distance-bounding protocols [146–148], and latency-based techniques [149, 150], most data geolocation approaches not not address the question of limiting data accessibility according to geographic location.

NIST described a proof of concept for geolocation of data in the cloud [151], relying on the combination of trusted computing, Intel TXT and a set of manual audit steps to verify and enforce data location policies. The use of hardware-based isolation environments has created the capability to *restrict* accessibility of cleartext data across jurisdictions (also refereed to as "geo-fencing") [152, 153] however still relying on the cloud storage provider to enforce the data location policies.

In Paper D, we describe an approach to control access to cleartext data in data centers, based on their geographic location. The approach is based on sealing cryptographic material used to protect data integrity and confidentiality to approved geolocation cells described by a set of geolocation coordinates. We used a hardware RoT to unseal the cryptographic material only if the geographic location of the platform is within one of the approved geolocation cells. As a result, data can only be accessed in plaintext if the storage is placed in one of the geolocation cells approved by the data data administrator. In all other cases, the data remains encrypted but can be replicated for redundancy purposes. While in the design and implementation phases we relied on gelocation data reported by the Global Positioning System (GPS), the solution can be implemented with other geolocation systems. The prototype was implemented using the Swift object store [154] (part of the OpenStack project); however data stored in other types of storage (such as block storage) could as well be geo-fenced based on the same principles.

While the approaches presented above provided multiple solutions to the data geolocation problem, this aspect remains unresolved in practical deployments, as users must trust the statements of service providers regarding data location.

3.3 Security in Software Defined Networking

SDN has challenged the network infrastructure security best-practices evolved over the decades since packet-switched digital communication gained overwhelming pop-

ularity [155]. This change triggered renewed research in network management security, re-definition of adversary models, re-assesment of security challenges and attack vectors. While significant work has been done on the security of the SDN model, important challenges remain. A review by Scott-Hayward et al. lists multiple security issues spanning across the architectural layers of SDN [156]. These include unauthorized access, data leakage, data modification, malicious or compromised applications, denial of service, configuration issues, as well as system level SDN security - such as lack of visibility of the network state. Since SDN has seen considerable security research efforts, many of the issues have been addressed to some level in [157–162] along with multiple other contributions. Along with the existing security challenges and solutions, a review by Ahmad et al. [163] highlighted several future directions for security in SDN, such as class-based application security, security scalability in SDN, synchronization of network security and network traffic, network security automation, and identity location (which could be addressed by applying approaches such as the host identity protocol [164]). In Paper E we described an SDN-specific threat model as well as attack vectors identified based on the adversary capabilities and prior work [155,165]. This review helped identify a list of ten security requirements towards SDN deployments. A part of the challenges identified in [163] (in particular class-based security and partly security scalability), as well as part of requirements identified in Paper E were later addressed through the mechanisms described in Papers F, G and in on-going work [166].

Protecting Software-Defined Networking Components

The adoption of software network components executing on commodity operating systems has increased the attack surface of network infrastructure: as highlighted in the European Telecommunications Standards Institute security specification for network function virtualization, software network components are vulnerable to both bugs in their own code and to risks introduced by the underlying trusted computing base (such as hypervisor introspection, vulnerabilities in the OS) [14]. Antikainen et al. described in [167] a series of attacks that can be performed once a virtual switch has been compromised. Such attacks include eavesdropping control plane communication, network state and topology spoofing, as well as denial of service.

Some of the above attacks can be prevented by protecting southbound communication using standard network security protocols such as TLS with server- or mutual authentication. However, while using TLS prevents certain classes of attacks - such as topology spoofing and traffic eavsdropping - it does not solve the issue but rather shifts the focus on protection of the authentication credentials. Virtual switches commonly contain kernel modules for performance reasons [95], which means that a virtual switch compromise can lead to a complete compromise of the compute host [168]. Beyond revealing authentication credentials, a compromised host can be used to launch attacks throughout the deployment, both on the network and compute infrastructure. Integrity monitoring and integrity verification can be used to prevent forwarding plane compromise. For example, the proliferation of novel and versatile TEEs opened the possibility of using TEEs to strengthen the security guarantees in SDN deployments.

Several related scenarios of protecting SDN components were investigated in [169] - protecting integrity of virtual network function states using TEEs; in Paper F- protecting integrity of forwarding plane components using TEEs; in [170] - enhancing security of the Tor ecosystem by using TEEs; and [171] - packet processing in TEEs. To the best of our knowledge, the contribution described in Paper F (introduced below) was the first to propose ensuring authenticity, as well as protecting confidentiality and integrity of software switches by placing them in TEEs.

In Paper F we address some of the risks introduced by the widespread use of virtual switches in network deployments. Virtual switches executing on commodity operating systems are often assigned the same trust level and privileges as hardware switches. However, commodity operating systems contain security flaws that can compromise virtual switches. On the other hand, the absence of a secure and scalable process for authentication and authorization of virtual network components prior to enrollment introduces the risk of impersonation of components enrolled into the SDN infrastructure.

We addressed both of the above risks in Paper F by deploying a custom implementation of simple a software switch in a TEE with remote attestation capability. We use the functionality of the TEE to distribute the key material used to establish an authenticated, confidentiality and integrity protected communication channel between the software switches and the network controller.

Another result introduced by this paper are the so-called *ephemeral flow-specific pre-shared keys*. TruSDN leverages two aspects of the popular OpenFlow protocol - namely centralized network control and forwarding the first packet of a flow to the network controller - to deliver on demand ephemeral pre-shared keys to communication endpoints. This approach allows to relax requirements for high-quality entropy available to virtual endpoints and reduce the time to establish a TLS session. Along with a further extension of this work [166], this approach highlights the opportunities for increased use of TEEs to secure network infrastructure.

Access Control in Network Infrastructure

Beyond the goal of separating forwarding and control planes of network infrastructure, SDN has created new possibilities for network management and network application deployment. While the initial abstractions of virtual switch, flow rules, southbound API and logically centralized network controller are available, the northbound API and the access control to SDN remain undefined.

The access control scheme for SDN controllers proposed in [172] - followed by a prototype described in [173] - mimics access control schemes for operating systems, with contextual adjustments. In particular, it introduced the logical separation of network components to reliably assign the object and mode of access, along with support for delegation of access permissions to network components. The *permissions* supported by the access control system included both operating system-type *read* and *write* permissions, and additional permissions for reading statistics, requesting information about

the objects and modifying their state. The *policies* supported by the proposed access control scheme reflect the separation of the administrator and management domains of the network deployment: a mandatory access control component managed by the network administrator allows to assign permissions for network subjects to perform actions on the network deployment; a discretionary access control component enables subjects to delegate the objects they own to peer subjects.

Despite the research efforts on network virtualization and SDN access control [172–176], replacing hardware network middleboxes with software implementations is not sufficient to ensure security while supporting multi-tenancy and deployment flexibility of virtual network infrastructure. In part, this is due to the lack of access control for network management applications, caused by missing or incomplete definitions and understanding of other network abstractions [172]; this also impedes defining an expressive north-bound API. The *intent framework* [177,178] implemented in the ONOS network controller [100] enabled a solution for SDN access control for network applications.

In manuscript G, we describe a north-bound network resource access control API for network operators and application developers. The approach is based on three principles: broad definition of *resources* in the software defined network model (partly induced by the additional permissions introduced in [172]); explicit assignment of resources and privileges for network applications; reuse of the Capability-Based Access approach, where subjects carry an unforgeable token describing their access rights in order to access a resource.

The definition of network resources adopted in manuscript G includes: *device resources* such as hardware or virtual switches and installed flows; *data resources*, such as the network topology and flow statistics; and *control resources*, such as management and security policies in the deployment. This definition enabled a change in the way network applications interact with the network controller (and implicitly the rest of the network deployment): access to resources in the network deployment is explicitly described for each application (or class of applications) using an *access mask*, which describes both the resources that the application can access and the actions that it can perform on those resources. An unforgeable token created based on the access mask (along with other information) is then supplied along with the *intents* submitted by applications to the network controller; the token is verified by a reference monitor in order to prevent violation of access policies in the event of a network controller compromise.

Ongoing Work and Future Challenges

In our on-going work [166], we aim to provide comprehensive user security guarantees regarding the trustworthiness the network infrastructure on all planes. This is done by leveraging two emerging aspects: logical centralization of network infrastructure offered by SDN; and increasing availability and hardware support for TEEs on commodity platforms. Future security research challenges in SDN include run-time validation of VNF states, compliance verification of forwarding network functionality, as well as identification of security mechanisms for federated SDN infrastructure.

4 Concluding Remarks

A common theme reoccurring throughout this thesis is the design and application of mechanisms to assess trustworthiness in cloud infrastructure deployments. The aim is to complement reputation and audits - *holistic* trust indicators operating on a scale of months and years - with additional, *focused* mechanisms operating on a finer scale tied to events such as component deployment, component initialization or data access. Establishing the trustworthiness of cloud deployments is relevant to both tenants and end-users on one hand, and operators of cloud infrastructure on the other hand.

For tenants and end-users, disclosures such as [1, 2] have demonstrated the naivety of relying on reputational trust for the confidentiality of data stored with major cloud service providers. This provides additional motivation to develop reliable and fine-grained mechanisms for assessing security of remote computing deployments before taking the decision to transfer sensitive data and computation to them.

For cloud service operators, the revelations that powerful adversaries have the capability to subvert their core internal infrastructure have highlighted the need for trust establishment mechanisms in enterprise deployments. Furthermore, the growing frequency of targeted attacks performed by increasingly well established Advanced Persistent Threat groups [179] re-emphasized the need to add additional layers of protection - well in line with the "defense in depth" approach.

Scientific publications

Overview

The goal of this thesis is to improve the security of cloud computation, data storage and network communication within cloud deployments. Each of the papers included in the thesis addresses either one specific aspect, or a combination of aspects in a comprehensive solution. Most of the papers include an implementation and evaluation of the described solution (see Table 2.1).

Table 2.1: Overview of the publication content areas and features.
Legend: □ = design; ⊡ = implementation; ■ = performance evaluation; * = aspect addressed; (*) = re-applied solution proposed in an earlier contribution.

		Content area			
		Virtual Machine Launch	Storage	Access Control	SDN
Paper A	□⊡	*			
Paper B	□⊡■	*	*		
Paper C	□⊡■		*	*	
Paper D	□⊡■	(*)	*		
Paper E	□				*
Paper F	□⊡■				*
Paper G	□⊡■			*	*

Publications included in this thesis address the following aspects:

- **Launch of virtual machines on attested hosts** (addressed in papers A, B) focuses on providing user guarantees regarding the compute host where a tenant virtual machine has been instantiated.
- **Storage protection** (addressed in papers B, C, D) focuses on several aspects of storage protection, such as transparent data encryption on compute hosts and data geo-fencing in geographically distributed cloud deployments.
- **Network infrastructure protection** (addressed in papers E, F, G) focuses on addressing some of the core threat vectors in the adversary model identified for Software-Defined Networking infrastructure.

Guide to publications Readers aiming to get familiar with the publications are encouraged to start with Paper A, which presents a protocol for launching a virtual machine instance on an attested host with additional, tangible user guarantees. The approach is complemented in Paper B with a *domain-based storage protection* protocol [132, 133] and presents a comprehensive solution for launching a virtual machine instance on a remote host and transparently encrypting its storage volumes using commodity software on compute hosts. Paper B thus introduces the reader to the work on storage protection, extended in papers C, D on respectively adding support for multi-tenancy to the *domain-based storage protection* protocol and geo-fencing plaintext data access on compute hosts based on their geographic location. Paper E is a transition towards security of the SDN infrastructure, as it describes an SDN-specific threat model and main threat vectors, which are later addressed in Paper F and Paper G. Paper F focuses on integrity of forwarding plane components and southbound communication security, while Paper G addresses access control on the northbound API.

Papers A, B, C, D, E, F, are included in *accepted* or *camera-ready* versions; Paper G is included in its *submitted* version. The included papers have been minimally updated to correct grammar and spelling errors and adapt them to the format of the thesis.

Included Contributions

Co-authors are abbreviated as follows: Christian Gehrmann (CG), Mudassar Aslam (MA), Antonis Michalas (AM), Fredric Morenius (FM)

Paper A: Trusted Launch of Virtual Machine Instances in Public IaaS Environments

Originaly published as *N. Paladi, C. Gehrmann, M. Aslam, and F. Morenius, "Trusted Launch of Virtual Machine Instances in Public IaaS Environments", in Proc. 15th International Conference on Information Security and Cryptology, ICISC' 12, pp. 309–323, Springer, December 2012.*

The final publication is available at Springer via `https://doi.org/10.1007/978-3-642-37682-5_22`

Content Prior to starting work on Paper A, several protocols for trusted launch of virtual machine instances on remote platforms have already been proposed, namely [109, 180]. Work on this paper was triggered by a simple question: what can prevent a dishonest provider from claiming to have performed a trusted launch while in effect presenting tenants with an instance executing on a different host with an entirely different software stack? In essence, tenants would significantly benefit from a *guarantee* that they are communicating with an instance launched according to a certain trusted launch protocol. In paper A, we addressed this by proposing and implementing a trusted launch protocol which includes an unforgeable token sealed to a certain state of the platform configuration registers and injected into the file system of the virtual machine instance launched following the protocol. As a result, tenants could verify the presence of the token on the file system and interpret that as a guarantee that the virtual machine instance has been created following a trusted launch.

Original Contributions N. Paladi designed and implemented the protocol for trusted launch of virtual machines with user guarantees, as well as written the initial content of the paper (based on his earlier M.Sc. thesis under the supervision of CG). CG refined the protocol and improved the manuscript both by directly editing it and by providing valuable comments on the content of the paper. MA contributed to the implementation of the protocol by a library for key sealing to the platform, as well as clarifying the inner workings of the OpenPTS stack and the TrouSerS TPM library. FM contributed comments on the adversary model in the context of infrastructure clouds as well as on the content of the manuscript.

Paper B: Providing User Security Guarantees in Public Infrastructure Clouds

Originally published as *N. Paladi, C. Gehrmann, and A. Michalas, "Providing User Security Guarantees in Public Infrastructure Clouds", IEEE Transactions on Cloud Computing, vol. PP, pp. 1– 1, February 2016.*

Content The owners of virtual machine instances executing on remote hosts may require confidentiality and integrity protection of data stored by their VM instances, along with guarantees about the software stack of the virtualization hosts. While encryption of data at rest is widely available from multiple cloud providers, this commands a leap of trust from the end-users, since the encryption keys are stored in the security domain of the cloud provider. Paper B addresses this by combining two previous results: an improved version of the trusted virtual machine launch protocol; a domain-based storage protection protocol first described in [132]; and the related patent [133]. In the presented solution, virtual machines are launched on hosts with an attested software stack and the disk encryption for each virtual machine is performed on the hypervisor level, while the encryption keys are stored in a minimal secure component trusted by the user. A beneficial side-effect of the solution is simplified data migration between cloud providers and the potential to reduce costs by enabling users to store data in commodity third-party storage facilities while protecting its confidentiality and integrity.

Original Contributions N. Paladi contributed major parts of the content along with the description and implementation of the improved virtual machine launch protocol as well as the entire description and partly the implementation of the domain-based storage protocol. CG provided parts of the content and comments that helped significantly improve the manuscript. In particular, his contributions to the security model and security analysis have considerably improved the structure and content of the paper. AM made major contributions to the content of the paper as well as to the improved design and implementation of the Domain-Based Storage Protection (DBSP) protocol.

Paper C: Domain-Based Storage Protection with Secure Access Control

Originally published as *N. Paladi, A. Michalas, and C. Gehrmann, "Domain Based Storage Protection with Secure Access Control for the Cloud," in Proc. 2014 International Workshop on Security in Cloud Computing, SCC '14, pp. 35–42, ACM, June 2014.*

Content Simplified data sharing is among the core benefits of cloud storage, however this aspect was insufficiently addressed in our previous work on the DBSP [132,133,181]. In Paper C we describe and implement a secure storage protection protocol to provide per-virtual machine instance access control and allow tenants to control, at virtual machine launch time, its read and write access rights over a storage device. We introduce an XML-based language framework allowing users to define role-based access control in order to grant access, based on permissions, to other users in the Infrastructure-as-a-Service cloud.

Original Contributions N. Paladi contributed main parts of the content along with the implementation of the protocol for the cloud infrastructure. AM initiated the effort to add access control to the earlier existing domain-based storage protocol, contributed important parts of the content and contributed to the implementation and evaluation of the solution. CG contributed with valuable comments on the content of the paper.

Paper D: Trusted Geolocation-Aware Data Placement in Infrastructure Clouds

Originally published as *N. Paladi, M. Aslam, and C. Gehrmann, "Trusted Geolocation-Aware Data Placement in Infrastructure Clouds", in Proc. 13th International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom '14, pp. 352–360, IEEE, September 2014.*

Content Considering the plethora of regulations on data protection across jurisdictions, limiting the geographic placement of data maintains its relevance. In Paper D we describe a novel approach to geo-fencing the access to plaintext data across data centers. This is done by sealing cryptographic material used to protect data integrity and confidentiality to approved *geolocation cells* described by a set of geolocation coordinates. We use a hardware root of trust to only unseal the cryptographic material if the geographic location of the platform - as reported by a Global Positioning System device - is within one of the approved geolocation cells. Since the time of the publication, storage of geo-tags in TPM registers has become one of the important use cases for TPM usage, mainly popularized by related solutions [13, 182, 183].

Original Contributions N. Paladi contributed the initial idea of restricting plaintext access to data based on geographic location of the storage nodes, as well as most of the content, implementation and evaluation of the solution. MA made significant contributions to the implementation of the solution, in particular to the coordinate data extraction from the GPS device and sealing data encryption key to the TPM PCRs. CG contributed with valuable comments on the content throughout the writing process.

Paper E: Towards Secure Multi-Tenant Virtualized Networks

Originally published as *N. Paladi and C. Gehrmann, "Towards Secure Multi-tenant Virtualized Networks", Proc. 14th International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom'15, pp. 1180-1185, IEEE, August 2015*

Content The Software-Defined Networking model has been widely used to deploy, manage and operate large-scale network architectures. With the separation of data and forwarding planes as one of its core premises, SDN challenges network security best practices and introduces both new capabilities for network security as well as new vulnerabilities. In paper E we analyze the SDN model and describe a range of SDN-specific attack vectors along with high-level approaches to mitigating them. This paper has laid out the groundwork for improvements to SDN security described in Paper F and Paper G.

Original Contributions N. Paladi contributed the entirety of the content in this work. CG significantly improved the paper by providing valuable comments on the content throughout the writing process

Paper F: TruSDN: Bootstrapping Trust in Cloud Network Infrastructure

Original published as *N. Paladi and C. Gehrmann, "TruSDN: Bootstrapping Trust in Cloud Network Infrastructure", in Proc. 12th International Conference on Security and Privacy in Communication Networks, SecureComm'16, Springer, October 2016.*

The final publication is available at Springer via `https://doi.org/10.1007/978-3-319-59608-2_6`

Content The proliferation of data centers and virtual machine endpoints has triggered the advances in virtualising the network equipment. Along with improved flexibility, this introduced new attack vectors and vulnerabilities in the network infrastructure. In paper F we address this by proposing TruSDN, a framework for bootstrapping trust in SDN infrastructure using Intel Software Guard Extensions. TruSDN allows to establish trust in deployed SDN components and protect communication between network endpoints. Paper F further introduces ephemeral flow-specific pre-shared keys, which leverage the SDN model to reduce the number of messages needed for key distribution between endpoints in network infrastructure. The prototype implementation - based on the OpenSGX emulator [184] due to the lack of access to a SGX-enabled hardware platform at the time when the contribution was written - has helped prove the concept and had yield valuable insights for enabling trust in SDN infrastructure components.

Original Contributions N. Paladi contributed the entirety of the content of this work. CG contributed with valuable comments during the solution design and on the content throughout the writing process.

Paper G: SDN Access Control for the Masses

N. Paladi and C. Gehrmann, "SDN Access Control for the Masses." Submitted for review, May 2017.

Content *Software Defined Networking* has so far been successful in defining and refining the abstraction layers on the forwarding and control planes. However, despite a maturing southbound interface and a range of proposed network controllers, the network operating system has currently poorly defined - if any - access control mechanisms that could be exposed to network applications. Available mechanisms allow only rudimentary control and do not provide any procedures to partition access to resources across multiple dimensions. We address this in Paper G by introducing a taxonomy of access models for SDN resources, as well as describing and implementing for the first time a North-Bound access control API and enforcement mechanism for SDN deployments. We demonstrate the feasibility of the solution by implementing it as an extension of ONOS, an open source SDN controller.

Original Contributions N. Paladi contributed the entirety of the content of this work. CG significantly improved the paper by providing valuable comments during the solution design and on the content throughout the writing process.

Further Contributions

Besides the included papers listed above, the PhD candidate is also co-author of the following work, not part of this thesis:

- N. Paladi, C. Gehrmann, and F. Morenius, "Domain-Based Storage Protection (DBSP) in Public Infrastructure Clouds", in *Proc. 18th Nordic Conference on Secure IT Systems*, NordSec '13, pp. 279–296, Springer, October 2013
- A. Michalas, N. Paladi, and C. Gehrmann, "Security aspects of e-Health systems migration to the cloud", *in Proc. 16th International Conference on e-Health Networking, Applications and Services*, HealthCom '14, pp. 212–218, IEEE, October 2014
- N. Paladi, "Towards Secure SDN Policy Management", in *Proc. 8th International Conference on Utility and Cloud Computing*, UCC '15, pp. 607–611, December 2015
- C. Gehrmann, F. Morenius, and N. Paladi, "Procedure For Platform Enforced Storage in Infrastructure Clouds," March 2016. Patent Application WO2014185845 A1
- N. Paladi, L. Karlsson, "Safeguarding VNF Credentials with Intel SGX", in *Proc. of the SIGCOMM Posters and Demos*, ACM, 2017)

Part II

Included Papers

Paper A

Trusted Launch of Virtual Machine Instances in Public IaaS Environments

Nicolae Paladi, Christian Gehrmann, Mudassar Aslam and Fredric Morenius

Abstract

Cloud computing and Infrastructure-as-a-Service (IaaS) are emerging and promising technologies, however their adoption is hampered by data security concerns. At the same time, Trusted Computing is experiencing an increasing interest as a security mechanism for IaaS. In this paper we present a protocol to ensure the launch of a virtual machine instance on a trusted remote compute host. Relying on Trusted Platform Module operations such as *binding* and *sealing* to provide integrity guarantees for clients that require a trusted virtual machine launch, we have designed a trusted launch protocol for virtual machine instances in public IaaS environments. We also present a proof-of-concept implementation of the protocol based on OpenStack, an open-source IaaS platform. The results provide a basis for the use of TC mechanisms within IaaS platforms and pave the way for a wider applicability of trusted computing to IaaS security.

1 Introduction

One of the distinguished trends in IT operations today is the consolidation of IT systems onto common platforms. A key technology in realizing this is system virtualization [185]. System virtualization makes it possible to streamline IT operations, save energy and obtain better utilization of hardware resources. A virtualized computing infrastructure allows clients to run own services in form of virtual machine (VM) on shared computing resources. This approach

however introduces new challenges, as it means that information previously controlled by one administrative domain and organization is now under the control of a third party provider and that the information owner loses direct control over how data and services are used and protected. IaaS [102] is one of the business models based on system virtualization and security aspects are among the main identified obstacles for its adoption[1]. The problems with securing IaaS are evident not least through the fact that widely known platforms such as Amazon EC2, Microsoft Azure, services provided by RackSpace and other IaaS services are plagued by vulnerabilities at several levels of the software stack, from the web based cloud management console [18] to VM side-channel attacks, to information leakage, to collocation with malicious virtual machine instances [17].

A promising approach towards handling IaaS security threats and a mean to provide service confidence is the use of Trusted Computing technologies as defined by the Trusted Computing Group (TCG). The core component in the TCG-defined security architecture is the Trusted Platform Module (TPM), a hardware module that can be used as a trust anchor for software integrity verification in open platforms that also offers protected storage for sensitive parameters. TPM usage and deployment models for IaaS clouds are currently an active research area [180,186–190]. Earlier research has introduced principles of a trusted IaaS platform [180], later extended to cover both trusted virtual machine launch [189] and VM migration [190]. These research results demonstrate principles of combining basic TPM attestation mechanisms with standard cryptographic techniques to design an infrastructure for VM protection. However, such solutions have limitations with respect to security, complexity and target compute host selection procedures.

In this paper we describe a trusted virtual machine launch process that addresses these limitations and present a trusted launch protocol that does not require secure pre-packaging of the virtual machine image on the client side. Furthermore, in order to be usable in a significant proportion of IaaS deployment scenarios and to provide full scheduling flexibility on the IaaS side, the protocol allows the IaaS provider to select a target trusted compute host without directly involving the client. The main contributions of this paper are:

1. Description of a trusted launch protocol for virtual machine instances in public IaaS environments.
2. Implementation of the proposed protocol based on a widely-known IaaS platform.

The paper is further organized as follows: In Section 2 we define the trust and attack models, formulate the problem area and define requirements for a scheme to address the identified issues; section 3 presents the main contribution of the paper, namely a platform-agnostic protocol for trusted virtual machine launching. In Section 4 we perform a security analysis of the proposed protocol and continue with a description of the prototype implementation based on the OpenStack IaaS platform in Section 5. In Section 6 we provide summaries of significant related work within trusted computing in IaaS environments. We conclude in Section 7 and provide a set of further research suggestions.

[1] AFCEA Cyber Committee – October, 2011, `http://www.afcea.org/mission/intel/documents/cloudcomputingsecuritylessonslearned_final.pdf`

2 Trust and Attack Models, Problem Description and Requirements

Next we describe the trust and attack models we assume in this paper, list the top security and general design requirements applicable given the defined trust and attack models and revisit virtual machine images in the context of a trusted virtual machine instance launch. We also discuss the characteristics that can be expected from a well-designed virtual machine instance launch.

2.1 Trust and attack models

In the trust model and consequently the attack model used in this paper, the client does not implicitly trust any aspect of the IaaS provider. Although potentially true for many IaaS environment types, this trust model should be particularly relevant to public IaaS environments (according to the definition in [64]), where the relationship between the client and the IaaS provider is often formal and lacks prerequisites for implicit trust.

We share the attack model with [180, 189, 190] which assume that privileged access rights can be maliciously used by IaaS provider remote system administrators (A_r). This scenario assumes that A_r can log in remotely to any host maintained by the IaaS provider and obtain root access. However, in this model A_r does not have physical access to the hosts. The only possibility for A_r to circumvent this constraint is by succeeding to force a client to launch their virtual machine instance on an A_r-controlled compute host outside the physically secured IaaS provider perimeter. Furthermore, we assume that an A_r obtaining remote root access to the compute host will not be able to access the volatile memory of any VM residing on the compute host at that time, i.e. the compute host offers VMs a closed box execution environment[2]. This assumption is required in order to ensure that A_r can not access the nonce provided by C and its implementation is out of the scope of this paper.

In a trusted virtual machine launch context this means that we consider both the attack where A_r attempts to launch a virtual machine instance on a non-trusted compute host instead of on a trusted one and the attack where A_r attempts to substitute the virtual machine image requested by the client with a maliciously modified virtual machine image.

In the current attack model, a virtual machine instance is considered trusted if and only if it fulfils the following criteria:

1. The virtual machine image used for the instance is itself trusted;
2. The virtual machine instance is started on a trusted compute host;
3. The virtual machine instance has the client-generated verification token injected (see section 3.1)

[2]This does not include any VMs which are part of the hosting infrastructrure, such as Xen dom0 VM)

2.2 Virtual machine images

As an implication of the above trust and attack models, we consider the following two properties of virtual machines in the context of trusted computing:

- No virtual machine instance, or any entity communicating with the virtual machine instance, can determine whether the hypervisor the virtual machine instance is running on is trusted or not.
- A virtual machine instance cannot be trusted to reliably determine if it has the configuration originally requested by the client.

To overcome these issues, we suggest a launch protocol where we use standard TPM v1.2 functionality to first ensure that the client can detect the situation when it is communicating with a virtual machine instance that is not launched on a trusted platform and subsequently utilize the trusted platform to verify the integrity of the virtual machine image prior to virtual machine launch.

It is essential, in the scope of the protocol, that no modifications or customizations of the virtual machine image to be launched are performed by the IaaS provider without the client's knowledge.

2.3 Requirements for a trusted virtual machine launch protocol

Considering the trust and attack models above, it is important for the client to be able to obtain reasonable security guarantees from the IaaS provider. These include both trustworthiness of the computing resources, as well as guarantees regarding VM integrity and confidentiality. In order to also be cost and implementation efficient, the underlying infrastructure should provide such guarantees with a minimal operational overhead without increasing structural complexity. The expectations can be summarized as a set of basic requirements towards a trustworthy virtual machine launch process:

1. The client shall have the mechanisms to ensure that the virtual machine instance has been launched on a trusted compute host.
2. The client should have the possibility to reliably determine that it is communicating with a virtual machine instance launched on a trusted compute host, and not with a different virtual machine instance.
3. The integrity of the virtual machine image scheduled to be launched must be verifiable by the target trusted compute host.
4. The trusted virtual machine launch procedure should be scalable and have a minimum impact on the performance of the IaaS platform.
5. Clients should have a transparent view of the trusted launch procedure.

3 A Trusted Launch Protocol for VM Images in IaaS Environments

Based on the above requirements for a trusted launch protocol for virtual machine instances in IaaS environments, we present a platform-agnostic protocol that shows principles of using TPM functionality to ensure the integrity of the compute host and of the virtual machine image requested to be launched by the client. The below protocol addresses the security concerns presented above by focusing on simplicity, transparency, scalability and minimal interference with the currently known setup of the IaaS implementations. Furthermore, the protocol is based on widely-used and verified techniques, such as hashing and asymmetric cryptography in combination with TPM functionality.

The protocol requires the participation of four entities, three of which are typically involved in virtual machine launch procedures in IaaS architectures:

1. *Client* (C) is a IaaS user and intends to launch a virtual machine instance. In this paper, C is considered to be a *non-expert*, i.e. one not capable of assessing the security of platform configurations based on values contained in the measurement list. C requires a virtual machine instance to be launched on a trusted platform. Furthermore, it is important for C to be able to either verify or trust the security of virtual machine images provided for launch.
2. *Scheduler* (S) is responsible for receiving requests for virtual machine instance launches from C, as well as scheduling and rescheduling of virtual machine instances on available compute hosts at the IaaS provider. S should be able to function with minimal involvement in the security-specific message passing.
3. The *compute host* (CH) is the target resource that will be chosen by S to run the particular virtual machine instance. CH represents a physical or virtual server that is able to host one or more virtual machine instances (however, this paper considers exclusively the case when the CH is a physical server). For the purposes of the proposed protocol, a CH must also be equipped with a TCG-compliant TPM as well as be immune to modification attempts when in a trusted state.
4. The *Trusted third party (*TTP*)* is, as the name implies, trusted by both the *Client* and the *IaaS provider* and can not be controlled or manipulated by the IaaS provider. The recent breaches of Certificate Authorities have emphasized the drawbacks of centralized security models and their susceptibility to attacks [191]. The more complex the operations performed by the TTP, the higher the probability of it having exploitable vulnerabilities. It is therefore important to keep the implementation of the TTP as simple as possible. The main task of the TTP is to attest the configuration of the CH that will host the virtual machine instance and assess its security profile according to predefined policies. Within the current trust model, TTPs could be implemented by an *expert* C, as long as the IaaS provider agrees to that and C has the capability to set up and operate an attestation and evaluation engine.

For the purpose of the protocol, we also introduce the concept '*security profile* of a CH':

Definition 3.1. A security profile (SP) is a verified setup of an OS including underlying libraries and configuration files, which is considered to be *trusted* by all parties. SP can range on an ascending integer scale which reflects the level of verification, from least to most strict (and hence more restrictive).

The information needed to calculate the SP and also to compare the setup of two CHs is stored in the *integrity measurement log* (IML), as the IML contains hashes of the components that were loaded or used during the boot sequence of the CH. The validity of the IML is confirmed through a signature using the attestation identity keys (AIK) of a TPM. The AIK are persistent, non-migratable keys that are used to sign and authenticate by the means of an AIK certificate (denoted by $AIK - cert$) the validity of the information provided by the TPM in case of an external attestation [36]. We thus assume that the SP of any given CH can be deterministically calculated by each of the parties involved in the protocol. [3]

3.1 Platform-agnostic protocol description

The following steps are required in order to perform a trusted virtual machine launch (Fig. A.1, the steps of the protocol correspond to the steps in the figure [4]).

1. Before initiating the launch procedure, C generates a sufficiently long nonce N, to be used as a proof token in communications between C and the virtual machine instance and must be kept confidential to untrusted parties throughout the launch process.
2. C creates a token which we denote by T, representing a data structure with information necessary for the trusted virtual machine launch. T contains N, the minimum SP and the hash of the virtual machine image used for launch, denoted by H_{vmi}[5]. Finally, the token is encrypted with the public key of TTP, represented by PK_{TTP}, while the encrypted token is represented by $\mathsf{T}_{PK_{TTP}}$.
3. C provides the *scheduler (S)* the following parameters:
   - virtual machine image identifier and optionally the virtual machine image to be launched;
   - SP;
   - URL of the TTP;
   - Encrypted token $\mathsf{T}_{PK_{TTP}}$ generated in step **(2)**.

   SP will determine the lower bound of trust level required from CH on which the VM will run, with stricter security profiles accepted.
4. *S* schedules a VM on the appropriate CH, depending on its membership in the respective security profile group and sends the CH a request to generate a bind key PK_{Bind}, also providing the URL of the TTP.
5. Once the destination CH receives the bind key request, it retrieves a PCR-locked non-migratable TPM-based bind key PK_{Bind}. This key can be periodically regenerated by CH according to a administrator-defined policy, using the current platform state represented by the TPM PCRs. It is important to note that the values of the PCRs should not necessarily be in a trusted state in order to create a trusted state bind key.
6. In order to prove that the bind key is a non-migratable, PCR-locked, asymmetric TPM key, CH uses the `TPM_CERTIFY_KEY` TPM command in order to retrieve the `TPM_CERTIFY_INFO`

[3] The methodology for calculating the SP of a CH is out of the scope of this paper.

[4] Due to space limitations, "Attestation data" was chosen as the condensed notation for: $\mathsf{T}_{PK_{TTP}}, PK_{Bind}, \texttt{TPM\_CERTIFY\_INFO}, H_{\texttt{TPM\_CERTIFY\_INFO}}{}^{AIK}, IML, AIK - cert$

[5] If non-repudiation of virtual machine launch is required, the client should also sign the virtual machine image hash and include the signature and corresponding client certificate into the token.

structure signed with the TPM attestation indentity key [36], which we denote by PK_{AIK}; we also denote the signed structure by
$H_{\texttt{TPM\_CERTIFY\_INFO}}{}^{AIK}$. The `TPM_CERTIFY_INFO` data structure contains the hash of the bind key and the PCR value required for the key usage.

7. CH sends an attestation request to the TTP through an HTTPS session using the URL supplied by C. The following arguments are sent in the request to TTP:
   - Client-provided token $\mathsf{T}_{PK_{TTP}}$
   - *Attestation data*, which includes the public bind key, the `TPM_CERTIFY_INFO` structure, the hash of `TPM_CERTIFY_INFO` signed with the AIK[6], the IML and the AIK-certificate collectively represented as:
     PK_{Bind}, `TPM_CERTIFY_INFO`, $H_{\texttt{TPM\_CERTIFY\_INFO}}{}^{AIK}$, IML, AIK-cert.
8. TTP uses its private key PrK_{TTP}, which corresponds to the public PK_{TTP} to attempt to decrypt the token $\mathsf{T}_{PK_{TTP}}$.
9. TTP validates the attestation information obtained from CH as follows:
   - Validates the AIK certificate;
   - Validates the structure of the AIK-signed `TPM_CERTIFY_INFO`;
   - Validates the key PK_{Bind} by comparing its digest with the digest received in `TPM_CERTIFY_INFO`;
   - Calculates the hash of the PCR values H_{PCR} based on the information in the IML and compares it with the hash of `PCR_INFO`,
     which is a component of `TPM_CERTIFY_INFO`
10. TTP examines the entries in the IML in order to determine the trustworthiness of the CH and decides whether SP is satisfied.
11. If step **10** is true, TTP encrypts N and the hash H_{vmi} with the bind key PK_{Bind} obtained from CH, to ensure that N is only available to CH in a trusted state. By sending N and H_{vmi} encrypted with the public key PK_{Bind} available to the trusted configuration of CH, the security perimeter expands to include three parties: C itself, TTP and CH in its trusted configuration. This implies that all actions performed by CH in its trusted configuration are trusted by default.
12. Prior to launching the VM, CH decrypts N and H_{vmi} using the TPM-issued PrK_{Bind}, which is available to it in its trusted configuration but stored in the TPM; next, CH compares H_{vmi} obtained from the TTP with the hash of the virtual machine image to be used for launch and accepts the image only in case the values are equal.
13. CH injects N into the virtual machine image prior to launching the virtual machine instance.
14. CH returns an acknowledgement to S to confirm a successful launch.
15. To verify that the virtual machine instance has been launched on a trusted platform, C challenges the virtual machine instance to prove its knowledge of N.

The fact that N is kept confidential allows it to be used as an authentication token while establishing a secure communication channel between C and the launched *VM* instance. N can be used as the pre-shared secret in order to add protection against man-in-the-middle

[6]Expressed as $H_{\texttt{TPM\_CERTIFY\_INFO}}{}^{AIK}$

attacks when using Diffie-Hellman key exchange, as specified in the password-authenticated key-exchange protocol [192].

Some of the operations can be optimized taking into account the operational environment. For example, the validity period of PK_{Bind} created in step **(5)** can be adjusted. In a similar way, TTP can have a cache of the PK_{Bind} keys created by CHs with verified trusted configurations. In this case, steps **(9)** and **(10)** can be skipped for a certain number of cases, which can also be regulated by an administrative policy. However, it is important to remember that the use of such a cache introduces further complexity to TTP, the analysis of which is out of the scope of this paper.

4 Protocol Security Analysis

In this section we present a critical review of the protocol and highlight improvement areas that were left as future work. We begin with a security analysis of the protocol, in order to outline its strengths and weaknesses.

Returning to the security concerns expressed in the requirements on the trusted launch protocol formulated in Section 2.3, they are addressed as follows. Let φ be the guest virtual machine instance launched on CH, then:

- R1: Following above protocol, C and φ have a shared secret N. The fact that φ is running on a *trusted platform* is ensured by the properties of the bind key used to seal the shared secret N to the trusted configuration of CH;
- R2: The fact that C is communicating with φ and not any other unexpected virtual machine instance φ' is ensured through the combination of: **a.** verification of CH by the TTP, **b.** presence of the token N injected into φ where N is only available to CH in a trusted state; **c.** the virtual machine image integrity verification performed by the CH prior to the launch. A failure at any of the steps of the above sequence would prevent the trusted virtual machine launch, a fact that would be verifiable by C.
- R3: Integrity of the virtual machine image is ensured through the verification performed by CH in a trusted state, prior to the trusted virtual machine launch. Thus, the virtual machine image is verified using the hash value obtained from the TTP. By comparing the hash of the virtual machine image with the expected H_{vmi} provided by C, CH ensures a one-to-one correspondence between the virtual machine image to be used for launch and the virtual machine image expected by C. The chain is completed once C verifies the presence of N injected into φ. The presence of the correct token N guarantees the integrity of φ requested by C.
- R4: Scalability of the protocol is ensured by the lightweight nature of operations that must be performed by both TTP and CH and the flexibility in the choice of TTP. While a challenging topic, especially in the case of high-availability and heavy load IaaS setups, the design of a scalable TTP architecture is out of the scope of this paper.
- R5: Transparency of the trusted virtual machine launch procedure is ensured by the introduction of client parameters, such as the URL of the TTP, the trust level of CH and the secret token generated by C. The ability to choose TTP opens the possibility for C to ensure the trustworthiness of the CH attestation procedure, either through audit controls of the TTP or by itself serving the role of TTP.

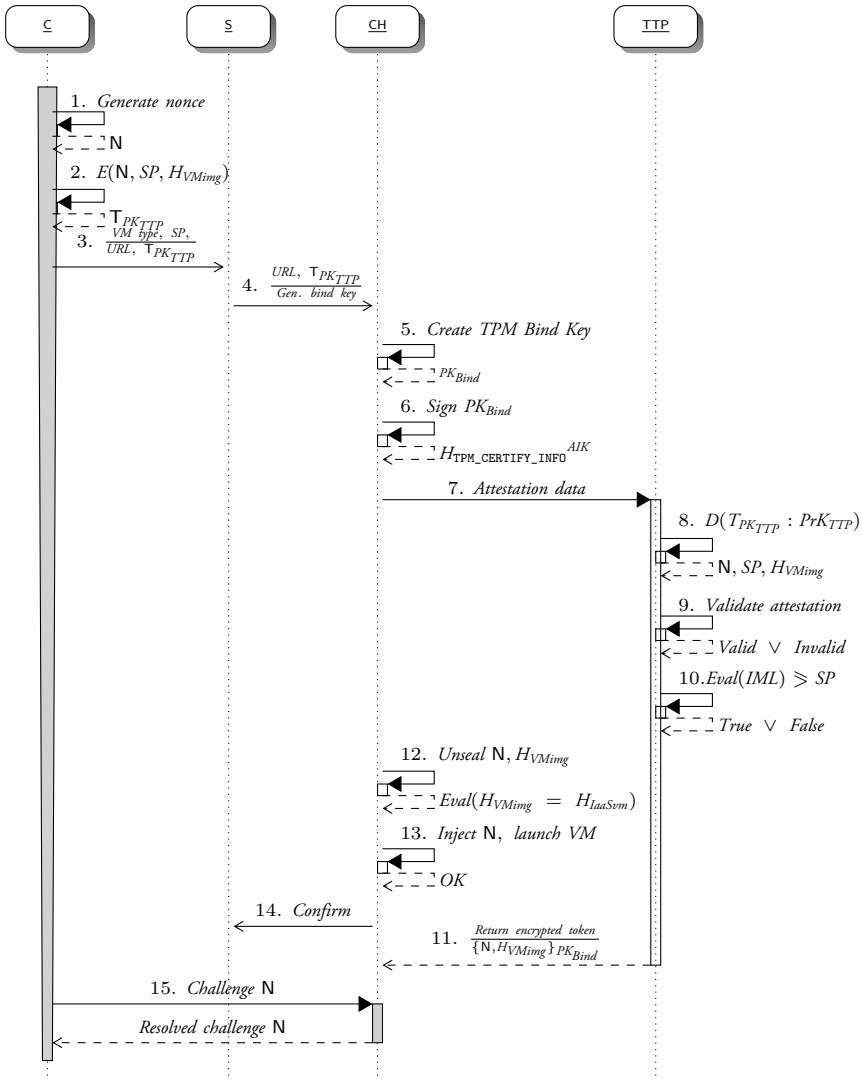


Figure A.1: Trusted virtual machine launch protocol: C: Client; S: Scheduler; CH: Compute Host; TTP: Trusted Third Party;

4.1 TTP verification model

The stateless architecture of the TTP implies that it does not maintain knowledge of N except for at the moment of sealing it to CH and does not maintain any session state at any point of the protocol. As a result, an A_r can only obtain N from TTP if they obtain TTP's private key PrK_{TTP}. Furthermore, assessment of the trust level of a CH according to a deterministic

algorithm which only takes two inputs (in the form of static set of reference measurement data and dynamic attestation calls from any CH) will be easily traceable and reproducible based on the original input data, without the need to recreate or rely on a certain state of the TPP's internal data. Finally, a stateless architecture of the TTP contributes indirectly towards requirement R4.

4.2 Protocol caveats

One aspect that requires more attention is the possibility of a post-launch modification of the software stack of CH. The runtime process infection method, which is a method for infecting binaries during runtime[7] is one of the malicious approaches that could be used to this end. This scenario is in fact a common threat to all TCG-based systems, also touched upon in [193], described in detail in [194] and should thus be prevented using means within the platform which is part of the trusted computing base verified at boot time, the presence of which is verified by the above protocol.

5 Protocol Implementation

In order to validate the assumptions made during the protocol design phase, we have implemented it as an extension to OpenStack, an open source IaaS platform chosen given the open access to its codebase, its large community and the traction it has gained. This section briefly introduces the OpenStack architectural model and changes made for the prototype implementation.

5.1 OpenStack IaaS platform

The Essex release of OpenStack comprises five core components (projects), namely Compute (Nova), Image Service (Glance), Object Storage (Swift), Identity Service (Keystone) and Dashboard (Horizon). Nova has several sub-components: nova-api, nova-compute, nova-schedule, nova-network, nova-volume, plus an SQL database and message queue functionality to pass messages between sub-components. OpenStack components affected by the protocol implementation are mentioned here in more detail:

- Nova-api is the interface for nova- compute and volume API calls. It is through this interface most of the cloud orchestration operations are performed. The interface supports both the OpenStack and Amazon EC2 API.
- Nova-compute handles virtual machine instance life cycle tasks through hypervisor API calls. Notably the libvirt and XenAPI hypervisor APIs are supported.
- Nova-schedule is responsible for selecting CH(s) to run virtual machine instances on. The CH selection process is determined by which scheduling policy/algorithm is employed.
- The nova SQL database holds tables and relations to describe the state of nova, such as launched instances and network configurations.

[7] Runtime process infection, `http://www.phrack.org/issues.html?issue=59&id=8&mode=txt`

- The Dashboard is a web based GUI for OpenStack operation and administration. It interfaces nova-api.

5.2 Prototype Implementation

Below are the main additions to OpenStack required for the prototype implementation.

Nova SQL database The nova SQL database has been extended to include tables to hold the available CHs and their SPs:

- An SP is an integer in the range 1-10, with a higher number being more trusted than a lower number.
- The security profile of a CH is global, rather than specific per tenant.

Dashboard and nova-api The Dashboard web based GUI has been extended to include the option to request CH attestation, minimum SP selection, token $\mathsf{T}_{PK_{TTP}}$ entry and TTP URL provision **(3)** into the *"Launch Instance"* dialog. This information is included in the OpenStack API HTTP payload to nova-api, which propagates the information to the scheduler.

In the prototype implementation, steps **(1)** and **(2)** are performed by a script which outputs $\mathsf{T}_{PK_{TTP}}$, which then can be manually input into the Dashboard dialog. Note that it is not an option to let Dashboard provide functionality for generating $\mathsf{T}_{PK_{TTP}}$, since Dashboard is not trusted by C.

Scheduler, compute host and virtualization driver The nova scheduler S is a central component as it decides on which CH a certain virtual machine instance will be launched. Each S works according to a specific configurable algorithm and several S implementations are available in OpenStack by default. In the SimpleScheduler implementation, S identifies the least loaded CH and schedules the virtual machine instance to be launched on that CH.

We extend the behaviour of the SimpleScheduler to include the policy that a CH must belong to a certain SP or stricter in order to be acceptable for hosting the virtual machine instance. This policy is realized as follows: first S looks up the recorded SP of CH in the nova database and proceeds if SP is sufficient compared to the requirements of C (corresponds to **(4)**). The second step is to request CH to attest itself with TTP. If SP was not sufficient, the next eligible CH is selected.

Steps **(5)**-**(7)** are perfomed by CH, followed by TTP in steps **(8)**-**(11)**. Token $T_{CH} = \{\mathsf{N}, H_{vmi}\}_{PK_{Bind}}$ is returned from TTP to CH after which CH includes the token in the return message to S . If the attestation was successful, S requests the now trusted CH to launch the virtual machine instance and includes T_{CH} in the request.

Next, CH decrypts T_{CH} and obtains N and H_{vmi}. To verify the integrity of the virtual machine image, H_{vmi} is included in the call to the virtualization driver (`libvirt` is used by the prototype), which fetches the virtual machine image from Glance and caches it locally on CH. The hash of the cached image is calculated and compared to H_{vmi}. If the hashes do not match, an exception is raised. Otherwise, the launch procedure continues **(12)** and the file injection capability of

Nova is used to inject N into the file system of the virtual machine image **(13)**. The virtual machine image is then used to launch the virtual machine instance on CH and steps **(14)** and **(15)** are completed.

6 Related Work

Application of trusted computing principles within IaaS environments has been the focus of several research papers examined below.

Santos et al propose the design of a "trusted cloud compute platform" (TCCP) that ensures VMs are running on a trusted hardware and software stack with a remote and initially untrusted CH[180]. The authors propose a remote attestation process where a trusted coordinator (TC) stores the list of attested CHs that run a "trusted virtual machine monitor" which can securely run the client's VM. A trusted CH maintains in its memory an individual *trusted key* used for identification each time the client C instantiates a VM on the trusted CH. The paper presents a good initial set of ideas for trusted virtual machine launch and migration, in particular the use of a TC. A limitation of this solution is that the trusted key resides in the memory of the trusted CH, which leaves the solution vulnerable to cold boot attacks [195] with keys extractable from memory. Furthermore, the authors require that the TC maintains information about all CH deployed on the IaaS platform, but do not mention mechanisms for anonymizing this information, making it valuable to an attacker and unacceptable for a public IaaS provider. Finally, the solution lacks both mechanisms for revocation of the trusted key and considerations for the re-generation of trusted key outside of CH reboot.

A decentralized approach to integrity attestation is adopted by Schiffman et al in [196]. The primary concerns addressed by this approach are the limited transparency of IaaS platforms and the limits to scalability imposed by third party integrity attestation mechanisms, as described in [180]. The authors examine a trusted cloud architecture where the integrity of the IaaS CH is verified by the IaaS client through a "cloud verifier" (CV) proxy that resides in the application domain of the IaaS platform provider and is accessible by the client. Thus, in the first step of the protocol the client evaluates the integrity of the CV in order to include the CV into its trust perimeter if the integrity level of the CV is considered satisfactory. Next, the CV sends attestation requests from CH, i.e. the CH where the guest virtual machine instance can potentially be deployed, thus extending the trust chain to the CH. Finally, CH verifies the integrity of the virtual machine image, which is countersigned by the CV and returned to the client which evaluates the virtual machine image integrity data and allows or disallows the virtual machine launch on the CH. While the idea of increasing the transparency of the IaaS platform for the client is indeed supported in industry [197, 198], the authors do not clarify how the introduction of an additional integrity attestation component in the architecture of the IaaS platform has positive effects on the transparency of the IaaS platform. Furthermore, the proposed protocol increases the complexity model for the IaaS client both by introducing the evaluation of integrity attestation reports of the CV and CH and introduction of additional steps in the trusted virtual machine launch, where the client has to take actions based on the data returned from the CV. This requires either human interaction or a fairly complex integrity attestation evaluation component (or a combination thereof) on the client side, making a wide-scale adoption of the solution difficult.

In [189], Aslam et al proposed principles for trusted virtual machine launch on public cloud platforms using trusted computing techniques. In order to ensure that the requested virtual

machine instance is launched on a CH with verifiable integrity, the client encrypts the virtual machine image (along with all injected data) with a symmetric key sealed to a particular configuration of CH, which is reflected through the values in the platform configuration registers (PCR) of the TPM deployed on the CH. The solution proposed by Aslam et al presents a suitable model in the case of trusted virtual machine launch scenarios for enterprise clients. It requires that the virtual machine image is pre-packaged and encrypted by C prior to IaaS launch. However the proposed model does not cover the very common scenario of launching an unmodified virtual machine image made available by the IaaS provider or uploaded by C. Furthermore, we believe that reducing the number of steps required from C will facilitate the adoption of the trusted IaaS model. Likewise, direct communication between C and CH, as well as significant changes to the existing virtual machine launch implementations in IaaS platforms hamper the implementation of this protocol. This paper reuses some of the ideas proposed in [189] and directly addresses the above limitations, namely actions to be performed by C, also touching upon the requirements towards the launched virtual machine instance and required changes to the IaaS platform.

7 Conclusion

In this paper we have presented a detailed trusted launch protocol for virtual machine instance launch in public IaaS environments. Furthermore, we have provided a prototype implementation of the launch protocol in OpenStack. Detailed performance measurement and evaluation, as well as alternative implementation choices have been left for future work.

The presented results make a case for broadening the range of use cases for trusted computing by applying it to IaaS environments, especially within the security model of an untrusted IaaS provider. Trusted computing offers capabilities to securely perform data manipulations on remote hardware owned and maintained by another party by potentially preventing the use of untrusted software on that hardware for such manipulations. The presented design is directly applicable to the process of developing a trusted virtualized environment, such as a public IaaS service.

Future research recommendations can be grouped into three categories: *First* is the extension of the trust chain to other operations on virtual machine instances (migration, suspension, updates, etc.), as well as data storage and virtual network communication security. The *second* category includes addressing certain assumptions of the proposed launch protocol, such as the assumption that the CH configuration is not changed after the trusted launch of the virtual machine instance, since even in the case of a bona fide IaaS provider the CH can be compromised through runtime process infection. A technique to enable C to either directly or through mediated access discover such events and protect the data used by the virtual machine instance is a promising research topic. The *third* category focuses on the design and implementation of the evaluation policies of the TTP. The current assumption is that the TTP has access to information regarding "secure" configurations and the PCR values, something which needs to be directly addressed as evaluating exactly how secure a certain software stack is a challenge. Furthermore, taking into account the diversity of available libraries as well as the different combinations in which they can be loaded during the boot process, verification of PCR values (such as values stored in `PCR10` and reference values in `binary_runtime_measurements`) becomes a less trivial task.

Paper B

Providing User Security Guarantees in Public Infrastructure Clouds

Nicolae Paladi, Christian Gehrmann and Antonis Michalas

Abstract

The infrastructure cloud (IaaS) service model offers improved resource flexibility and availability, where tenants – insulated from the minutiae of hardware maintenance – rent computing resources to deploy and operate complex systems. Large-scale services running on IaaS platforms demonstrate the viability of this model; nevertheless, many organizations operating on sensitive data avoid migrating operations to IaaS platforms due to security concerns. In this paper, we describe a framework for data and operation security in IaaS, consisting of protocols for a trusted launch of virtual machines and domain-based storage protection. We continue with an extensive theoretical analysis with proofs about protocol resistance against attacks in the defined threat model. The protocols allow trust to be established by remotely attesting host platform configuration prior to launching guest virtual machines and ensure confidentiality of data in remote storage, with encryption keys maintained outside of the IaaS domain. Presented experimental results demonstrate the validity and efficiency of the proposed protocols. The framework prototype was implemented on a test bed operating a public electronic health record system, showing that the proposed protocols can be integrated into existing cloud environments.

1 Introduction

Cloud computing has progressed from a bold vision to massive deployments in various application domains. However, the complexity of technology underlying cloud computing introduces novel security risks and challenges. Threats and mitigation techniques for the IaaS model have been under intensive scrutiny in recent years [21, 180, 196, 199], while the industry has invested in enhanced security solutions and issued best practice recommendations [200]. From an end-user point of view the security of cloud infrastructure implies unquestionable trust in the cloud provider, in some cases corroborated by reports of external auditors. While providers may offer security enhancements such as protection of data at rest, end-users have limited or no control over such mechanisms. There is a clear need for usable and cost-effective cloud platform security mechanisms suitable for organizations that rely on cloud infrastructure.

One such mechanism is platform integrity verification for compute hosts that support the virtualized cloud infrastructure. Several large cloud vendors have signaled practical implementations of this mechanism, primarily to protect the cloud infrastructure from insider threats and advanced persistent threats. We see two major improvement vectors regarding these implementations. First, details of such proprietary solutions are not disclosed and can thus not be implemented and improved by other cloud platforms. Second, to the best of our knowledge, none of the solutions provides cloud tenants a proof regarding the integrity of compute hosts supporting *their* slice of the cloud infrastructure. To address this, we propose a set of protocols for trusted launch of virtual machines (VM) in IaaS, which provide tenants with a proof that the requested VM instances were launched on a host with an expected software stack.

Another relevant security mechanism is encryption of virtual disk volumes, implemented and enforced at compute host level. While support for data encryption at rest is offered by several cloud providers and can be configured by tenants in their VM instances, functionality and migration capabilities of such solutions are severely restricted. In most cases cloud providers maintain and manage the keys necessary for encryption and decryption of data at rest. This further convolutes the already complex data migration procedure between different cloud providers, disadvantaging tenants through a new variation of vendor lock-in. Tenants can choose to encrypt data on the operating system (OS) level within their VM environments and manage the encryption keys themselves. However, this approach suffers from several drawbacks: first, the underlying compute host will still have access encryption keys whenever the VM performs cryptographic operations; second, this shifts towards the tenant the burden of maintaining the encryption software in all their VM instances and increases the attack surface; third, this requires injecting, migrating and later securely withdrawing encryption keys to each of the VM instances with access to the encrypted data, increasing the probability than an attacker eventually obtains the keys. In this paper we present DBSP (domain-based storage protection), a virtual disk encryption mechanism where encryption of data is done directly on the compute host, while the key material necessary for re-generating encryption keys is stored in the volume metadata. This approach allows easy migration of encrypted data volumes and withdraws the control of the cloud provider over disk encryption keys. In addition, DBSP significantly reduces the risk of exposing encryption keys and keeps a low maintenance overhead for the tenant – in the same time providing additional control over the choice of the compute host based on its software stack.

We focus on the Infrastructure-as-a-Service model – in a simplified form, it exposes to its tenants a coherent platform supported by *compute hosts* which operate VM guests that communicate through a virtual network. The system model chosen for this paper is based on

requirements identified while migrating a currently deployed, distributed electronic health record (EHR) system to an IaaS platform [201].

1.1 Contribution

We extend previous work applying Trusted Computing to strengthen IaaS security, allowing tenants to place hard security requirements on the infrastructure and maintain exclusive control of the security critical assets. We propose a security framework consisting of three buiding blocks:

- Protocols for trusted launch of VM instances in IaaS;
- Key management and encryption enforcement functions for VMs, providing transparent encryption of persistent data storage in the cloud;
- Key management and security policy enforcement by a Trusted Third Party (TTP);

We describe several contributions that enhance cloud infrastructure with additional security mechanisms:

1. We describe a trusted VM launch (TL) protocol which allows tenants – referred to as *domain managers* – to launch VM instances exclusively on hosts with an attested platform configuration and reliably verify this.
2. We introduce a domain-based storage protection protocol to allow domain managers store encrypted data volumes partitioned according to administrative *domains*.
3. We introduce a list of attacks applicable to IaaS environments and use them to develop protocols with desired security properties, perform their security analysis and prove their resistance against the attacks.
4. We describe the implementation of the proposed protocols on an open-source cloud platform and present extensive experimental results that demonstrate their practicality and efficiency.

1.2 Organization

The rest of this paper is organized as follows. In Section 2 we describe relevant related work on *trusted virtual machine launch* and *cloud storage protection*. In Section 3 we introduce the system model, as well as the threat model and problem statement. In Section 4 we introduce the protocol components, and the TL and DBSP protocols as formal constructions. In Section 5, we provide a security analysis and prove the resistance of the protocols against the defined attacks, while implementation and performance evaluation results are described in Section 6. We discuss the protocol application domain in Section 7 and conclude in Section 8.

2 Related Work

We start with a review of related work on trusted VM launch, followed by storage protection in IaaS.

2.1 Trusted Launch

Santos et al. [180] proposed a "Trusted Cloud Compute Platform" (TCCP) to ensure VMs are running on a trusted hardware and software stack on a remote and initially untrusted host. To enable this, a trusted coordinator stores the list of attested hosts that run a "trusted virtual machine monitor" which can securely run the client's VM. Trusted hosts maintain in memory an individual *trusted key* used for identification each time a client launches a VM. The paper presents a good initial set of ideas for trusted VM launch and migration, in particular the use of a *trusted coordinator*. A limitation of this solution is that the trusted coordinator maintains information about all hosts deployed on the IaaS platform, making it a valuable target to an adversary who attempts to expose the public IaaS provider to privacy attacks.

A decentralized approach to integrity attestation is adopted by Schiffman et al. [196] to address the limited transparency of IaaS platforms and scalability limits imposed by third party integrity attestation mechanisms. The authors describe a trusted architecture where tenants verify the integrity of IaaS hosts through a trusted *cloud verifier* proxy placed in the cloud provider domain. Tenants evaluate the cloud verifier integrity, which in turn attests the hosts. Once the VM image has been verified by the host and countersigned by the cloud verifier, the tenant can allow the launch. The protocol increases the complexity for tenants both by introducing the evaluation of integrity attestation reports of the cloud verifier and host and by adding steps to the trusted VM launch, where the tenant must act based on the data returned from the cloud verifier. Our protocol maintains the VM launch traceability and transparency without relying on a proxy verifier residing in the IaaS. Furthermore, the TL protocol does not require additional tenant interaction to launch the VM on a trusted host, beyond the initial launch arguments.

Platform attestation prior to VM launch is also applied in [202], which introduces two protocols – "TPM-based certification of a Remote Resource" (TCRR) and "VerifyMyVM". With TCRR a tenant can verify the integrity of a remote host and establish a trusted channel for further communication. In "VerifyMyVM", the hypervisor running on an attested host uses an emulated TPM to verify on-demand the integrity of running VMs. Our approach is in many aspects similar to the one in [202] in particular with regard to host attestation prior to VM instance launch. However, the approach in [202] requires the user to *always* encrypt the VM image before instantiation, thus complicating image management. This prevents tenants from using commodity VM images offered by the cloud provider for trusted VM launches. We overcome this limitation and generalize the solution by adding a verification token, created by the tenant and injected on the file system of the VM instance *only* if it is launched on an attested cloud host.

In [189], the authors described a protocol for trusted VM launch on public IaaS using trusted computing techniques. To ensure that the requested VM instance is launched on a host with attested integrity, the tenant encrypts the VM image (along with all injected data) with a symmetric key sealed to the host configuration, reflected in the platform configuration registers (PCR) values of the host TPM. The proposed solution is suitable in trusted VM launch scenarios for enterprise tenants as it requires pre-packaging and encrypting the VM image prior to IaaS launch. However, similar to [202], this prevents tenants from using commodity VM images offered by the cloud provider to launch VM instances on trusted cloud hosts. Furthermore, we believe that reducing the number of steps required from the tenant can facilitate the adoption of the trusted IaaS model. We extend some of the ideas proposed in [189], address the above limitations – such as additional actions required from tenants – and also address the requirements towards the launched VM instance and required changes to cloud platforms.

2.2 Secure Storage

Cooper at al described in in [203] a secure platform architecture based on a secure root of trust for grid environments – precursors of cloud computing. Trusted Computing is used as a method for dynamic trust establishment within the grid, allowing clients to verify that their data wil be protected against malicious host attacks. The authors address the malicious host problem in grid environments, with three main risk factors: trust establishment, code isolation and grid middleware. The solution established a minimal trusted computing base (TCB) by introducing a security manager isolated by the hypervisor from grid services (which are in turn performed within VM instances). The secure architecture is supported by protocols for data integrity protection, confidentiality protection and grid job attestation. In turn, these rely on client attestation of the host running the respective jobs, followed by interaction with the security manager to fulfill the goals of the respective protocols. We follow a similar approach in terms of interacting with a minimal TCB for protocol purposes following host attestation. However, in order to adapt to the cloud computing model we delegate the task of host attestation to an external TTP as well as use TPM functionality to ensure that sensitive cryptographic material can only be accessed on a particular attested host.

In [204], the authors proposed an approach to protect access to outsourced data in an owner-write-users-read case, assuming an "honest but curious service provider". Encryption is done over (abstract) blocks of data, with a different key per block. The authors suggest a key derivation hierarchy based on a public hash function, using the hash function result as the encryption key. The scheme allows to selectively grant data access, uses over-encryption to revoke access rights and supports block deletion, update, insertion and appending. It adopts a lazy revocation model, allowing to indefinitely maintain access to data reachable prior to revocation (regardless of whether it has been accessed before access revocation). While this solution is similar to our model with regard to information blocks and encryption with different symmetric keys, we propose an active revocation model, where the keys are cached for a limited time and cannot be retrieved once the access is revoked.

The "Data-Protection-as-a-Service" (DPaaS) platform [205] balances the requirements for confidentiality and privacy with usability, availability and maintainability. DPaaS focuses on shareable logical data units, confined in isolated partitions (e.g. VMs of language-based features such as Caja, Javascript) or containers, called Secure Execution Environments (SEE). Data units are encrypted with symmetric keys and can be stored on untrusted hardware, while containers communicate through authenticated channels. The authors stress the verifiability of DPaaS using trusted computing and the use of the dynamic root of trust to guarantee that computation is performed on a "secure" platform. The authors posit that DPaaS fulfills confidentiality and privacy requirements and facilitates maintenance, logging and audit; provider migration is one of the aspects highlighted, but not addressed in [205]. Our solution resembles DPaaS in the use of SEE based on software attestation mechanisms offered by the TPM, and in the reliance on full disk encryption to protect data at rest and support for flexible access control management of the data blocks. However, the architecture outlined in [205] does not address bootstrapping the platform (e.g. the VM launch) and provides few details about the key management mechanism for the secure data store. We address the above shortcomings, by describing in detail and evaluating protocols to create and share confidentiality-protected data blocks. We describe cloud storage security mechanisms that allow easy data migration between providers without affecting its confidentiality.

Graf et al. [206] presented an IaaS storage protection scheme addressing access control. The authors analyse access rights management of shared versioned encrypted data on cloud infra-

structure for a restricted group and propose a scalable and flexible key management scheme. Access rights are represented as a graph, making a distinction between data encryption keys and encrypted updates on the keys and enabling flexible join/leave client operations, similar to properties presented by the protocols in this paper. Despite its advantages, the requirement for client-side encryption limits the applicability of the scheme in [206] and introduces important functional limitations on indexing and search. In our model, all cryptographic operations are performed on trusted IaaS compute hosts, which are able to allocate more computational resources than client devices.

Santos et al. [129] proposed Excalibur, a system using trusted computing mechanisms to allow decrypting client data exclusively on nodes that satisfy a tenant-specified policy. Excalibur introduces a new trusted computing abstraction, *policy-sealed data* to address the fact that TPM abstractions are designed to protect data and secrets on a standalone machine, at the same time over-exposing the cloud infrastructure by revealing the identity and software fingerprint of individual cloud hosts. The authors extended TCCP [180] to address the limitations of binary-based attestation and data sealing by using property-based attestation [207]. The core of Excalibur is '*the monitor*', which is a part of the cloud provider, which organises computations across a series of hosts and provides guarantees to tenants. Tenants first decide a policy and receive evidence regarding the status of the monitor along with a public encryption key, and then encrypt their data and policy using ciphertext-policy attribute-based encryption [128]. To decrypt, the stored data hosts receive the decryption key from the monitor who ensures that the corresponding host has a valid status and satisfies the policy specified by the client at encryption time. Our solution is similar to the one in [129], with some important differences: *1)* In contrast with [129] our protocols were implemented as a code extension for Openstack. Furthermore, the presented measurements were made after we deployed the protocols for a part of the Swedish electronic health records management system in an infrastructure cloud. Thus, our measures are considered as realistic since the experiments were done under a real electronic healthcare system; *2)* Excalibur does not present a security analysis, substituted instead by the results of ProVerif (an automated tool) regarding the correctness of their protocol. Furthermore, we introduced a new list of attacks that can be applied to such systems, which can be considered as a contribution to protocol designers since can avoid common pitfalls and design even better protocols in the future.

In [122] the authors presented a cryptographic cloud storage built on an untrusted IaaS infrastructure. The approach aims to provide confidentiality and integrity, while retaining the benefits of cloud storage – availability, reliability, efficient retrieval and data sharing – and ensuring security through cryptographic guarantees rather than administrative controls. The solution requires four client-side components: *data processor, data verifier, credential generator, token generator*. Some important building blocks are: *Symmetric searchable encryption (SSE)*, appropriate in settings where the data consumer is also the one who generates it (efficient for single writer-single reader (SWSR) models); *Asymmetric searchable encryption (ASE)*, appropriate for many writer single reader (MWSR) models, offers weaker security guarantees as the server can mount a dictionary attack against the token and learn the search terms of the client; *Efficient ASE*, appropriate in MWSR scenarios where the search terms are hard to guess, offers efficient search but is vulnerable to dictionary attacks; *Multi-user SSE*, appropriate for single writer/many reader settings, allows the owner to – besides encrypting indexes and generating tokens – revoke user search privileges over data; *Attribute based encryption*, introduced in [208], provides users with a decryption key with certain associated attributes, such that a message can be encrypted using a certain key and a policy. In such a scheme, the message can only be decrypted only if the policy matches the key used to encrypt it; finally, *proofs of storage* allow a client to verify that data integrity has not been violated by the server.

The concepts presented in [122] are promising – especially considering recent progress in searchable encryption schemes [209]. Indeed, integrating searchable and attribute-based encryption mechanisms into secure storage solutions is an important direction in our future work. However, practical application of searchable encryption and attribute-based encryption requires additional research.

Earlier work in [131, 132] described interoperable solutions towards trusted VM launch and storage protection in IaaS. We extend them to create an *integrated* framework that builds a trust chain from the domain manager to the VM instances and data in their administrative domain, and provide additional details, proofs and performance evaluation.

3 System Model and Preliminaries

In this section we describe the system and threat model, as well as present the problem statement.

3.1 System Model

We assume an IaaS system model (e.g. OpenStack, a popular open-source cloud platform) as in [64]: providers expose a *quota* of network, computation and storage resources to its tenants – referred to as *domain managers* (Figure B.1). Domain managers utilize the quota to launch and operate *VM guests*. Let $\mathcal{DM} = \{DM_1, \ldots, DM_n\}$ be the set of all domain managers in our IaaS. Then, $\mathcal{VM}_i = \left\{vm_1^i, \ldots, vm_n^i\right\}$ is the set of all VMs owned by each domain manager DM_i. VM guests operated by $\mathcal{DM}$ are grouped into *domains* (similar to *projects* in OpenStack) which comprise cloud resources corresponding to a particular organization or administrative unit. $\mathcal{DM}$ create, modify, destroy domains and manage access permissions of VMs to data stored in the domains. We refer to $\mathcal{D}_i = \left\{D_1^i, \ldots, D_n^i\right\}$ as the set of all domains created by a domain manager DM_i.

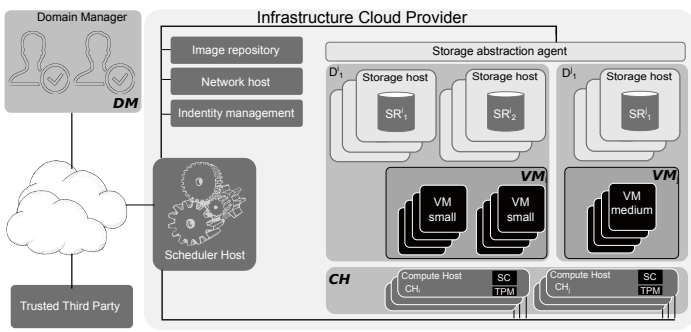


Figure B.1: High level view of the IaaS model introduced in Section 3.

Requests for operations on VMs (launch, migration, termination, etc.) received by the IaaS are managed by a *scheduler* that allocates (reallocates, deallocates) resources from the pool of available *compute hosts* according to a resource management algorithm. We assume in this work compute hosts that are *physical* – rather than virtual – servers. We denote the set of all

compute hosts as $\mathcal{CH} = \{CH_1, \ldots, CH_n\}$. We denote a VM instance vm_l^i running on a compute host CH_i by $vm_l^i \mapsto CH_i$ and its unique identifier by $idvm_l^i$.

The *Security Profile* (*SP*), defined in [131], is a function of the verified and measured deployment of a *trusted computing base* – a collection of software components measurable during a platform boot. Measurements are maintained in protected storage, usually located on the same platform. We expand this concept in Section 4. Several functionally equivalent configurations may each have a different security profile. We denote the set of all compute hosts that share the same security profile SP_i as $\mathcal{CH}_{SP_i}$. VMs intercommunicate through a virtual network overlay, a "software defined network" (SDN). A domain manager can create arbitrary network topologies in the same domain to interconnect the VMs without affecting network topologies in other domains.

I/O virtualization enables device aggregation and allows to combine several physical devices into a single logical device (with better properties), presented to a VM [210]. Cloud platforms use this to aggregate disparate storage devices into highly available logical devices with arbitrary storage capacity (e.g. *volumes* in OpenStack). VMs are presented with a logical device through a single access interface, while replication, fault-tolerance and storage aggregation are hidden in the lower abstraction layers. We refer to this logical device as *storage resource* (SR); as a storage unit, an SR can be any unit supported by the disk encryption subsystem.

3.2 Threat Model

We share the threat model with [131,132,180,189], which is based on the Dolev-Yao adversarial model [211] and further assumes that privileged access rights can used by a remote adversary *Adv* to leak confidential information. *Adv*, for example a corrupted system administrator, can obtain remote access to any host maintained by the IaaS provider, but cannot access the volatile memory of guest VMs residing on the compute hosts of the IaaS provider. This property is based on the closed-box execution environment for guest VMs, as outlined in Terra [212] and further developed in [107, 112].

Hardware Integrity Media revelations have raised the issue of hardware tampering en route to deployment sites [2, 213]. We assume that the cloud provider has taken necessary technical and non-technical measures to prevent such hardware tampering.

Physical Security We assume physical security of the data centres where the IaaS is deployed. This assumption holds both when the IaaS provider owns and manages the data center (as in the case of Amazon Web Services, Google Compute Engine, Microsoft Azure, etc.) and when the provider utilizes third party capacity, since physical security can be observed, enforced and verified through known best practices by audit organizations. This assumption is important to build higher-level hardware and software security guarantees for the components of the IaaS.

Low-Level Software Stack We assume that at installation time, the IaaS provider reliably records integrity measurements of the low-level software stack: the Core Root of Trust for measurement; BIOS and host extensions; host platform configuration; Option ROM code, configuration and data; Initial Platform Loader code and configuration; state transitions and

wake events, and a minimal hypervisor. We assume the record is kept on protected storage with read-only access and the adversary cannot tamper with it.

Network Infrastructure The IaaS provider has physical and administrative control of the network. *Adv* is in full control of the network configuration, can overhear, create, replay and destroy all messages communicated between $\mathcal{DM}$ and their resources (VMs, virtual routers, storage abstraction components) and may attempt to gain access to other domains or learn confidential information.

Cryptographic Security We assume encryption schemes are semantically secure and the *Adv* cannot obtain the plain text of encrypted messages. We also assume the signature scheme is unforgeable, i.e. the *Adv* cannot forge the signature of DM_i and that the $\mathsf{M}AC$ algorithm correctly verifies message integrity and authenticity. We assume that the *Adv*, with a high probability, cannot predict the output of a pseudorandom function. We explicitly exclude denial-of-service attacks and focus on *Adv* that aim to compromise the confidentiality of data in IaaS.

3.3 Problem Statement

The introduced *Adv* has far-reaching capabilities to compromise IaaS host integrity and confidentiality. We define a set of attacks available to *Adv* in the above threat model.

Given that *Adv* has full control over the network communication within the IaaS, one of the available attacks is to inject a malicious program or back door into the VM image, prior to instantiation. Once the VM is launched and starts processing potentially sensitive information, the malicious program can leak data to an arbitrary remote location without the suspicion of the domain manager. In this case, the VM instance will not be a *legitimate* instance and in particular not the instance the domain manager *intended* to launch. We call this type of attack a *VM Substitution Attack*:

Definition 3.1 (Successful VM Substitution Attack)**.** Assume a domain manager, DM_i, intends to launch a particular virtual machine vm_l^i on an arbitrary compute host in the set $\mathcal{CH}_{SP_i}$. An adversary, *Adv*, succeeds to perform a **VM substitution attack** if she can find a pair (CH, vm) : $CH \in \mathcal{CH}_{SP_i},\ vm \in \mathcal{VM},\ vm \neq vm_l^i,\ vm \mapsto CH$, where vm will be accepted by DM_i as vm_l^i.

A more complex attack involves reading or modifying the information processed by the VM directly, from the logs and data stored on *CH* or from the representation of the guest VMs' drives on the *CH* file system. This might be non-trivial or impossible with strong security mechanisms deployed on the host; however, *Adv* may attempt to circumvent this through a so-called *CH Substitution Attack* – by launching the guest VM on a compromised *CH*.

Definition 3.2 (Successful CH Substitution Attack)**.** Assume DM_i wishes to launch a VM vm_l^i on a compute host in the set $\mathcal{CH}_{SP_i}$. An adversary, *Adv*, succeeds with a **CH substitution attack** *iff* $\exists\ vm_l^i \mapsto CH_j,\ CH_j \in \mathcal{CH}_{SP_j},\ SP_j \neq SP_i$: vm_l^i will be accepted by DM_i.

Depending on the technical expertise of DM_i, *Adv* may still take the risk of deploying a concealed – but feature-rich – malicious program in the guest VM and leave a fall back option in

case the malicious program is removed or prevented from functioning as intended. *Adv* may choose a *combined VM and CH substitution attack*, which allows a modified VM to be launched on a compromised host and present it to DM_i as the intended VM:

Definition 3.3 (Successful Combined VM and CH Substitution Attack)**.** Assume a domain manager, DM_i, wishes to launch a virtual machine vm_l^i on a compute host in the set $\mathcal{CH}_{SP_i}$. An adversary, *Adv*, succeeds to perform a **combined CH and VM substitution attack** if she can find a pair (CH, vm), $CH \in \mathcal{CH}_{SP_j}$, $SP_j \neq SP_i$, $vm \in \mathcal{VM}$, $vm \neq vm_l^i, vm \mapsto CH$, where vm will be accepted by DM_i as vm_l^i.

Denote by $\mathcal{D}_{vm}^i$ the set of storage domains that $vm \in \mathcal{VM}$, $vm \mapsto CH_i$ can access. We define a *successful storage compute host substitution attack* as follows[1]:

Definition 3.4 (Successful Storage CH Substitution Attack)**.** A DM_i wishes to launch or has launched an arbitrary virtual machine vm_l^i on a compute host in the set $\mathcal{CH}_{SP_i}$. An adversary *Adv* succeeds with a **storage CH substitution attack** if she manages to launch $vm_l^i \mapsto CH_j$, $CH_j \in \mathcal{CH}_{SP_j}$, $SP_j \neq SP_i$ and $\mathcal{D}^i_{vm_l^i} \cap \mathcal{D}^j_{vm_l^i} \neq \emptyset$.

If access to the data storage resource is given to all VMs launched by DM_i, *Adv* may attempt to gain access by launching a VM that *appears* to have been launched by DM_i. Then, *Adv* would be able to leak data from the domain owned by DM_i to other domains. This infrastructure-level attack would not be detected by DM_i and requires careful consideration. A formal definition of the attack[1] follows.

Definition 3.5 (Successful Domain Violation Attack)**.** Assume DM_i has created the domains in the set $\mathcal{D}_i$. An adversary *Adv* succeeds to perform a **domain violation attack** if she manages to launch an arbitrary VM, vm_m^j on an arbitrary host CH_j, i.e. $vm_m^j \mapsto CH_j$, where $\mathcal{D}^j_{vm_m^j} \cap \mathcal{D}_i \neq \emptyset$.

4 Protocol Description

We now describe two protocols that constitute the core of this paper's contribution. These protocols are successively applied to deploy a cloud infrastructure providing additional user guarantees of cloud host integrity and storage security. For protocol purposes, each domain manager, secure component and trusted third party has a public/private key pair (pk/sk). The private key is kept secret, while the public key is shared with the community. We assume that during the initialization phase, each entity obtains a certificate via a trusted certification authority. We first describe the cryptographic primitives used in the proposed protocols, followed by definitions of the main protocol components.

4.1 Cryptographic Primitives

The set of all binary strings of length n is denoted by $\{0,1\}^n$, and the set of all finite binary strings as $\{0,1\}^*$. Given a set U, we refer to the i^{th} element as v_i. Additionally, we use the

[1] In this definition we exclude the possibility of legal domain sharing which would be a natural requirement for most systems. However, with our suggested definition, the legal sharing case can be covered by extending the domain manager role such that it is allowed not to a distinct entity but a role that is possibly shared between domain managers that belong to different organizations.

following notations for cryptographic operations throughout the paper:

- For an arbitrary message $m \in \{0,1\}^*$, we denote by $c = \mathsf{Enc}(K, m)$ a symmetric encryption of m using the secret key $K \in \{0,1\}^*$. The corresponding symmetric decryption operation is denoted by $m = \mathsf{Dec}(K, c) = \mathsf{Dec}(K, \mathsf{Enc}(K, m))$.
- We denote by pk/sk a public/private key pair for a public key encryption scheme. Encryption of message m under the public key pk is denoted by $c = \mathsf{Enc}_{\mathsf{pk}}(m)$[2] and the corresponding decryption operation by $m = \mathsf{Dec}_{\mathsf{sk}}(c) = \mathsf{Dec}_{\mathsf{sk}}(\mathsf{Enc}_{\mathsf{pk}}(m))$.
- A digital signature over a message m is denoted by $\sigma = \mathsf{Sign}_{\mathsf{sk}}(m)$. The corresponding verification operation for a digital signature is denoted by $b = \mathsf{Verify}_{\mathsf{pk}}(m, \sigma)$, where $b = 1$ if the signature is valid and $b = 0$ otherwise.
- A Message Authentication Code (MAC) using a secret key K over a message m is denoted by $\mu = \mathsf{MAC}(K, m)$.
- We denote by $\tau = \mathsf{RAND}(n)$ a random binary sequence of length n, where $\mathsf{RAND}(n)$ represents a random function that takes a binary length argument n as input and gives a random binary sequence of this length in return[3].

4.2 Protocol Components

Disk encryption subsystem a software or hardware component for data I/O encryption on storage devices, capable to encrypt storage units such as hard drives, software RAID volumes, partitions, files, etc. We assume a software-based subsystem, such as *dm-crypt*, a disk encryption subsystem using the Linux kernel Crypto API.

Trusted Platform Module (TPM) a hardware cryptographic co-processor following specifications of the Trusted Computing Group (TCG) [36]; we assume $\mathcal{CH}$ are equipped with a TPM v1.2. The tamper-evident property facilitates monitoring $\mathcal{CH}$ integrity and strengthens the assumption of physical security. An active TPM records the platform boot time software state and stores it as a list of hashes in platform configuration registers (PCRs). TPM v1.2 has 16 PCRs reserved for *static measurements* (PCR0 - PCR15), cleared upon a hard reboot. Additional runtime resettable registers (PCR16-PCR23) are available for *dynamic measurements*. *Endorsement keys* are an asymmetric key pair stored inside the TPM in the trusted platform supply chain, used to create an *endorsement credential* signed by the TPM vendor to certify the TPM specification compliance. A message encrypted ("bound") using a TPM's public key is decryptable only with the private key of the same TPM. *Sealing* is a special case of binding – bound messages are only decryptable in the platform state defined by PCR values. *Platform attestation* allows a remote party to authenticate a target platform and obtain a guarantee that it – up to a certain level in the boot chain – runs software that is identical to the expected one. To do this, an attester requests – accompanied by a nonce – the target platform to produce an attestation quote and the measurement aggregate, or Integrity Measurement List (IML). The TPM generates the attestation quote – a signed structure that includes the IML and the received nonce – and returns the quote and the IML itself. The attestation quote is signed

[2] Alternative notations used for clarity are $\{m\}_{\mathsf{pk}}$ or $\langle m \rangle_{\mathsf{pk}}$.

[3] We assume that a true random function in our constructions is replaced by a pseudorandom function the input-output behaviour of which is "computationally indistinguishable" from that of a true random function.

with the TPMs *Attestation Identity Key (AIK)*. The exact IML contents are implementation-specific, but should contain enough data to allow the *verifier* to establish the target platform [214] integrity. We refer to [36] for a description of the TPM, and to [131,132,202] for protocols that use TPM functionality.

Trusted Third Party (TTP) an entity trusted by the other components. TTP verifies the TPM endorsement credentials on hosts operated by the cloud provider and *enrolls* the respective TPMs' AIKs by issuing a signed AIK certificate. We assume that TTP has access to an access control list (ACL) describing access and ownership relations between $\mathcal{DM}$ and $\mathcal{D}$. Furtermore, TTP communicates with $\mathcal{CH}$ to exchange integrity attestation data, authentication tokens and cryptographic keys. TTP can attest *platform integrity* based on the integrity attestation quotes and the valid AIK certificate from a TPM, and seal data to a trusted host configuration. Finally, TTP can verify the authenticity of $\mathcal{DM}$ and perform necessary cryptographic operations. In this paper, we treat the TTP as a "black box" with a limited, well-defined functionality, and omit its internals. Availability of the TTP is essential in the cloud scenario – we refer the reader to the rich body of work on fault tolerance for approaches to building highly available systems.

Secure Component (SC) this is a verifiable execution module performing confidentiality and integrity protection operations on VM guest data. SC is present on all $\mathcal{CH}$ and is responsible for enforcing the protocol; it acts as a mediator between the $\mathcal{DM}$ and the TTP and forwards the requests from $\mathcal{DM}$ to either the TTP or the disk encryption subsystem. SC must be placed in an isolated execution environment, as in the approaches presented in [107, 112].

4.3 Trusted Launch Construction

We now present our construction for the TL, with four participating entities: *domain manager*, *secure component*, *trusted third party* and *cloud provider* (with the 'scheduler' as part of it). TL comprises a public-key encryption scheme, a signature scheme and a token generator. Figure B.2 shows the protocol message flow (some details omitted for clarity).

$\mathsf{TL.Setup}$: Each entity obtains a public/private key pair and publishes its public key. Below we provide the list of key pairs used in the following protocol:

- $(\mathsf{pk}_{\mathsf{DM_i}}, \mathsf{sk}_{\mathsf{DM_i}})$ – public/private key pair for DM_i;
- $(\mathsf{pk}_{\mathsf{TTP}}, \mathsf{sk}_{\mathsf{TTP}})$ – public/private key pair for TTP;
- $(\mathsf{pk}_{\mathsf{TPM}}, \mathsf{sk}_{\mathsf{TPM}})$ – TPM bind key pair;
- $(\mathsf{pk}_{\mathsf{AIK}}, \mathsf{sk}_{\mathsf{AIK}})$ – TPM attestation identity key pair;

$\mathsf{TL.Token}$: To launch a new VM instance vm_l^i, DM_i generates a token by executing $\tau = \mathsf{RAND}(n)$ and calculates the hash (H_1) of the VM image (vm_l^i) intended for launch, the hash (H_2) of $\mathsf{pk}_{\mathsf{DM_i}}$, and the required security profile SP_i. Finally, $\mathcal{D}^i_{vm_l^i}$ describes the set of domains that vm_l^i with the identifier $idvm_l^i$ shall have access to; the six elements are concatenated into: $m_1 = \left\{ \tau \parallel H_1 \parallel H_2 \parallel SP_i \parallel idvm_l^i \parallel \mathcal{D}^i_{vm_l^i} \right\}$. DM_i encrypts m_1 with $\mathsf{pk}_{\mathsf{TTP}}$ by running $c_1 = \mathsf{Enc}_{\mathsf{pk}_{\mathsf{TTP}}}(m_1)$.

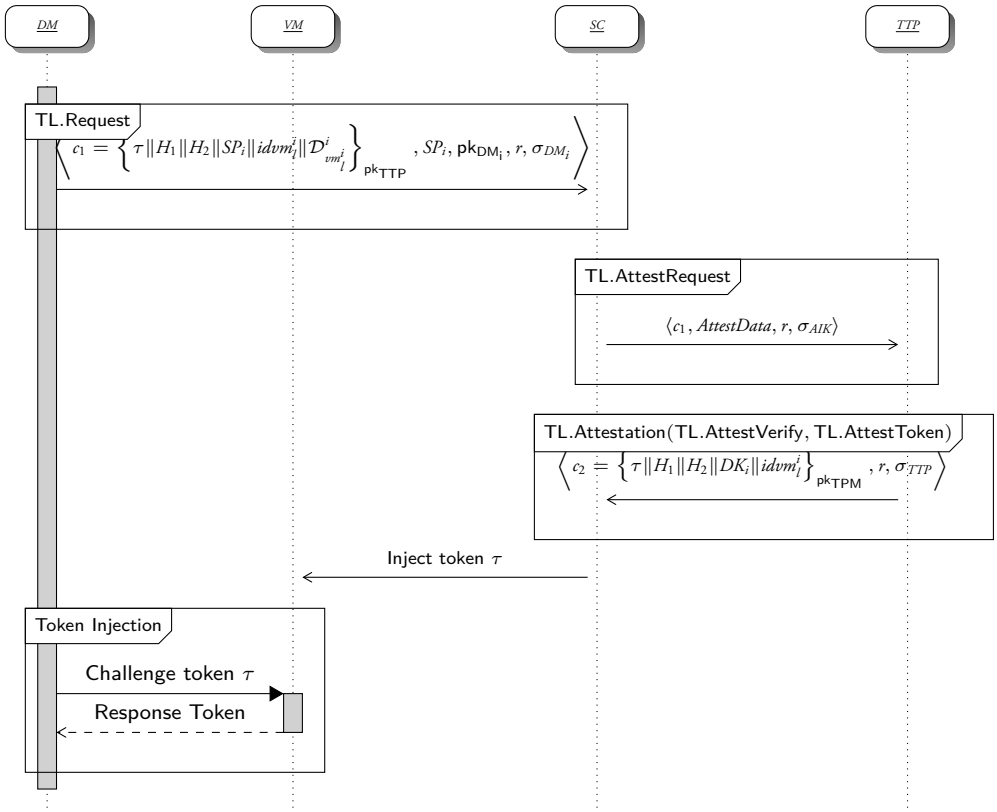


Figure B.2: Message Flow in the Trusted VM Launch Protocol.

Next, DM_i generates a random nonce r and sends the following arguments to initiate a trusted VM launch procedure: $\langle c_1, SP_i, \mathsf{pk}_{\mathsf{DM_i}}, r\rangle$, where c_1 is the encrypted message generated in TL.Token, SP_i is the requested security profile and $\mathsf{pk}_{\mathsf{DM_i}}$ is the public key of DM_i. The message is signed with $\mathsf{sk}_{\mathsf{DM_i}}$, producing σ_{DM_i}. Upon reception, the scheduler assigns the VM launch to an appropriate host with a security profile SP_i, e.g. host CH_i. In all further steps, the nonce r and the signature of the message are used to verify the freshness of the received messages.

Upon reception, SC verifies message integrity and TL.Token freshness by checking respectively the signature σ_{DM_i} and nonce r. When SC first receives a TL.Request message, it uses the local TPM to generate a new pair of TPM-based public/private bind keys, $(\mathsf{pk}_{\mathsf{TPM}}, \mathsf{sk}_{\mathsf{TPM}})$, which can be reused for future launch requests, to avoid the costly key generation procedure. Keys can be periodically regenerated according to a cloud provider-defined policy. To prove that the bind keys are non-migratable, PCR-locked, public/private TPM keys, SC retrieves the `TPM_CERTIFY_INFO` structure, signed with the TPM attestation identity key $\mathsf{pk}_{\mathsf{AIK}}$ [36] using the `TPM_CERTIFY_KEY` TPM command; we denote this signed structure by $\sigma_{\mathtt{TCI}}$. `TPM_CERTIFY_INFO` contains the bind key hash and the PCR value required to use the key; PCR values must not necessarily be in a trusted state to create a trusted bind key pair. This mechanism is explained in further detail in [131].

Next, SC sends an attestation request (TL.AttestRequest) to the TTP, containing the encrypted message (c_1) generated by DM_i in TL.Token, the nonce r and the attestation data ($AttestData$), used by the TTP to evaluate the security profile of CH_i and generate the corresponding TPM bind keys. SC also requests the TPM to sign the message with $\mathsf{sk}_{\mathsf{AIK}}$, producing σ_{AIK}. $AttestData$ includes the following:

- the public TPM bind key $\mathsf{pk}_{\mathsf{TPM}}$;
- the `TPM_CERTIFY_INFO` structure;
- σ_{TCI}: signature of`TPM_CERTIFY_INFO` using $\mathsf{sk}_{\mathsf{AIK}}$;
- IML, the integrity measurement list;
- the TCI-certificate;

Upon reception, TTP verifies the integrity and freshness of TL.AttestRequest, checking respectively the signature σ_{AIK} and nonce r. Next, TTP verifies – according to its ACL – the set $\mathcal{D}^i_{vm^i_l}$ to ensure that DM_i is authorised to allow access to the requested domains for vm^i_l and decrypts the message $m_1 := \mathsf{Dec}_{\mathsf{sk}_{\mathsf{TTP}}}(c_1)$, decomposing it into τ, H_1, H_2, SP_i. Finally, TTP runs an attestation scheme to validate the received attestation information and generate a new attestation token.

Definition 4.1 (Attestation Scheme)**.** An attestation scheme, denoted by TL.Attestation, is defined by two algorithms (AttestVerify, AttestToken) such that:

1. AttestVerify is a deterministic algorithm that takes as input the encrypted message from the requesting DM_i and attestation data, $\langle c_1, AttestData\rangle$, and outputs a result bit b. If the attestation result is positive, $b = 1$; otherwise, $b = 0$. We denote this by $b := \mathsf{AttestVerify}(c_1, \sigma_{AIK}, AttestData)$.
2. AttestToken is a probabilistic algorithm that produces a TPM-sealed attestation token. The input of the algorithm is the result of AttestVerify, the message m to be sealed and the CH $AttestData$. If AttestVerify evaluates to $b = 1$, the algorithm outputs an encrypted message c_2. We write this as $c_2 \leftarrow \mathsf{AttestToken}(b, m, AttestData)$. Otherwise, if AttestVerify evaluates to $b = 0$, AttestToken returns $\perp$.

In the attestation step, TTP first runs AttestVerify to determine the trustworthiness of the target CH_i. In AttestVerify, TTP verifies the signature σ_{TCI} and σ_{AIK} against a valid AIK certificate contained in $AttestData$ and examines the entries provided in the IML. AttestVerify returns $b = 0$ and TTP exits the protocol if the entries differ from values expected for the security profile SP_i. Otherwise, AttestVerify returns $b = 1$ and TTP runs AttestToken to generate a new encrypted attestation token for CH_i. Having verified that the entries in IML conform to the security profile SP_i, TTP generates a symmetric domain encryption key, DK_i, to protect the communication between the SC and TTP in future exchanges. Finally, TTP seals $m_2 = \left\{\tau \| H_1 \| H_2 \| DK_i \| idvm^i_l\right\}$ to the trusted platform configuration of CH_i, using the key $\mathsf{pk}_{\mathsf{TPM}}$ received through the attestation request. The encrypted message ($c_2 \leftarrow \mathsf{AttestToken}(b, m_2, AttestData)$, r), along with a signature (σ_{TTP}) produced using $\mathsf{sk}_{\mathsf{TTP}}$ is returned to SC.

Upon reception, SC checks the message integrity and freshness before unsealing it using the corresponding TPM bind key $\mathsf{sk}_{\mathsf{TPM}}$. The encrypted message is unsealed to the plain text $m_2 = \left\{\tau \| H_1 \| H_2 \| DK_i \| idvm^i_l\right\}$ only if the platform state of CH_i has remained unchanged. SC calculates the hash (H'_1) of the VM image supplied for launch and verifies that its identifier matches the expected identifier $idvm^i_l$; SC also calculates the hash of $\mathsf{pk}_{\mathsf{DM_i}}$ received from the

cloud provider, denoted by H_2'. Finally, SC verifies that $H_1 = H_1'$ and only in that case injects τ into the VM image. Likewise, SC verifies that the public key registered by DM_i with the cloud provider in step TL.Setup has not been altered, i.e. $H_2 = H_2'$ and only in that case injects $\mathsf{pk}_{\mathsf{DM_i}}$ into the VM image prior to launching it.

In the last protocol step, DM_i verifies that vm_l^i has been launched on a trusted platform with security profile SP_i, while vm_l^i verifies the authenticity of DM_i. This is done by establishing a secure pre-shared key TLS session [215] between vm_l^i and DM_i using τ as the pre-shared secret.

4.4 Domain-Based Storage Protection Construction

We now continue with a description of the DBSP protocol. Along with three of the entities already active in the TL protocol – *domain manager*, *secure component*, the *trusted third party* – DBSP employs a fourth one: *the storage resource*. In this case, DM_i interacts with the other protocol components through a VM instance vm_l^i running on CH_i. We assume that vm_l^i has been launched following the TL protocol. The DBSP protocol includes a public and a private encryption scheme, a pseudorandom function for domain key generation, a signature scheme and a random generator. Figure B.3 presents the DBSP protocol mesage flow.

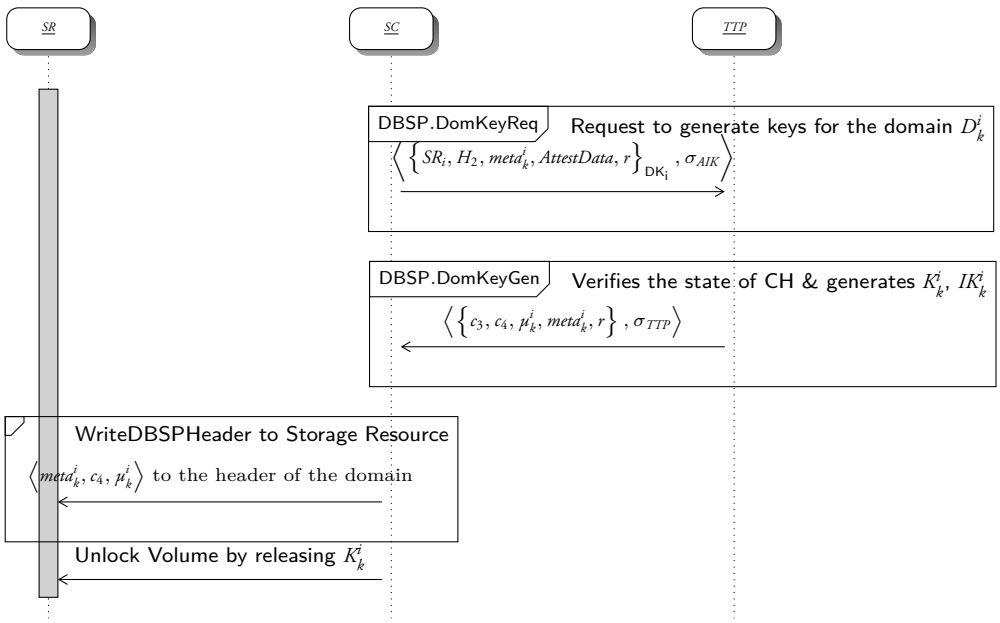


Figure B.3: Message Flow in the Domain-Based Storage Protection Protocol.

DBSP.Setup: We assume that in TL.Setup, each entity has obtained a public/private key pair and published pk.

Assume DM_i requests access for a certain VM vm_l^i to a storage resource SR_i in the domain $D_k^i \in \mathcal{D}_{vm_l^i}^i$. The request is intercepted by the SC, which proceeds to retrieve from TTP a symmetric encryption key for the domain D_k^i.

DBSP.DomKeyReq: SC sends to TTP a request to generate keys for the domain D_k^i. The request contains the target storage resource SR_i, hash H_2 of $\mathsf{pk}_{\mathsf{DM_i}}$, the nonce r and $meta_k^i$, containing the unique domain identifier and the security profile required to access the domain D_k^i, i.e., $meta_k^i = \left\{D_k^i, SP_i\right\}_{\mathsf{pk}_{\mathsf{TTP}}}$; SC uses the symmetric key DK_i received during TL.Attestation to protect message confidentiality, and the local TPM to sign the message with $\mathsf{sk}_{\mathsf{AIK}}$, producing σ_{AIK} (see DBSP.DomKeyReq in Figure B.3).

Upon the reception of DBSP.DomKeyReq, TTP verifies the freshness and integrity of the request and proceeds to the next protocol step, DBSP.DomKeyGen, only if this verification succeeds.

DBSP.DomKeyGen: A probabilistic algorithm enabling TTP to generate a symmetric encryption key (K_k^i) and integrity key (IK_k^i) for a domain D_k^i. TTP generates a nonce using a random message $m_i \in \{0,1\}^n$ by executing $n_i = \mathsf{RAND}(m_i)$. Next, TTP uses a PRF to generate the keys for domain D_k^i, by evaluating the following:

$$K_k^i = PRF\left(K_{TTP}, D_k^i \| SP_i \| n_i\right),$$

$$IK_k^i = PRF\left(K_{TTP}, D_k^i \| n_i\right),$$

where K_{TTP} is a master key that does not leave the security perimeter of TTP, K_k^i is a symmetric encryption key to confidentiality protect the data and IK_k^i a symmetric key to verify the integrity of the stored data.

TTP seals K_k^i and IK_k^i to the trusted configuration of CH_i by calculating

$$c_3 = \mathsf{Enc}_{\mathsf{pk}_{\mathsf{TPM}}}\left(K_k^i \| IK_k^i\right).$$

TTP encrypts the generated nonce n_i and the provided security profile SP_i by evaluating $c_4 = \mathsf{Enc}_{\mathsf{K}_{\mathsf{TTP}}}(n_i \| SP_i)$ to later use it for verification. Next, TTP generates a message authentication code μ by evaluating $\mu_k^i = \mathsf{MAC}(K_{TTP}, n_i \| SP_i)$. The domain key generation algorithm is denoted by $\left(c_3, c_4, \mu_k^i\right) \leftarrow \mathsf{DBSP.DomKeyGen}(n_i, K_{TTP}, \mathsf{sk}_{\mathsf{TPM}})$.

Having generated the domain key, TTP responds to the DBSP.DomKeyReq by sending $\{c_3, c_4, \mu_k^i, meta_k^i, r\}$ with the signature σ_{TTP}. Upon reception, SC first verifies message integrity and freshness, and calls the local TPM to unseal c_3, producing $K_k^i \| IK_k^i$ if and only if CH_i remains in the earlier trusted state. Next, SC stores $meta_k^i$, c_4 and μ_k^i in the domain header and uses K_k^i, IK_k^i as inputs to the disk encryption subsystem on CH_i, which decrypts and verifies the data integrity of the mounted volume hosting D_k^i before providing plain text access to vm_l^i.

To recreate the encryption and integrity keys for the domain D_k^i, SC sends a request similar to DBSP.DomKeyReq, adding to the message the values c_4 and μ_k^i, which are stored in the domain header. Upon reception, TTP verifies the integrity of the received value c_4 by calculating $\mu_k^i = \mathsf{MAC}(K_{TTP}, n_i \| SP_i)$. If the integrity verification of c_4 is positive, TTP decrypts it to $n_i \| SP_i = \mathsf{Dec}_{\mathsf{sk}_{\mathsf{TTP}}}(c_4)$ and calculates the domain key as in DBSP.DomKeyGen, using the existing token n_i instead of generating a new one[4].

[4]Key retrieval is currently not covered in the security analysis due to space limitations

5 Security Analysis

We now analyse the TL and DBSP protocols in the presence of an adversary. We prove the security of both schemes through a theoretical analysis, showing that our protocols are resistant to the attacks presented in Section 3.3.

Proposition 1 (VM Substitution Soundness)**.** The TL protocol is sound against the VM substitution attack.

Proof: An adversary *Adv* trying to launch $vm \neq vm_l^i$ on *CH* can only get *vm* accepted by DM_i if the last mutual authentication step in the trusted launch procedure is successful. In turn, this step only succeeds if at least one of the following two options is true:

a. The secure component SC uses a different token, $\tau' \neq \tau$ accepted by DM_i in the final secure channel establishment.

b. The secure component SC on *CH* uses the very same token τ used by DM_i when launching vm_l^i.

Option a can only succeed if *Adv* can break the mutual authentication in the secure channel setting. Given that the selected secure channel scheme is sound and τ is sufficiently long and selected using a sound random generation process, the *Adv* fails to break the last protocol step. Hence, as long as the secure channel protocol is sound, the overall protocol construction is also sound against this attack option.

Option b can only succeed if the adversary either manages to guess a value $\tau' = \tau$ when launching *vm* or manages to either obtain τ when DM_i launches vm_l^i or replace the association between τ and vm_l^i with an association between τ and *vm* when DM_i launches vm_l^i, by attacking any of the protocol steps preceding the final mutual authentication step. A successful attack has in this case the probability $\tau' = \tau$ equals to $1/2^n$, where n is the length of the token value and is infeasible if n is large enough. Below, we show why the adversary also fails with respect to the last option.

- TL.Token. Assume the adversary intercepts the TL.Token message. Then the adversary has two options: she might either try to modify the TL.Token message (option 1) with the goal to replace the association between τ and the vm_l^i with τ and *vm*, or she might try to obtain the secret value τ (option 2) and then launch *vm* with this τ value on an arbitrary valid provider platform. We discuss both these options below.

  - TL.Token *Option 1*: A modification can only be achieved by the adversary by either breaking the public key encryption scheme used to produce c_1 or trying to make this modification on c_1 by direct modification (without first decrypting it) and sign the modified c_1 with an own selected private key. The former option fails due to the assumption of public key encryption scheme soundness and the latter due to that modifying a public encrypted structure without knowledge of the private key is infeasible.
  - TL.Token *Option 2*: Direct decryption of c_1 fails due to the assumption of soundness of the public key encryption scheme used to produce c_1. The only remaining alternative for the adversary is relaying the TL.Token to a platform $CH' \in \mathcal{CH}_{SP_i}$, which is under the full control of the adversary. Further, *Adv* follows the protocol

and issues the command TL.AttestRequest using the intercepted c_1, *AttestData* and σ_{AIK}. However, this fails at the TL.Attestation step since *AttestData* does not contain a valid AIK certificate unless the adversary has managed to get control of a valid platform in the provider network with a valid certificate *or* she has managed to break the AIK certification scheme. The former option violates the assumption of physical security of the provider computing resources while the latter option violates the assumption of a sound public key and AIK certification schemes.

- TL.AttestRequest. The adversary could either try to impersonate this message with the goal of obtaining τ or the association between τ and vm_l^i. This impersonation attempt fails as the whole sent structure is signed with the $\mathsf{pk}_{\mathsf{AIK}}$ with a secure public key signing scheme. Furthermore, attempts to resend an old *valid* TL.AtttestRequest fail as the H_1 verification that the SC receives in return fails as it does point on the *old* VM. Similarly, any attempts to modify TL.AttestRequest fail as the whole structure is signed with a secure signature scheme.
- TL.Attestation. Any attempt by the adversary to obtain τ would be equal to breaking the public key encryption of TL.AttestToken. Similarly, any attempt to modify c_2 fails due to the fact that modification of a public encrypted structure without knowledge of the private key is unfeasible if the public key encryption scheme is sound. Any attempt by the adversary to replace an old recorded valid TL.AttestToken message fails as such messages do contain a VM image hash H_1 different than the one expected by the SC. □

Proposition 2 (CH Substitution Soundness)**.** The TL protocol is sound against the CH substitution attack.

Proof: DM_i intends to launch a virtual machine vm_l^i on an arbitrary compute host CH_i with a security profile SP_i. An adversary *Adv* trying to launch vm_l^i on $CH_j \in \mathcal{CH}_{SP_j}$, $SP_j \neq SP_i$, can only get vm_l^i accepted by DM_i if the last mutual authentication step in the trusted launch procedure is successful. In turn, this step can only succeed if at least one of the following two options is true:

a. The secure component SC is using a different token, $\tau' \neq \tau$ that is accepted by DM_i in the final secure channel establishment.
b. The secure component SC on CH_j is using the very same token τ used by DM_i when launching vm_l^i.

Option a is impossible as proved in Proposition 1. *Option b* can only succeed if the adversary either manages to guess a value $\tau' = \tau$ when launching vm_l^i or manages to induce the TTP to seal the token τ to the configuration of CH_j. Finding $\tau' = \tau$ is infeasible for the adversary as shown in Proposition 1. Below, we show why the adversary also fails with respect to the second option.

Assume *Adv* intercepts the TL.Token message. Then it has two options: either attempt to launch vm_l^i on a compute host $CH_j \notin \mathcal{CH}_{SP_i}$ or on $CH_j \in \mathcal{CH}_{SP_i}$.

- TL.Token $CH_j \notin \mathcal{CH}_{SP_i}$: The *Adv* can replace the following information from the TL.Token message: SP_i with SP_j, $\mathsf{pk}_{\mathsf{DM_i}}$ with $\mathsf{pk}_{\mathcal{ADV}}$, which is a public key generated by the *Adv* and σ_{c_1} with $\sigma_{\mathcal{ADV}} = \mathsf{Sign}_{\mathsf{sk}_{\mathcal{ADV}}}(\mathsf{c_1})$. By doing this, she can successfully proceed beyond the TL.AttestRequest step since SC is not able to detect the substitution. However, this attack fails at the TL.Attestation step since the *AttestData* sent to the TTP evaluates to a security profile $SP_j \neq SP_i$ in contradiction with the preference of DM_i contained in c_1.

- TL.Token $CH_j \in \mathcal{CH}_{SP_i}$: The *Adv* can replace the following information from the TL.Token message: $\mathsf{pk}_{\mathsf{DM_i}}$ with $\mathsf{pk}_{\mathcal{ADV}}$, which is a public key generated by the *Adv* and σ_{c_1} with $\sigma_{ADV} = \mathsf{Sign}_{\mathsf{sk}_{\mathcal{ADV}}}(\mathsf{c_1})$. By doing this, he can successfully proceed beyond the TL.AttestRequest step since SC is unable to detect the substitution. However, this attack fails at the TL.Attestation step since the $\mathsf{pk}_{\mathsf{AIK}}$ key used to produce the signature σ_{AIK} is not among the keys enrolled with the TTP according to Section 4.2.

The cases of TL.AttestRequest and TL.Attestation fail as demonstrated in Proposition 1. □

Proposition 3 (Combined VM and CH Substitution Soundness)**.** The TL protocol is sound against the VM and CH substitution attack.

Proof: The exculpability of the VM substitution attack and the CH substitution attack implies that the TL protocol is secure against the combined VM and CH substitution attack. □

Proposition 4 (Storage CH Substitution Soundness)**.** The DBSP protocol is sound against the storage CH attack.

Proof: Adversary *Adv* can only succeed with a storage CH substitution attack if she manages to launch a VM instance $vm_l^i \mapsto CH_i$, $CH_i \in \mathcal{CH}_{SP_i}$ on a host $CH_j \in \mathcal{CH}_{SP_j}$, $SP_j \neq SP_i$ and $\mathcal{D}^i_{vm_l^i} \cap \mathcal{D}^j_{vm_l^i} \neq \emptyset$. This can only be achieved if she requests launch of vm_l^i on a platform with profile SP_j. According to Proposition 2 and Proposition 3, such launch requests are rejected by DM_i; however, this does not prevent the *Adv* from attempting these options. The following two alternatives are available to the adversary:

a. The *Adv* launches $vm_l^i \mapsto CH_j$ on a platform under its own control (i.e. outside the provider domain).
b. The *Adv* launches $vm_l^i \mapsto CH_j$ on a valid platform in the provider network.

Option a: This option implies that the TL.AttestRequest step fails as shown in the proof of Proposition 1. In this case, the platform controlled by *Adv* does not get the symmetric key DK_i in return to the attestation request. Without access to DK_i, the only remaining option for the adversary is to attempt to break the final key request or the disc encryption scheme. Thus the following options are available:

- DBSP.DomKeyReq : The first option is to intercept a valid DBSP.DomKeyReq message for a storage domain $D_k^i \in \mathcal{D}^i_{vm_l^i}$ and replace the intercepted signature σ_{AIK} with her own own signature, σ'_{AIK} over the very same encrypted request (encrypted with a valid DK_i). However, similar to the earlier attempt to perform a TL.AttestRequest, this fails since the *Adv* does not have access to a valid attestation key. Any other attempt to send the adversary's own DBSP.DomKeyReq fails for the same reason.
- DBSP.DomKeyGen : The remaining option is to observe a valid DBSP.DomKeyGen for a domain $D_k^i \in \mathcal{D}^i_{vm_l^i}$ and attempt to access the encrypted storage keys. The latter fails due to the assumption of the TPM public key scheme soundness.
- *Attack Storage Encryption Scheme:* The remaining option for the *Adv* in this case is to directly break the disc encryption scheme. However, this is infeasible according to the disc encryption scheme soundness.

Option b: According to this option, the *Adv* tries launching vm_l^i using TL.Token on a platform with profile SP_j using its own credentials. The following impersonation alternatives are available:

- *Own token:* The adversary *Adv* sends a TL.Token message required by the protocol: $\mathsf{Enc}_{\mathsf{pk}_{\mathsf{TTP}}}\left(\tau \parallel H_1 \parallel H_2 \parallel SP_j \parallel idvm_l^i \parallel \mathcal{D}^i_{vm_l^i}\right), SP_j, \mathsf{pk}_{\mathcal{ADV}}, r, \sigma_{\mathcal{ADV}}$, where H_2 *either* is the hash of $\mathsf{pk}_{\mathsf{DM_i}}$ or the hash of $\mathsf{pk}_{\mathcal{ADV}}$. If the first option is used, the SC obtains in return to TL.AttestRequest, i.e. the TL.Attestation message, a sealed value with a hash $H_2' \neq H_2$ which causes the SC to abort the launch. If the second option is used, the complete launch procedure succeeds as expected. However, when the SC later requests the key for SR_i using the DBSP.DomKeyReq message, it includes the hash H_2 of the the *Adv* public key ($\mathsf{pk}_{\mathcal{ADV}}$) in the encrypted and signed request. *Adv* cannot change the hash value in this request unless she breaks the signature scheme of the request. Upon receiving the request, TTP identifies that *Adv* is *not* allowed to access $D_k^i \in \mathcal{D}^i_{vm_l^i}$ and does not return the storage keys in DBSP.DomKeyGen.
- *Legitimate token:* In this option, the *Adv* observes a valid c_1 in TL.Token for *another vm* with access rights to the intended domain and uses it to launch an own valid TL.Token message: $c_1, SP_j, \mathsf{pk}_{\mathcal{ADV}}, r, \sigma_{ADV}$. However, in this case the TL.AttestRequest fails as the profile in c_1 does not match the platform attested data. Furthermore, if the SC receives a reply to TL.AttestRequest, i.e. a TL.Attestation message, it would receive a sealed value with a hash $H_2' \neq H_2$, causing the SC to abort the launch. □

Proposition 5 (Domain Violation Attack)**.** The DBSP protocol is sound against the domain violation attack.

Proof: Similar to the proof of Proposition 4, *Adv* has the following two options:

a. The *Adv* launches $vm_m^j \mapsto CH_j$ on a platform under its control (i.e. outside the provider domain).
b. The *Adv* launches $vm_m^j \mapsto CH_j$ on a valid platform in the provider network.

Option a: This option fails in analogy with the proof of Proposition 4, as *Adv* fails to successfully launch vm_m^j and her remaining options are to either attack the final key request or the disc encryption scheme, which both fail (see proof of Proposition 4).

Option b: In analogy with the proof of Proposition 4, *Adv* has only two options available: a full impersonation with an own chosen token of type $\mathsf{Enc}_{\mathsf{pk}_{\mathsf{TTP}}}\left(\tau \parallel H_1 \parallel H_2 \parallel SP_j \parallel idvm_m^j \parallel \mathcal{D}^j_{vm_m^j}\right)$, SP_j, $\mathsf{pk}_{\mathcal{ADV}}$, r, $\sigma_{\mathcal{ADV}}$, $\mathcal{D}^j_{vm_m^j} \subseteq \mathcal{D}_i$, or a partial impersonation reusing an observed c_1 of type $c_1, SP_j, \mathsf{pk}_{\mathcal{ADV}}, r, \sigma_{\mathcal{ADV}}$ for a subset of target storage domain. Both options fail in analogy with the arguments presented for the proof of Proposition 4. □

6 Implementation and Results

We next describe the implementation of the TL and DBSP protocols followed by experimental evaluation results.

6.1 Test bed Architecture

We describe the infrastructure of the prototype and the architecture of a distributed EHR system installed and configured over multiple VM instances running on the test bed.

Infrastructure Description

The test bed resides on four Dell PowerEdge R320 hosts connected on a Cisco Catalyst 2960 switch with 801.2q support. We used Linux CentOS, kernel version 2.6.32[5] and the OpenStack cloud computing platform[6] (version Icehouse) using KVM virtualization support. The prototype IaaS includes one *"controller"* running essential platform services (scheduler, PKI components, SDN control plane, VM image storage, etc.) and three compute hosts running the VM guests. Compute hosts dedicate most of their resources to the VM guests, while the controller runs essential platform services, such as: the scheduler, database wrappers, PKI components, SDN control plane, web graphical user interface, VM image storage service, etc. All hosts run additional processes necessary to support and integrate the IaaS platform functionality. The topology of the prototype SDN reflects three larger domains of the application-level deployment (front-end, back-end and database components) in three virtual LAN (VLAN) networks.

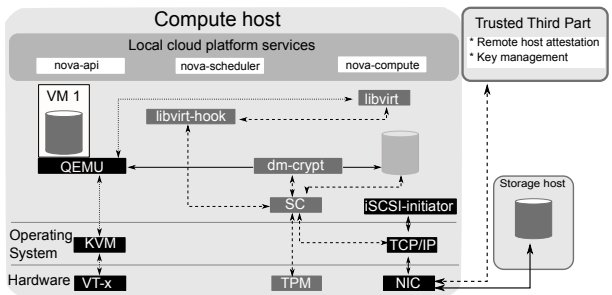


Figure B.4: Placement of the SC in the prototype implementation. **'nova-api'**, **'nova-api'**, **'nova-compute'**: implementation-specific OpenStack components; **'QEMU'**: open-source machine emulator and virtualizer; **'KVM'** virtualization infrastructure for the Linux kernel; **'VT-x'**: processor extensions for virtualization support; **'libvirt'**: virtualization API; **'libvirt-hook'**: libvirt infrastructure for customization scripts; **'dm-crypt'**: disk encryption library; **'SC'**: secure component; **'TPM'**: Trusted Platform Module; **'iSCSI-initiator'**: endpoint to initiate the iSCSI protocol; **'TCP/IP'**: TCP/IP stack; **'NIC'**: network interface card.

The compute hosts use libvirt[7] for virtualization functionality. To implement the DBSP protocol we modified libvirt 0.10.2 and used the "libvirt-hooks" infrastructure to implement the SC for the TL and DBSP protocols. SC unlocks the volumes on compute hosts and interacts with the TPM and TTP (see Figure B.4). It uses a generic server architecture where the SC daemon handles each request in a separate process. An inter process communication (IPC)

[5]Full version identifier: 2.6.32-358.123.2.openstack.el6.x86_64
[6]OpenStack project website: `https://www.openstack.org/`
[7]Libvirt website: http://libvirt.org/

Table B.1: Overhead for unlocking a volume with DBSP (all times in ms)

Process	Event	Time
QEMU	Begin handle unlock request	0.083
SC	Requesting key from TTP	0.609
SC	Unseal key in TPM	2700.870
SC	Unlocking volume with cryptsetup	11.834
QEMU	End handle unlock request	26
	TOTAL	2714.004

protocol defines the types of messages processed by the SC. The IPC protocol uses sychronous calls with several types of requests for the respective SC operations; the response contains the exit code and response data. A detailed architecture of SC, including the main libraries that it relies on, is presented in Figure B.5.

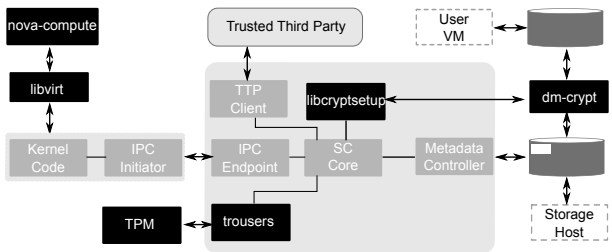


Figure B.5: Close-up view of the secure component implementation architecture, presented as a combination of components and existing libraries. Components are capitalized, while the libraries start with lowercase. **'nova-compute'**: implementation-specific OpenStack component; **'libvirt'**: virtualization API; **'Kernel Code'**: Linux Kernel; **'IPC Initiator'**: code to initiate inter-process communication calls to the secure component; **'IPC Endpoint'**: code to terminate inter-process communication calls to the secure component; **'TTP Client'**: client code to communicate with the TTP; **'SC Core'**: secure component kernel code; **'Metadata Controller'**: component to format and parse storage resource metadata; **'libcryptsetup'**: communication api for dm-crypt; **'dm-crypt'**: disk encryption library; **'trousers'**: TPM access library; **'TPM'**: Trusted Platform Module.

Application Description

The prototype also includes a distributed EHR system deployed over seven VM instances. This system contains one client VM, two front-end VMs, two back-end VMs, a database VM and an auxiliary external database VM. Six of the VM instances operate on Microsoft Windows Server 2012 R2, with one VM running the client application operates on Windows 7. The components of the EHR system communicate using statically defined IP addresses on the respective VLANS described in Section 6.1. Load balancing functionality provided by the underlying IaaS allots the load among front-end and back-end VM pairs. The hosts of the cluster are compatible with the TL protocol, which allows an infrastructure administrator to perform a trusted launch of VM instances on qualified hosts. Similarly, the infrastructure administrator can apply the DBSP protocol to protect sensitive information stored on the database servers.

6.2 Performance evaluation

Trusted launch Figure B.6 shows the duration of a VM launch over 100 successful instantiations: the TL protocol extends the duration of the VM instantiation (which does not include the OS boot time) on average by 28%. However, in our experiments we have used a minimalistic VM image (13.2 MB), based on CirrOS [8], while launching larger VM images takes significantly more time and proportionally reduces the overhead induced by TL.

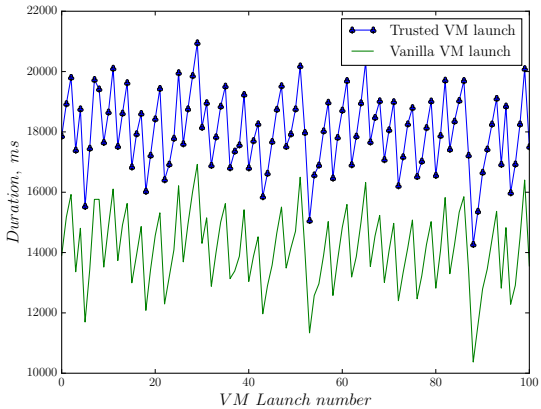


Figure B.6: Overhead induced by the TL protocol during VM instantiations.

DBSP Processing time Table B.1 shows a breakdown of the time required to process a storage unlock request, an average of 10 executions. Processing a volume unlock request on the prototype returns in ≈2.714 seconds; however, this operation is performed *only when attaching* the volume to a VM instance and does not affect the subsequent I/O operations on the volume. A closer view highlights the share of the contributing components in the overall overhead composition. Table B.1 clearly shows that the TPM unseal operation lasts on average ≈2.7 seconds, or 99.516% of the execution time. According to Section 4.2, in this prototype we use TPMs v1.2, since a TPM v2.0 is not available on commodity platforms at the time of writing. Given that the vast majority of the execution time is spent in the TPM unseal operation, implementing the protocol with a TPM v2.0 may yield improved results.

DBSP Encryption Overhead Next, we examine the processing overhead introduced by the DBSP protocol. Figure B.7 presents the results of a disk performance benchmark obtained using IOmeter[9]. To measure the effect of background disk encryption with DBSP, we attached two virtual disks to a deployed server VM described in 6.1. The storage volumes were physically located on a different host and communicating over iSCSI. We ran a benchmark with two parallel workers on the plaintext and DBSP-encrypted volumes over 12 hours. Next, we disabled in the host BIOS the AES-NI acceleration, created and attached a new volume to the VM and reran the benchmark. This has produced three performance data result sets: plaintext, DBSP encryption and DBSP encryption with AES-NI acceleration. Figure B.7 summarises the *total*

[8] CirrOS project website: https://launchpad.net/cirros
[9] IOmeter project website: `http://iometer.org`

IO, read IO and *write IO* results. It is visible that the measurements *'4 KiB aligned (*DBSP*) with AES-NI'* and *'1 MiB (*DBSP*) with AES-NI'* are roughly on par with the plaintext baseline: *'4 KiB aligned'* and *'1 MiB'*. The performance overhead induced by background encryption is at 1.18% for read IO and 0.95% for write IO. We can expect that this performance penalty will be further reduced as the hardware support for encryption is improved. Disk encryption without hardware acceleration (*'4 KiB aligned (*DBSP*)'* and *'1 MiB (*DBSP*)'*) is significantly slower, as expected, with a performance penalty of respectively 49.22% and 42.19% (total IO). It is important to reemphasize that the runtime performance penalty is determined exclusively by the performance of the disk encryption subsystem. DBSP only affects the time required to unlock the volume when it is attached to the VM instance, as presented in Table B.1.

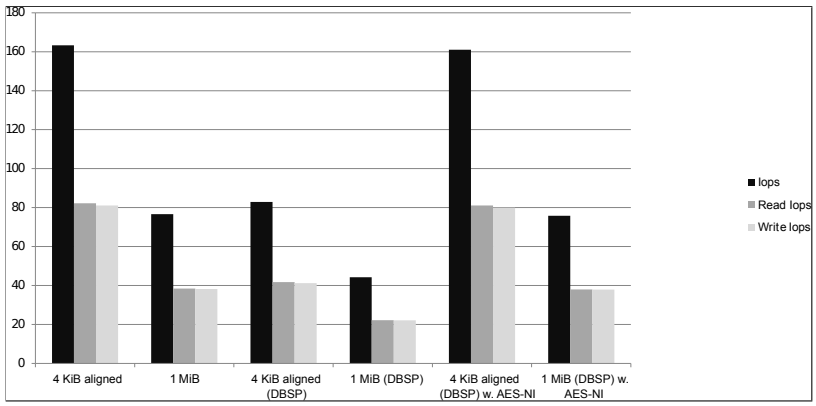


Figure B.7: Benchmarks results on identical drives: plaintext, with DBSP, with DBSP and AES-NI acceleration.

7 Application Domain

The presented results are based on work in collaboration with a regional public healthcare authority to address some of its concerns regarding IaaS security. We have deployed the prototype described in Section 6, further extended by integrating a medication database, and evaluated it through end-user validation and performance tests. Our results demonstrate that it is both *possible* and *practical* to provide strong platform software integrity guarantees to IaaS tenants and efficiently isolate their data using established cryptographic tools. Platform integrity guarantees allow tenants to take better decisions on both workload migration to the cloud and workload placement within IaaS. This contrasts with the current, "flat" trust model, where all IaaS hosts declare the same – but *unverifiable* for the tenant – trust level.

An essential conclusion of this practical exercise is that the additional cost of providing security guarantees can be *effectively offset* by composing cloud services from different competing providers, without having to delegate the trust among these providers. Thus, in our cloud model the tenant can purchase cheaper cloud disk storage without any additional risk for data confidentiality.

Another conclusion is that while organizations operating on sensitive data, e.g. public health-

care authorities, consider the risks of migrating data to IaaS clouds as unacceptable, the majority of available providers use commercial-off-the-shelf (COTS) cloud platforms with limited capabilities to enhance the security of their deployments, failing to meet the customer requirements. This demonstrates the need to incorporate integrity verification and data protection mechanisms into popular COTS cloud platforms by default. We hope that these important lessons will inspire new secure, usable and cost-effective solutions for cloud services.

On the practical side, specifically regarding the role of the TTP, we envision two scenarios. The TTP could either be managed by the tenant itself (for organizations with enough resources and expertise), or by an external organization (similar to a certificate authority). The first scenario allows the tenant to retain the benefits of cloud services along with additional security guarantees. Similarly, in the second scenario, smaller actors can obtain the same benefits without the need to invest into own attestation infrastructure. In both scenarios, in order to protect the cloud provider the TTP would only operate on a *physical slice* of the resources (i.e. a subset of compute hosts) that correspond to the respective tenant domains.

8 Conclusion

From a tenant point of view, the cloud security model does not yet hold against threat models developed for the traditional model where the hosts are operated and used by the same organization. However, there is a steady progress towards strengthening the IaaS security model. In this work we presented a framework for trusted infrastructure cloud deployment, with two focus points: VM deployment on trusted compute hosts and domain-based protection of stored data. We described in detail the design, implementation and security evaluation of protocols for trusted VM launch and domain-based storage protection. The solutions are based on requirements elicited by a public healthcare authority, have been implemented in a popular open-source IaaS platform and tested on a prototype deployment of a distributed EHR system. In the security analysis, we introduced a series of attacks and proved that the protocols hold in the specified threat model. To obtain further confidence in the semantic security properties of the protocols, we have modelled and verified them with ProVerif [216]. Finally, our performance tests have shown that the protocols introduce a insignificant performance overhead.

This work has covered only a fraction of the IaaS attack landscape. Important topics for future work are strengthening the trust model in cloud network communications, data geolocation [217], and applying searchable encryption schemes to create secure cloud storage mechanisms. Our results show that it is possible and practical to provide strong platform software integrity guarantees for tenants and efficiently isolate their data using established cryptographic tools. With reasonable engineering effort the framework can be integrated into production environments to strengthen their security properties.

Paper C

Domain Based Storage Protection with Secure Access Control for the Cloud

Nicolae Paladi, Antonis Michalas and Christian Gehrmann

Abstract

Cloud computing has evolved from a promising concept to one of the fastest growing segments of the IT industry. However, many businesses and individuals continue to view cloud computing as a technology that risks exposing their data to unauthorized users. We introduce a data confidentiality and integrity protection mechanism for Infrastructure-as-a-Service (IaaS) clouds, which relies on trusted computing principles to provide transparent storage isolation between IaaS clients. We also address the absence of reliable data sharing mechanisms, by providing an XML-based language framework which enables clients of IaaS clouds to securely share data and clearly define access rights granted to peers. The proposed improvements have been prototyped as a code extension for a popular cloud platform.

1 Introduction

Cloud computing continues its path towards wider adoption, and more companies attempt to tap into the promise of cost savings. Evidence to the success of the Infrastructure-as-a-Service (IaaS) model are both the increasing competition among IaaS cloud providers and the rush to migrate to IaaS clouds among businesses.

Moving traditional infrastructure to shared virtualized environments raises new security challenges. We can hope that users are aware of such security issues and strive to obtain from

IaaS clouds security properties – such as execution isolation and control over data – which are on a par with on-site deployments. However, considering that clients of IaaS clouds share execution and storage resources with other tenants, anonymous to them, currently available security solutions have proved to be insufficient. In [17], the authors have achieved to map the cloud infrastructure, collocate a malicious virtual machine (VM) instance with a target instance and launch side-channel attacks to extract information. The authors of [18] describe a range of attacks on management interfaces of public clouds using signature wrapping and XSS attacks. As a result, the attackers would be able to compromise the control interfaces of the IaaS cloud and misuse the cloud resources of other tenants. Finally, a recent example are the "dirty disks" of a public IaaS provider [21], where clients were able to read from improperly sanitised storage devices data stored by previous clients. This directly points to one of the unsolved problems in public IaaS clouds – ensuring data protection and secure data sharing.

Full-disk encryption has emerged as a solid solution for data confidentiality protection and is also mentioned in [21] as a solution to the "dirty disks" problem. However, full-disk encryption creates hurdles for data sharing, widely recognized as an essential feature for cloud applications [205]. Despite the variety of available open source cloud management platforms (e.g OpenStack, Eucalyptus, OpenNebula), allocation of read-write permissions for shared data between collaborating tenants still remains an open problem. In this paper we address the outlined gap. We improve and extend previous work by adding capabilities to both grant access to data to other IaaS cloud clients and assign access permissions.

1.1 Our Contribution

The contribution of this work is twofold. We first present a secure storage protection protocol that provides per-VM instance access control and allows the client to control a VM instance's read and write access rights over a storage device at launch time. We introduce an XML-based language framework that allows users to define role-based access control in order to grant access, based on permissions, to other users in the IaaS cloud. Our protocol allows a granular access rights management per VM instance and storage device. In addition, we analyse our protocol and show it is resistant under malicious behaviors. Second, we complement the analysis with extensive experimental results that show the effectiveness of the protocol.

1.2 Organization

In Section 2, we review some of the most important protocols that provide domain storage protection in public IaaS clouds and mechanisms for secure data sharing in clouds. In Section 3, we describe the problem of data protection in IaaS clouds and define the important terms used throughout the paper. In Section 4, we describe the system model of a cloud platform (*CP*) which stands at the basis of our protocol implementation. In Section 5, we present our protocol for secure storage protection data sharing mechanism in IaaS clouds. Section 7 contains experimental results of the protocol benchmarks, while Section 8 concludes the paper.

2 Related Work

The importance of data confidentiality protection and isolation of data between IaaS cloud tenants is underlined by the attention it has received from the research community.

In [218], authors propose a full disk background encryption model by introducing TCVisor, a hypervisor with a parapass-through architecture that introduces *TPM* support and novel key-management approach. Support for *TPM* is added in order to store parts of cryptographic keys and whole-disk checksums for integrity checking. In addition to that, Merkle trees are used for integrity verification and protection of the root value relying on *TPM* functionality. However, the poor description of storing/sealing the root value of the Merkle tree hash, raises doubts about protocol's validity.

The authors of [107] focus on hypervisor-level data protection and introduce Cloudvisor – a security monitor underneath the commodity hypervisor which provides protection to the hosted VMs. CloudVisor runs in host mode and encrypts the data exchange between a VM and the hypervisor and verifies the integrity, freshness and ordering of disk I/O data. One immediate limitation of the solution in [107] are the severe functionality limitations, such as support for a single VM instance. Our protocol uses the functionality offered by commodity hypervisors in order to ensure data protection and does not introduce such severe limitations.

A solution for management of encrypted data is described in [204], where each information block is encrypted with a different symmetric key, thus aiming for a cryptography-based access control. An 'information block' represents an abstract concept of arbitrary size. The paper assumes a lazy revocation model, where a user indefinitely maintains access to the data that she could reach prior to revocation (regardless of whether or not the data has been accessed before access revocation). While similar to our model in aspects such as information blocks and encryption with different symmetric keys, we propose an active revocation model, where the keys can not be retrieved once the access is revoked.

Few of the IaaS storage protection schemes address the problem of sharing files with certain permissions. In [206], authors analysed access rights management of shared versioned encrypted data on cloud infrastructure for a restricted group. in their model they proposed an adoption for enabling scalable and flexible key management within cloud. By representing access rights as a graph and based on [219], authors were able to distinguish between the keys used for encrypting data and the encrypted updates on the keys, enabling flexible join/leave operations of clients. Despite being an attractive approach, the requirement for client-side encryption limits the applicability of the scheme and ignores the limitations to functionality (such as indexing and search) that it introduces. In our model all cryptographic operations are performed on trusted IaaS compute hosts, which are able to allocate more computational resources than client devices.

Data-Protection-as-a-Service (DPaaS) [205] is a conceptual architecture which aims to address the need for integrity, privacy, access transparency, ease of verification and rich computation in a cloud environment. DPaaS recognises the difficulties with full disk encryption and focuses on data sharing, proposing flexible data units access control lists. Despite highlighting a range of important issues related to cloud data protection, DPaaS falls short of proposing a clear implementation strategy and specific sharing mechanisms that could be used by cloud tenants. In the current paper, we address many of the concerns highlighted in [205], propose an XML-based framework to enable data sharing and describe a test implementation in the context of a cloud platform.

3 Preliminaries

Our protocol assumes that basic functionality normally provided by a *CP*, such as registration and authentication of a user, is available. Similar to [132], the active parties in our protocol are domain managers (d), virtual machines (*VM*), a secure component (*SC*) as well as a trusted third party (*TTP*). Domain managers can launch new VM instances, which can in turn create data and securely share it with other VM instances both within the same and other IaaS clouds. The proposed protocol also relies on the capabilities of the Trusted Platform Module (*TPM*) [36].

For the purposes of our protocol, each domain manager, *SC* and *TTP* has a public/private key pair (pk/sk). The private key is kept secret, while the public key is shared with the community. Furthermore, we assume that during the initialization phase, each entity obtains a certificate via the certification authority provided by the *CP*. These keys and certificates will be used to protect internal message exchanges and hence the communication between the parties is assumed to be secure. Finally, our protocol also relies on pseudorandom functions [220] – a major tool for the design of shared key cryptography protocols – to create symmetric keys.

Next, we define the main components of our protocol.

Disk encryption subsystem The disk encryption subsystem is a software or hardware component responsible for encryption and decryption of data during respectively writes or reads from a storage device. It can encrypt storage units such as whole hard drives, partitions, software RAID volumes, logical volumes, and files. For simplicity, this paper assumes a software-based disk encryption subsystem, such as *dm-crypt*, a popular open-source disk encryption subsystem which uses the Linux kernel Crypto API.

Domain Manager (d_i) Domain Managers are responsible for launching virtual machines and handling the VM instances that they create. Let $DM = \{d_1, \ldots, d_n\}$ be the set of all domain managers in our IaaS cloud. Then, the set of all VMs that each domain manager d_i owns is defined as $VM_i = \left\{vm_1^i, \ldots, vm_n^i\right\}$.

Domain (Dom_i) A domain is an abstract concept referring to a collection of data. A domain Dom_i can be created only from a domain manager which is also responsible for granting permissions to VM instances within the cloud environment. As a storage unit, a domain can be any unit supported by the disk encryption subsystem. Let $D_i = \left\{Dom_1^i, \ldots, Dom_n^i\right\}$ be the set of all domains created by a domain manager d_i.

Trusted Platform Module (*TPM*) *TPM* is a tamper-evident hardware cryptographic coprocessor which follows the specifications of the Trusted Computing Group (*TCG*) [36]. In this work, we assume that the IaaS compute hosts are equipped with a *TPM* v1.2 chip. An active *TPM* records the software state of the platform at boot time and stores it in its platform configuration registers (PCRs) as a list of hashes. *TPM* enables data protection by securely maintaining cryptographic keys, as well as though the set of functions it exposes. The *bind* and *seal* functions are particularly relevant for the proposed solution. According to [36], a message encrypted ("bound") using a particular *TPM*'s public key is decryptable only by using the private key of the same *TPM*. Sealing is a special case of the binding functionality, where the

encrypted messages produced through binding are only decryptable in a certain platform state (defined by the PCR values) to which the message is *sealed*. This ensures that an encrypted message can only be decrypted by a platform found in a certain prescribed state. We refer to [36] for a detailed description of the bind and seal operations.

Trusted Third Party (*TTP*) In this paper we assume a "trusted third party", which is trusted by the community and plays a key role in our protocol. We rely on the commonly supported proposition that a large code base normally contains a proportionally large number of vulnerabilities [221]. To reduce the code base, it is important that the *TTP* only supports the minimal necessary functionality. *TTP* is able to communicate with components deployed on compute hosts to exchange integrity attestation information, authentication tokens and cryptographic keys. In addition, *TTP* can attest platform integrity based on the integrity attestation quotes provided by the *TPM* on the respective compute hosts, as well as seal data to a trusted configuration of the hosts. Finally, *TTP* can verify the authenticity of a client as well as perform necessary cryptographic operations.

Secure Component (*SC*) *SC* is a verifiable execution module which performs confidentiality and integrity protection operations on guest VM instance data. *SC* is present on all compute hosts and acts as a mediator between the *CP* and the *TTP*. *SC* is responsible for forwarding the requests of domain managers to either the *TTP* or the disk encryption subsystem, depending on the type of request. In addition, *SC* is the only entity from which *TTP* accepts requests.

Definition 3.1 (Pseudorandom Function)**.** Let $PRF\ (K, c)$ be a family of functions[1] with two inputs, a secret key K and a content c. We say that *PRF* is a pseudorandom function iff the input-output behavior of a random instance of the family is "computationally indistinguishable" from that of a random function.

In this paper, we focus on the following problem:

Problem Statement: *A domain manager d_i, operates a set of VM instances $VM_i = \left\{vm_1^i, \ldots, vm_n^i\right\}$. In addition to that, d_i operates a set of domains $D_i = \left\{Dom_1^i, \ldots, Dom_k^i\right\}$ made available to the VM instances as storage devices. Finally, a different domain manager d_j operates a set of VMs $VM_j = \left\{vm_1^j, \ldots, vm_n^j\right\}$. We aim to create secure mechanisms that will satisfy the following requirements:*

- *Data stored in each Dom_l^i should be encrypted;*
- *Plaintext data from each Dom_l^i should be revealed only to VM instances with corresponding access privileges;*
- *Access privileges for members of VM_i to domains in D_i should be exclusively controlled by domain manager d_i;*
- *d_i should be able to share access privileges for domains in D_i to other domain managers, e.g. d_j;*

[1] A function family is a map $F : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$, where $\mathcal{K}$ is the set of keys of F, $\mathcal{D}$ is the domain of F and $\mathcal{R}$ is the range of F. The two-input function F takes a key K and an input X to return a point Y denoted by $F(K, X)$.

Adversarial Model Similar to existing works in the area, we assume that the adversary is acting under the Dolev-Yao adversarial model [211]. In this model, malicious nodes can overhear all messages and may attempt to use them to learn information that should otherwise remain private. Adversaries can also create, replay and destroy messages; however, they are not able to break any cryptographic mechanism.

The notation $E_i(.)$ will refer to the results of the application of an asymmetric encryption function that only entity i can decrypt with her private key.

4 IaaS Cloud System Model

We consider an IaaS cloud model as defined by the NIST, where an IaaS cloud provides "processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications." [64]. The model is based on a *CP* deployed on multiple server platforms. The *CP* is a distributed middleware composed of a series of *management services* on management hosts and corresponding *service agents* deployed on service hosts.

Service hosts can be dedicated to compute resources (*compute hosts* hosting virtual machines) and/or storage resources. Management services control vital aspects of the *CP* such as scheduling, networking, identity, volume and virtual machine image management. In a typical *CP* architecture, management services and service clients communicate among themselves using the advanced message queuing protocol (AMQP) based on a publish-subscribe model. The capabilities of a *CP* are exposed to *domain managers* through a set of APIs, graphical or command line interfaces. Domain managers use the functionality of the *CP* in order to operate VM instances, create storage volumes and custom network topologies using software defined networks.

The IaaS cloud is maintained by a *cloud service provider*, an organization responsible for the operation of the IaaS cloud. The cloud service provider can be either private or public. In this paper we assume a public cloud provider, with multiple domain managers sharing physical resources through a virtualization layer. On the physical compute host level, virtualization between the domain managers is ensured by the hypervisor; communication isolation is ensured though VLAN tagging (using the IEEE 801.2Q tags); *CP* level isolation relies on the authentication service, which authenticates domain managers based on their credentials.

Domain managers can create and attach block storage volumes to one or more virtual machine instances in the cloud environment. Support for storage encryption is offered by a standard disk encryption subsystem. Domain managers can also grant access rights on a certain volume to their peers.

5 Protocol Description

Our work is an extension of the protocol presented in [132] where the authors introduced the principles of "domain-based storage protection" (DBSP) in a public IaaS cloud. DBSP is based on a set of protocols that allow an IaaS client to shift the responsibility for data confidentiality and integrity to an external *TTP* – away from the IaaS provider. This approach relies on two

protocols: initial data write operation and subsequent data read and write operations. The core idea of this approach is to store information necessary to derive the decryption key for a given data volume in a header appended to the volume itself. The decryption key can only be derived by the *TTP* using the information stored at write time in the volume header and *TTP*'s own secret key. Besides withdrawing data protection responsibility from the IaaS provider, this enables a fluid migration of the IaaS client's encrypted data assets to a different IaaS provider, while maintaining trust in the same *TTP*. In addition, the approach in [132] allows to precede the release of the decryption key with a remote attestation of the platform done by the same *TTP*. Remote attestation would: *(i)* ensure that the execution platform is in a certain trusted state and *(ii)* allow the *TTP* to seal the decryption key to the trusted configuration of the host to prevent its misuse in the form of migration to other platforms or usage in a different platform configuration.

Having briefly covered the background, we proceed with a high-level overview of our protocol. A domain manager d_i launches n VM instances and a set of domains $D_i = \left\{Dom_1^i, \ldots, Dom_n^i\right\}$ that the VM instances can access to read and write data. To this end, d_i authenticates to the *CP* and requests to generate a domain Dom_k^i. The request also describes the VM instances belonging to the set VM_i that should have access to the specified domain and the respective access rights. The *CP* is responsible for creating Dom_k^i and allocating the corresponding disk space. During this process, the *SC* (part of the *CP*) contacts *TTP*, to generate a symmetric key ($K_{Dom_k^i}$[2]) that will be used to encrypt data in Dom_k^i. Following the successful creation of Dom_k^i, a domain manager must prove the right of a certain VM instance to access Dom_k^i.

In the following protocol, the participants exchange a number of messages. In order to ensure the integrity of the communication, we assume that each message is signed by the sender and the receiver can easily verify it.

5.1 Domain Sharing

One of the challenges of cloud computing is to enable users to securely administer data in a shared environment. Despite the fact that the protocol in [132] achieves protection of data in the cloud, it is considered a rudimentary work since it lacks sharing functionality. In the following paragraphs, we bridge this gap by presenting an extension of the protocol introduced in [132], which can be added to a typical *CP* to allow a domain manager to share a storage domain with other VM instances in the IaaS cloud.

Domain Registration Assume that a domain manager d_i wishes to create a domain Dom_k^i. As a first step, d_i defines the parameters needed to create the domain (e.g volume, size, name, etc.) and a description of the type of data stored in Dom_k^i. This description constitutes the metadata ($meta_k^i$) of Dom_k^i and will be used for domain discovery and data search. Upon receiving a domain creation request, the *CP* generates an XML document (Listing C.1), allocates the corresponding disk space (by e.g. creating a logical volume) and adds $meta_k^i$ to the header of the allocated volume. It also adds the domain credential that will later used by the *TTP* to the header of the allocated volume.

[2] All data in a single domain is protected with the same storage protection master key, the domain key. This key is generated by the *TTP* and cannot ever leave *TTP*'s logical perimeter.

```
1  <DomainCredential scope="create">
2     <CredentialID>cred:id</CredentialID>
3     <Timestamp>issue:time</Timestamp>
4     <DomainDescription>
5        <DomainID Encoding="xmlenc:rsa">Dom_k^i</DomainID>
6        <DomainName lang="EN">dom:name</DomainName>
7        <DomainManager Encoding="xmlenc:rsa">
8              E_TTP (d_i)
9        </DomainManager>
10       <DomainVolume Encoding="xmlenc:rsa">
11             E_di (volume : id)
12       </DomainVolume>
13       <DomainSize>E_di (disk : size)</DomainSize>
14       <Metadata>meta_k^i</Metadata>
15    </DomainDescription>
16 </DomainCredential>
```

Listing C.1: Credential specification for the creation of a new domain.

Once a domain has been created, d_i can grant access permissions to multiple VM instances. We analyse the problem of sharing a domain in the following two use cases:

A. *Grant access to Dom_k^i for a VM instance in the set VM_i:* Assume d_i intends to grant access to Dom_k^i for a VM instance vm_l^i, which is part of the set VM_i. To do this, d_i requests the launch of a new virtual machine vm_l^i and defines the access domain(s) and respective permissions for vm_l^i. More precisely, d_i generates and sends to the *CP* an XML document as shown in Listing C.2 where each encrypted element (*i.e xmlenc:rsa*) is protected with pk_{TTP} and thus *TTP* is the only one who can decrypt it. Prior to launching the VM instance with the requested domain(s) attached, *SC* checks that element *DomainDescription* matches the one stored in the header of the domain Dom_k^i. If it does, *SC* updates the XML structure of Listing C.1 by adding the element *VirtualMachine* contained in the VM instance launch request.

Once vm_l^i is launched, d_i generates a credential as shown in Listing C.3 that will be used later to prove that d_i has granted access permissions for Dom_k^i to vm_l^i.

B. *Grant access to Dom_k^i for a VM instance in the set VM_j:* Assume d_i intends to grant access to Dom_k^i to a VM instance vm_l^j, which is part of the set VM_j (operated by domain manager d_j). As we have mentioned earlier, a VM instance must receive a credential from the corresponding domain manager in order to access files in a specific domain. In this case though, the manager of Dom_k^i is not the owner of vm_l^j, so in order to grant access to vm_l^j, d_j requests a valid credential from d_i. Upon reception – if d_i accepts to give access to vm_l^j – it generates a credential as described in Listing C.2 and sends it to the *CP*. Domain manager d_i also generates a random nonce r_j and sends $E_{d_j}(r_j)$ to d_j as well as $E_{SC}(r_j)$ to the *CP*. Upon reception, d_j decrypts it with sk_{d_j}, and sends it to *CP* which validates that $D(E_{SC}(r_j)) = D\left(E_{d_j}(r_j)\right)$. Then, *SC* adds the corresponding VM to the credential of Dom_k^i as described in the previous case. More exactly, *SC* will add a nonce, the *VMID* and the access permissions (presented in Listing C.4) to the credential of Dom_k^i (XML document in Listing C.1).

5.2 Domain Access

Next, we describe the domain confidentiality protection mechanism and present the protocol to retrieve encryption keys and provide access to plain text data for authorized VM instances.

```
1  <DomainCredential scope="addVM">
2     <CredentialID>cred:id</CredentialID>
3     <Timestamp>issue:time</Timestamp>
4     <DomainDescription>
5        <DomainID>Dom_k^i</DomainID>
6        <DomainName lang="EN">dom:name</DomainName>
7        <DomainManager>manager:id</DomainManager>
8     </DomainDescription>
9     <VirtualMachine manager="d_i">
10       <VMID Encoding="xmlenc:rsa">E_TTP(vm_l^i)</VMID>
11       <Nonce Encoding="xmlenc:rsa">E_TTP(r)</Nonce>
12       <permissions Encoding="xmlenc:rsa">
13          <permission>r</permission>
14          <permission>w</permission>
15       </permissions>
16    </VirtualMachine>
17 </DomainCredential>
```

Listing C.2: Credential Specification for the addition of a VM to a domain.

We assume that vm_l^i requires access to the data in Dom_k^i. To grant access, *SC* must retrieve from *TTP* the symmetric key (K_k^i) used to confidentiality protect data in Dom_k^i. The domain manager operating vm_l^i sends a request to *CP* in order to mount Dom_k^i to the virtual machine instance; the call is forwarded to and processed by *SC*. In the request, d_i sends the previously generated credential (Listing C.3) and proves that vm_l^i has access to Dom_k^i. *SC* extracts from the header of the domain Dom_k^i a data structure (Listing C.1) that contains information about the domain and sends it to *TTP* along with the unique identifier of vm_l^i.

As a first case, we assume that no data has been stored in Dom_k^i yet, which implies that K_k^i has not been generated yet. When *TTP* receives the message from *SC*, it first decrypts $E_{TTP}\left(vm_l^i\right)$ from the XML document presented in Listing C.3 and locats the *ID* of the VM instance contained in the credential. Next, *TTP* checks if the corresponding block (*i.e* where *VMID* element is equal with vm_l^i) exists in the credential of the domain. If it does, *TTP* decrypts the metadata and checks that values match in both XML files. It then finds the permissions of vm_l^i for Dom_k^i by decrypting *permissions* from the domain credential.

Once *TTP* has validated that vm_l^i is authorized to access Dom_k^i, it performs a remote attestation of the compute host where vm^i will be launched (for simplicity, we assume that this is is also the source of the key request). The remote attestation involves obtaining a quote of the compute host's *TPM* platform configuration registers to evaluate whether the platform can be trusted. We leave out the minutiae of remote attestation and evaluation of platform trust level and refer the reader to [131].

In the event of a positive result of the *TPM* remote attestation, *TTP* generates a symmetric key (K_k^i) that encrypts data in the domain. To create the key, *TTP* generates a random nonce r_k and evaluates the following:

$$K_k^i = PRF\left(meta_k^i \| r_k, K_{TTP}\right),$$

where $meta_k^i \| r_k$ is respectively the concatenation of metadata and the random generated nonce, and K_{TTP} is a master key that does not leave the security perimeter of *TTP*. After generating the symmetric key for Dom_k^i, *TTP* seals it to the trusted configuration of the compute host (similar to the key sealing procedures already described in [131, 132]) and returns to *SC* the response shown in Listing C.5.

```
1  <DomainCredential scope="accessDomain">
2     <CredentialID>cred:id</CredentialID>
3     <Timestamp>issue:time</Timestamp>
4     <DomainDescription>
5        <DomainID Encoding="xmlenc:rsa">dom:id</DomainID>
6        <DomainName lang="EN">dom:name</DomainName>
7        <DomainManager>manager:id</DomainManager>
8        <Metadata Encoding="xmlenc:rsa">E_TTP(meta_k^i)</Metadata>
9     </DomainDescription>
10    <VirtualMachine manager="d_i">
11       <VMID Encoding="xmlenc:rsa">E_TTP(vm_l^i)</VMID>
12       <Nonce Encoding="xmlenc:rsa">E_TTP(r)</Nonce>
13    </VirtualMachine>
14 </DomainCredential>
```

Listing C.3: Credential for presentation of granted permissions for a domain.

Upon receiving the message, *SC* first decrypts $E_{SC}\left(vm_l^i\right)$ and checks if the request was sent from the VM instance contained in the response. If it was, *SC* calls the local *TPM* to unseal the key which – if the compute host remained in the earlier trusted state – reveals K_k^i. *SC* then uses it as input to the disk encryption subsystem on the compute host where vm_l^i is running. The disk encryption subsystem seamlessly decrypts the mounted volume hosting Dom_k^i. Next, the volume containing Dom_k^i is mounted as a disk device on vm_l^i – with read-write or read-only rights, depending on the permissions granted by the domain owner.

The case where K_k^i has already been generated is similar, with the only difference that to recalculate K_k^i, *TTP* will have to decrypt $E_{TTP}(r_k)$ contained in the updated metadata and use it as an input to the pseudorandom function.

5.3 Revocation

There are cases when credentials of a VM may need to be revoked if a VM instance misbehaved, lost access rights to a domain, or permissions have been changed. In this section we describe the mechanism to change or revoke the permissions of a VM instance for a specific domain.

Following our previous scenario, we assume that d_i wants to change the access rights of vm_l^i for the domain Dom_k^i. We analyse the following two scenarios for d_i:

A. *Prevent vm_l^i from accessing Dom_k^i:* Assume d_i wants to completely remove vm_l^i from the list of VMs that are authorized to access Dom_k^i. First, d_i generates the XML file shown in Listing C.6 and sends it to *CP*, which forwards the request to the *SC* on one of the host platforms. Upon reception, *SC* extracts the credential for Dom_k^i from the header of the volume and sends it to *TTP* along with the XML received from d_i. *TTP* decrypts $E_{TTP}\left(vm_l^i\right)$ and finds the *ID* of the VM that should remove its access rights. Then, *TTP* finds the corresponding block in the XML that contains all the VM instances that have access to Dom_k^i and removes it. Finally, *TTP* returns to *SC* an updated XML document which does not contain vm_l^i and *SC* updates the header of Dom_k^i with the fresh credential.

B. *Change permissions of vm_l^i on Dom_k^i:* In this case we assume that d_i intends to just change the permission for vm_l^i 'read-write' to 'read'. The procedure that is followed is identical to the one in scenario **A**. d_i generates a new credential for vm_l^i (Listing C.7) and sends it to *TTP* via *CP*. Additionally, *SC* sends to *TTP* the credential of the domain that is stored in the header of the volume. *TTP* follows the same steps in order to update the credential of Dom_k^i. Following

```
1  <VirtualMachine manager="d_i">
2      <VMID Encoding="xmlenc:rsa">E_TTP (vm_l^j)</VMID>
3      <Nonce Encoding="xmlenc:rsa">E_TTP (r')</Nonce>
4      <permissions Encoding="xmlenc:rsa">
5          <permission>r</permission>
6      </permissions>
7  </VirtualMachine>
```

Listing C.4: Credential Specification for the addition of a VM to a domain.

the successful update of the domain credential, *SC* sends the fresh credential to d_i who can use it in the future in order to prove that vm_l^i is authorized to access the corresponding domain under certain permissions.

In both cases A and B, d_i has the option to receive from the *TTP* a confirmation that the permissions for vm_l^i have indeed been withdrawn or modified. To do this, prior to the request d_i generates a random nonce r_{rev}, encrypts it with *TTP* and sends it along with the credential of vm_l^i. Upon reception *TTP* – apart from altering the permissions of the corresponding VM – decrypts $E_{TTP}(r_{rev})$ and returns to d_i $H(r_{rev}||vm_l^i))$[3]. Given that by definition the *TTP* will not deviate from the protocol, the returned hash is an implicit confirmation of the fact that TTP has received the update request and has modified the credential accordingly.

6 Security Analysis

In this section, we analyse the behaviour of our protocol in several attack scenarios. In all of the attack scenarios, we assume that the involved parties follow the Dolev-Yao adversarial model [211] and can overhear all messages and may attempt to use them in order to learn information that otherwise should remain private or gain access to domains that are not authorized to.

Unauthhorized access to a domain: Assume that a malicious domain manager d_m attempts to gain unauthorized access to a domain Dom_k^i for a VM instance vm_l^m. To do so, the domain manager will have to prove that she owns a credential for accessing Dom_k^i. The domain manager self-generates a credential and presents it to the *CP* in order to gain access to Dom_k^i (as shown in Listing C.3). This can be easily done since the encrypted information contained in a credential is mainly generated using the public key of *TTP*, which is also responsible for validating the correctness of the credential. As described in Section 5, *SC* retrieves the corresponding metadata from the header of Dom_k^i and forwards both artefacts to *TTP*. Upon reception, *TTP* verifies the correctness of the credential received from d_m. To this end, *TTP* decrypts the information contained in both artefacts and finds out that the *ID* of vm_l^m is not in the list of the authorized VM instances for the domain Dom_k^i. Thus, this attack cannot be launched.

Using a valid credential from another domain manager: In such a scenario we assume that a malicious domain manager d_m attempts to gain unauthorized access to a domain Dom_k^i for vm_k^m by providing a valid credential that belongs to another VM instance, i.e. effectively impersonating the righteous domain manager. We assume that d_m gets a valid credential for Dom_k^i that was created for vm_k^i. This can be done in two different ways: either d_m can intercept a message in which d_i sends the credential to *SC* in order to access Dom_k^i; or – if we assume that

[3] $H(.)$ is a secure cryptographic hash function such as *SHA3*

```
<response>
  <DomainKey Encoding="xmlenc:rsa">
    E_dmcrypt(K_k^s)
  </DomainKey>
  <VMID>E_SC(vm_l^i)</VMID>
  <Metadata>meta_k^i || E_TTP(r_k)</Metadata>
  <permissions>
    <permission>r</permission>
    <permission>w</permission>
  </permissions>
</response>
```

Listing C.5: Response of *TTP* after the generation of the domain symmetric key.

```
<VMCredential scope="Revoke">
  <CredentialID>cred:id</CredentialID>
  <Timestamp>issue:time</Timestamp>
  <DomainDescription>
    <DomainID Encoding="xmlenc:rsa">dom:id</DomainID>
    <DomainName lang="EN">dom:name</DomainName>
    <DomainManager>manager:id</DomainManager>
  </DomainDescription>
  <VirtualMachine manager="d_i">
    <VMID Encoding="xmlenc:rsa">E_TTP(vm_l^i)</VMID>
  </VirtualMachine>
</VMCredential>
```

Listing C.6: Request to revoke the credential of a VM.

```
<VMCredential scope="UpdatePermissions">
  <CredentialID>cred:id</CredentialID>
  <Timestamp>issue:time</Timestamp>
  <DomainDescription>
    <DomainID Encoding="xmlenc:rsa">dom:id</DomainID>
    <DomainName lang="EN">dom:name</DomainName>
    <DomainManager>manager:id</DomainManager>
  </DomainDescription>
  <VirtualMachine manager="d_i">
    <VMID Encoding="xmlenc:rsa">E_TTP(vm_l^i)</VMID>
    <Nonce Encoding="xmlenc:rsa">E_TTP(r')</Nonce>
    <permissions Encoding="xmlenc:rsa">
      <permission>r</permission>
    </permissions>
  </VirtualMachine>
</VMCredential>
```

Listing C.7: Request for altering permissions of vm_l^i on domain Dom_k^i.

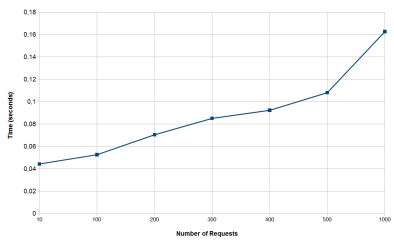
d_i is also acting maliciously – d_i can cooperate with d_m, and reveal to d_m the credential created for vm_l^i. In both cases, d_m will be able to convince *TTP* about the validity of the credential. Thus, *TTP* will first *attest the trusted configuration* of the host where the virtual machine vm_l^m will reside, then will calculate the domain key and will send back to *SC* the metadata showed in Listing C.5. Upon reception, *SC* first decrypts $E_{SC}\left(vm_k^i\right)$ and checks whether the domain manager requested to grant access to the VM instance stated in the response of the *TTP*. In the above attack scenario, *SC* will drop the request since the *VMID* received from *TTP* does not correspond to the VM instance for which access to Dom_k^i was requested. We can conclude that such an attack would only be possible if malicious domain managers can change their identity.

Using *remote TPM attestation* and the *TPM seal* operation, we obtain the confidence that *SC*

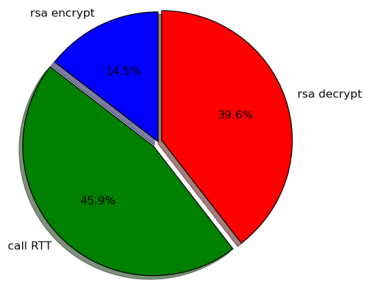
will act according to the protocol and will verify that the requesting VM instance identifier matches the VM instance identifier authorized in the response obtained from the *TTP*. The domain decryption key is only made available to the target compute host with a trusted platform configuration, and can not be accessed in plain text once the compute host changes its platform configuration.

7 Experimental Results

In order to measure the performance of the protocol, we have implemented the *SC* as a client and a server application serving the role of the *TTP*. Our implementation follows the protocol: *SC* creates a request for an encryption key and sends it to *TTP*, which derives the encryption key and returns it in encrypted form together with the meta data.



(a)



(b)

Figure C.1: Performance evaluation: (a) Time required by *TTP* to process a request and to generate a key for a domain; (b) Proportion of execution time spent in functions during a key request.

The experiments aimed to analyze two main performance metrics: processing time and communication overhead. To this end, we ran several experiments, in order to measure the time to request a new encryption key, the duration of the most computation-intensive or network-intensive operations, as well as to measure the performance of *TTP*.

In the first phase of our experiments we measured the performance of *TTP* when serving multiple parallel requests from *SC*. We tested the time *TTP* needed in order to perform *encryption/decryption* operations, *generate a domain key* as well as to *parse an XML* for 10 to 1000 parallel requests. For encryption and decryption, we used the RSA cryptosystem with a key length of 1024 bits. Figure C.1a illustrates the results in seconds as a function of the number of requests. As seen from the graph, the required processing time is negligible and does not constitute any real burden to the functionality of the *CP*. We we have found that, on average, the time needed for *TTP* to successfully respond to *SC* when receiving 1000 parallel requests is approximately equal to 0.16 seconds.

In the second phase of our experiments, we measured the communication delay for a single request sent to *TTP* by *SC* (with a sample of 1000 sequential requests), as well as the impact of domain key requests on the duration of the VM instance launch. Table C.1 and Figure C.1b show respectively the absolute and relative execution times for operations performed by the

Table C.1: Execution times (in seconds) for some functions in the secure component, 1000 requests.

Cumulative time	Per call	Function
3.300	0.001	RSA Encryption
10.445	0.010	Call RTT
9.001	0.009	RSA Decryption
24.974	0.025	Total Execution Time

SC to obtain an encryption key. Figure C.1b indicates that most of the execution time is spent on the decryption of the domain key and the call round trip time (call RTT). The absolute duration of the encryption key request is on average 0.025 seconds.

In addition to that, we deployed an IaaS cluster using version "Havana" of OpenStack, a popular *CP* in order to measure the time needed for a VM to be launched. This time would then be compared with the time needed for the generation of a domain key. As the process of key generation for the domain of a new VM instance is taking place in parallel with the VM launch, this comparison would be a good metric to see whether our protocol affects the performance of the *CP* or not. According to our measurements, the average time to launch a VM instance is 20.57 seconds while the average time for a domain key request is 0.025 seconds. Taking into consideration the fact that a domain key request will usually take place during the launch of a VM, our protocol does not affect the overall performance of the *CP*.

8 Conclusion

In this paper we have considered the problem of secure storage in IaaS environments. More precisely, we proposed a protocol that ensures confidentiality and integrity protection of stored information in a cloud environment. Furthermore, we presented an XML-based language framework that allows the clients of IaaS clouds to securely share their data and assign different access rights to users. The analysis was coupled with experimental results which showed that the proposed language adds only a reasonable overhead to the operation of a cloud management platform. In our future work, we aim to improve the protocol and reduce the trust base by removing the need for a TTP. While this may affect the performance of the protocol, it would allow us to consider more complex attack scenarios which better reflect the complexity of information flow in IaaS clouds.

Paper D

Trusted Geolocation-Aware Data Placement in Infrastructure Clouds

Nicolae Paladi, Mudassar Aslam and Christian Gehrmann

Abstract

Data geolocation in the cloud is becoming an increasingly pressing problem, aggravated by incompatible legislation in different jurisdictions and compliance requirements of data owners. In this work we present a mechanism allowing cloud users to control the geographical location of their data, stored or processed in plaintext on the premises of Infrastructure-as-a-Service cloud providers. We use trusted computing principles and remote attestation to establish platform state. We enable cloud users to confine plaintext data exclusively to the jurisdictions they specify, by sealing decryption keys used to obtain plaintext data to the combination of cloud host geolocation and platform state. We provide a detailed description of the implementation as well as performance measurements on an open source cloud infrastructure platform using commodity hardware.

1 Introduction

Reliance on third-party providers for cloud storage and computing decouples data management from both data ownership and responsibility for correct data usage. A data owner loses control over the geographical placement of data once it is transferred to a cloud provider and earlier agreements become the only available tool to manage future data placement. In some cases, transfer of sensitive data to other countries is illegal and while agreements can be a basis for compensation, they can only help *post factum*, when the damage is already done.

The physical location of data storage and processing in cloud environments matters for several reasons: tax rates may differ based on where a transaction is conducted (rather than where the entity is registered); compliance rules or privacy laws may require that certain categories of data are not stored or processed in a different jurisdiction; finally, organizations with geographically distributed field offices might conduct operations – such as certain types research – which are illegal in some countries (e.g. stem cell research [222]). The above reasons are relevant to both data processing and storage. In [147], the authors clarify the importance of geolocation assurance mechanisms in cloud storage services. One of the central arguments is that data geolocation affects its confidentiality and privacy status. Similarly, [13, 149, 150, 223, 224] mention concerns – such as compliance requirements – about the geolocation of data in cloud computing environments in general and Infrastructure-as-a-Service (IaaS) in particular.

Calls for legal mitigation of issues related to data location in the cloud involve regulating cloud storage services through binding inter-government regulations [224]. However, this in only part of the solution, since agreements can be covertly breached – sometimes long before this fact is revealed to other parties.

Recent innovations in data center design – such as 'modular data centres' – improve the mobility of data centres. A central component of modular data centres are the standardized ISO 6346 weatherproof containers, capable of housing thousands of servers and necessary related components. This allows data center modules to be easily moved across large distances using standard transportation means. The idea was originally described in [225] and widely developed since then [226, 227]. This approach differs from the traditional, static data centre architecture and along with advances in distributed storage systems architecture highlights the necessity to consider the geolocation of hosts when transferring data to a cloud storage provider.

Any practical solution for protection of data in cloud environments must consider its impact on functionality – a major driving force for adoption of cloud computing – such that e.g. distributed data processing capabilities are minimally affected. We propose a solution that combines *geolocation data* and *trusted computing* principles to allow data to be processed and transferred in plaintext only to geolocations approved by the data owner, without affecting data processing capabilities of distributed data stores. We present a prototype implementation based on Swift, a known distributed object storage system [228].

1.1 Contribution

Our contribution is as follows. First, we describe a protocol to securely store location information on cloud host platforms and later use this information to ensure that data is only available in plaintext on platforms that are placed in geolocations sanctioned by the data owner. Second, we provide a detailed implementation description of the above protocol, based on a popular cloud operating system and a known distributed object storage. Finally, we provide a security analysis of the chain of trust that allows to seal data to a given platform state extended to include the geolocation of the platform.

1.2 Organization

In Section 2 we provide an overview of the related work that addresses data geolocation and data protection in cloud storage. We continue by defining the important terms in Section 3 and presenting the system model in Section 4. We present the protocol in Section 5, the detailed implementation in Section 6 and the security analysis in Section 7. The prototype evaluation results are presented in Section 8, followed by a conclusion in Section 9.

2 Background and Related Work

Related work on the topic has predominantly focused on establishing the fact that a certain piece of data *is* stored in a certain location, ignoring any potential replicas.

The problem of "data sovereignty", defined as "establishing data location at a granularity sufficient for placing it within the borders of a nation-state" was first introduced in [149]. The proposed solution combines provable data possession (PDP) schemes with a network-delay based protocol for data geolocation, in order to get a proof of the fact that the data is located in the respective data centre. This early paper lacks a specific adversary model and describes only a high-level solution.

In a follow-up, Gondree and Peterson propose a "constraints-based data geolocation" solution to determine the location of data and "bind" it to specific locations[1] [150]. The adversary model assumes an economically rational adversary aiming to reduce costs through data migration in spite of contractual agreements. The protocol assumes an initial model building stage, where landmarks (L) throughout the analysed geographical region each build a latency-distance estimation model. Using this model, each landmark issues PDP challenges to the storage and generates a circular constraint of a radius centred on L. The geolocation step of the protocol uses the intersection of geolocation constraints to determine the region where the data resides. The solution suffers from a series of limitations: it requires a set of landmarks close to the data centres of the cloud service provider; incorrectly assumes that the cloud service provider does not have dedicated communication channels between its data centres and finally, does not discuss location-based storage protection and rather just verifies that a certain file *is* placed on a given host. The authors of [229] outline some ideas regarding the use of Trusted Platform Modules (TPM) on server platforms in the context of data location in cloud networks. The solution assumes that the identity of the server's TPM is stored along with the server's geographical position by the Certificate Authority and retrieved when needed. The solution further assumes a "Location verification and integrity check" module implemented in a hypervisor and suggests a two-phase protocol: the *initialization phase* includes remote attestation of the host and verification of its location; the *verification phase* includes a protocol to confirm the identity of the host based on communication with the TPM deployed on it. This solution is similar to our approach in the use of TPM as a hardware root of trust; however, it assumes that verification of the location is done through administrative methods, i.e. costly physical visit of the facilities. Furthermore, the paper does not describe any implementation results.

The National Institute of Standards and Technology (NIST) has described a proof of concept

[1] Binding is here used in the sense of detecting occurrences of data misplacement, rather than data binding in the meaning common in trusted computing

implementation for trusted geolocation in the cloud [13]. The proof of concept uses a combination of trusted computing, Intel Trusted Execution Technology (TXT) and a set of manual audit steps to verify and apply data location policies. The protocol establishes an automated hardware root of trust – defined as "inherently trusted combination of hardware and firmware that maintains the integrity of the geolocation information and the platform", in order to manage geolocation restrictions for hosts within an infrastructure cloud platform. The solution assumes that geolocation information is provisioned to the platform via an out-of-band mechanism and – along with platform metadata – stored in the TPM. This information is later accessed in order to verify the integrity of the host and the location of the platform. Similar to both our approach and [229], the use of TPM for platform identification offers a reliable, hardware-based root of trust. The solution in [13] assumes remote platform attestation – including location data – in order to establish the trustworthiness of the platform, which is a significant improvement compared to earlier work. However, we see several limitations of this approach and address them in this paper.
First, the protocol in [13] does not provide any cryptographic protection of data; rather, data placement is scheduled based on placement policies and thus data confidentiality depends on the correctness of the location policy. We believe this approach does not protect data from accidental or malicious policy misconfiguration, in which case plaintext data could be scheduled to an untrusted host. We address this by requiring that all uploaded client data is confidentiality and integrity protected and is only stored in plaintext *in the jurisdictions defined by the user*, a property achieved by performing remote attestation of the storage hosts and sealing the confidentiality and integrity protection keys to the platforms with a correct configuration.
Second, [13] assumes out-of-band provisioning of geolocation data to the storage hosts, without further clarification of the data format and delivery mechanisms. In this paper, we provide a detailed description of the format of data required for the geolocation of storage hosts in an infrastructure cloud. Furthermore, we address the question of secure out-of-band geolocation data delivery to storage hosts and also suggest a *complementary* geolocation acquisition model using dedicated GPS receivers.
Third, [13] does not describe a mechanism to re-provision geolocation tags and thus does not hold in the case of modular data centres mentioned in § 1. Our proposed solution – which assumes a distributed geolocation information acquisition model – holds even in the cases when data hosts are relocated.

In [132] the authors discuss principles of domain-based storage protection in public infrastructure clouds. The principles outlined in the paper associate all objects stored in the IaaS cloud with explicit *storage domains*. A storage domain in this context corresponds to an organization or administrative unit that uses public cloud services (including the storage service) offered by the provider. All data in a single domain is protected with the same storage protection master key, the *domain key*. The paper further suggests that at guest VM launch, it is securely associated with a particular storage domain throughout its lifetime. Keys used for data encryption, decryption, integrity protection and verification in a single domain are derived by an external, trusted third party (*TTP*). We extend this protocol to include information about the geographical placement of data. We redefine the concept of "administrative domain" in [132] to also include a certain geographical area corresponding to a jurisdiction.
Use of GPS signals in the context of data centres has been described in [230], where GPS and atomic clocks are used for time synchronization in order to implement externally-consistent distributed transactions. Besides addressing the limitations of the above papers, our solution discusses cloud data storage protection including *replicas* of the data scattered throughout the distributed data store, something which – to the best of our knowledge – has not been done earlier.

3 Preliminaries

3.1 Definitions

IaaS (cloud) platform (IP)

We assume an IaaS platform model as defined by NIST in [64], which offers "processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications"; according to the same definition, users do not have control over the underlying infrastructure. IaaS platforms in this paper are assumed to include a large data store distributed over several data centres in distinct geographical areas.

User (U)

Users are capable to access (read and write) data objects in the cloud data store. Let $U = \{u_1, \ldots, u_n\}$ be the set of all users of a certain IP. Then, the set of all data objects that a certain user u_1 owns is denoted $f_1 = \left\{f_1^1, \ldots, f_n^1\right\}$.

Cloud Service Provider (*CSP*)

We refer to a *CSP* as an entity that operates an IP and makes it available for *users*. The *CSP* includes both the case when the respective entity owns and physically manages its data centres and the case when an IP is deployed on computing resources provided by a third party supplier. The IP operated by the *CSP* may be deployed throughout arbitrarily many data centres.

Geolocation (L)

We refer to a *geolocation cell* L as a bounding area (e.g. country, region, territory, etc.) defined by a set of location points represented by their latitude and longitude ($l_i = lat_i, lon_i$) such that $\mathsf{L} = \{l_1, l_2, ...l_n\}$. Each l_i represents the location of an IP in the data centre; every data centre is associated with at most one L and no two geolocation cells overlap.

Jurisdiction (J)

We refer to a jurisdiction as "the territory or sphere of activity over which the legal authority of a court or other institution extends" [231]. Let J_i, J_j be two jurisdictions with incompatible data protection regulations. Consider a user u_1 that operates on privacy-sensitive data and uses the services of a *CSP* with data centres present in both J_i and J_j. For compliance reasons, u_1 may only process the data in J_i and faces penalties if data is processed or stored in plaintext in J_j[2]. A valid jurisdiction is a non-empty set of Ls.

[2]Operating on encrypted text currently allows an impractically restricted set of operations [232]

Trusted Platform Module (*TPM*)

A *TPM* is a tamper-evident hardware cryptographic coprocessor built according to the specifications of the Trusted Computing Group [36]. In this work, we assume that all IaaS hosts underlying the IP are equipped with a *TPM* v1.2 chip. An active *TPM* records the software state of the platform at boot time and stores it in its platform configuration registers (PCRs) as a list of hashes. A *TPM* enables data protection by securely maintaining cryptographic keys, as well as through the set of functions it exposes. The *bind* and *seal* functions are particularly relevant for the proposed solution. According to [36], a message encrypted ("bound") using a particular *TPM*'s public key is decryptable only by using the private key of the same *TPM*. Sealing is a special case of binding, where the encrypted messages produced through binding are only decryptable in a certain platform state (defined by the PCR values). This ensures that an encrypted message can only be decrypted by a platform found in a certain prescribed state. We refer to [36] for a detailed coverage of the bind and seal operations.

Trusted Third Party (*TTP*)

The *TTP* is an entity which is trusted by the community and plays a key role in our protocol. The *TTP* is able to communicate with components deployed on compute hosts to exchange integrity attestation information, authentication tokens and cryptographic keys. In addition, the *TTP* can attest platform integrity based on the integrity attestation quotes provided by the *TPM* on the respective compute hosts, as well as seal data to a trusted configuration of the hosts. Finally, the *TTP* can verify the authenticity of a client as well as perform necessary cryptographic operations.

Trusted Platform (*TP*)

In this paper, we define trusted platforms as server platforms the integrity and trusted state of which has been attested by the *TTP*. The trusted platforms of an IP comprise the *Trusted Computing Pool* (T), introduced in [13], that is the collection of trusted platforms in a certain IaaS cloud platform.

3.2 Adversary model

We share the adversary model with [131, 180] which assume that privileged access rights can be maliciously used by CSP remote system administrators (A_r). This scenario assumes that A_r can log in remotely to any host of the CSP and obtain root access. However, in this model A_r does not have physical access to the hosts. We add a geolocation aspect to the security model: u_1 requires assurance that her data is not stored or processed in plaintext outside jurisdiction J_i. The *CSP* may experience intermittent errors and has an incentive to optimize costs by placing or processing data in a different jurisdiction, e.g. J_j. We explicitly exclude Denial-of-Service attacks from our model, since we assume an economically rational CSP interested in maximizing its profits by continuing to provide services to users.

3.3 Problem Statement

Assume an authorized user u_1 writes a file f_1 to the storage provided by *CSP*. A trusted distributed storage system shall then satisfy the following properties:

1. The file f_1, as well as its replicas, must only be stored and processed in plaintext in the set of jurisdictions J_i defined by u_1.
2. The allowed jurisdictions J_i must be specified once, when f_1 is first written to the distributed storage. It shall be impossible for an adversary to subsequently change the association between f_1 and the set of allowed jurisdictions.
3. Let f_1' be a file derived from a processing operation on file f_1. The system shall make sure that f_1' inherits all the jurisdiction restrictions from f_1;

4 System Model

We consider a public IP managed by a *CSP* where multiple users lease processing and storage capacity. To benefit from properties such as increased availability, scale flexibility and use of distributed data processing algorithms, data is stored in distributed data stores. This is a common system model in modern infrastructures (including infrastructure clouds) that deal with massive amounts of data and allow data to be reliably stored, replicated and retrieved within a very short time. Examples of such systems are Google BigTable [233], Amazon Dynamo [85], Windows Azure [88], etc. Current distributed data stores store redundant replicas (often *eventually synchonized*) of data on different hosts, thus offering scalability and intra-data center resilience to hardware failures. From a geographical point of view, a distributed storage is either deployed within one data center (and hence in one jurisdiction), or spans several data centres (and possibly several jurisdictions). In the latter case, in order to separate data that is subject to conflicting regulations, users may choose to store the data in two or more distinct IaaS platforms IP_1 and IP_2 – as for example in the case of Amazon Govcloud [23]. However, this restricts geographic redundancy and reduces service availability guarantees. Another possibility is to deploy distributed storage systems across data centres – there are efforts towards this both in academic research ([234–236]) and industry implementation[3]. Emerging capabilities of distributed data stores add geographical redundancy, such that the data store is deployed across geographically distinct data centres and offers *inter-data center* redundancy on a global scale while maintaining the *eventual* synchrony of data. For simplicity, we assume a specific subtype of distributed data stores, namely distributed *object storages* depicted in Fig. D.1 and described in § 4.1.

According to the model, the domain of the *CSP* includes the IaaS cloud platform components, the hypervisor, as well as the underlying hardware.

[3]See for example issue HDSF-1432, discussing implementation of HDSF deployment across data centres, *https://issues.apache.org/jira/browse/HDFS-1432*

4.1 Distributed object storage

We assume an object storage capable of storing binary objects (similar to Amazon S3) which can be geographically distributed across multiple data centres[4]. Below, we provide a simplified description of some important components of a distributed object storage (see Fig. D.1).

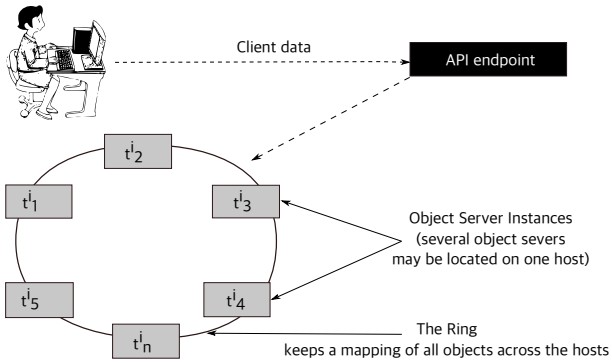


Figure D.1: Sketch of a distributed object storage

The **API endpoint** is the public API of the object storage. Upon each request, the proxy server looks up the location of the accounts or objects in *the ring*, routes requests and performs error handling.

The ring is a logical structure implemented a distributed hash table that maps the names of stored entities to their physical location. Rings maintain the mapping using *zones*, *devices*, *partitions* and *replicas*. Partitions have a *replication degree n* and their locations are stored in the mapping maintained by the ring. Data can be isolated using the zones of the ring, where each zone can be a data center, switch, cabinet, server or drive. In this model, we assume that each zone is a geographically distinct data centre.

Object server is a simple binary large object storage that can *store, retrieve* and *delete* objects on local devices. The object meta data is stored in the *extended file attributes* (otherwise known as "xattrs").

Replicators maintain state consistency in the face of network or node failures. This is done by comparing data with each remote copy to ensure freshness. Replication updates are push-based (rsync file replication to peers).

5 Protocol Description

We describe a mechanism used by IP hosts to acquire their geographical location – mapped to a geolocation cell (L) – and securely store it in TPM PCR 15. Next, we present a protocol that allows the data owner to specify the list of geolocation cells where data is stored (or processed)

[4]While this is a fairly recent development, such distributed object storages already exist; see for example `https://github.com/openstack/swift/blob/master/CHANGELOG`

in plaintext (Fig. D.3), relying on the *TPM* functionality to seal the data decryption key to the state of PCRs `0-7, 15`.

5.1 Geolocation

The proposed solution is based on sealing user data to a set of geolocation cells, making plaintext data only available to hosts physically placed in the jurisdiction authorized by the user. This requires *extending* [36] the platform geolocation cell value L to a dedicated TPM register during platform boot (we use PCR `15`). The geolocation cell value is obtained through *reverse geocoding*, which is the process of matching a location point with an administrative unit, e.g. country, region, municipality, etc. For a detailed review of the concept, see [237] [5].

Listing D.1: Sample reverse geocoding result

```
<reversegeocode timestamp="Thu, 22 May 14 08:17:17 +0000">
  <addressparts>
    <suburb>Kista</suburb>
    <city>Stockholm</city>
    <county>Stockholms lan</county>
    <state>Stockholms lan</state>
    <country>Sweden</country>
    <country_code>se</country_code>
  </addressparts>
</reversegeocode>
```

In our example, reverse geocoding maps any given location point l_i to a geolocation cell L. One approach to supply geolocation cell information to the platform is through out-of-band provisioning at an earlier stage and store it in the TPM, as suggested in [13]. However, such out-of-band provisioning can be prone to administrator error or misuse, therefore we propose to report L to the TPM through an isolated daemon process (D), and introduce two models for such a daemon to obtain L. The first model (presented in Fig. D.2) reuses the concept of "geolocation master nodes" described in [230], where certain hosts are equipped with GPS receivers and dedicated antennas, and are physically separated to reduce the effect of antenna failures, radio interference and spoofing. We assume that the master geolocaton node is previously attested and therefore trusted. The geolocation master node identifies its location point l_i and uses a local *geocoding database* to map l_i to a geolocation cell L. At boot time, D running on the storage hosts obtains L from the geolocation master and stores it in the TPM register. Note that this approach does not rule out pre-configuring geolocation cell information out-of-band, as suggested in [13], but reduces the amount of administrative operations required on IP and eliminates a potential attack vector. In the second model, the daemon process D running on the storage host, uses a navigation device natively attached to the host's motherboard to obtain its location point l_i; next, the daemon process D uses a local geocoding database to map l_i to a geolocation cell L; finally, it extends L into the TPM register (see § 3.1). While the feasibility of this model depends on the particular data centre architecture, it *complements* the approach described in model 1 without additional assumptions. We have chosen this latter approach for the rest of the paper, given the wider applications it enables for mobile platforms equipped with a hardware root of trust. The functionality of the location reporting D is explained below and presented in Algorithm 1.

[5] A sample call to the OpenSteet Map API – `http://nominatim.openstreetmap.org/reverse?format=xml&lat=59.406318&lon=17.947&zoom=12&addressdetails=1` – produces the reply presented in Fig. D.1

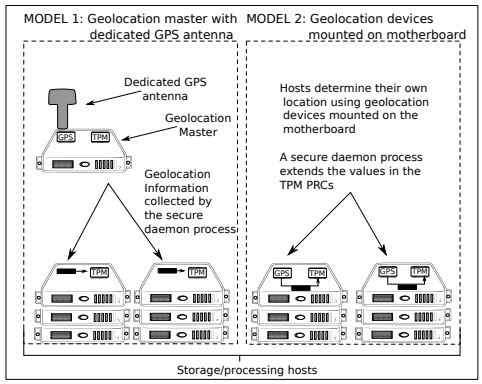


Figure D.2: Geolocation cell reporting to TPM

Data: Navigation Device Port
Result: Geolocation cell value hash in TPM PCR `15`
initialization: enable GPS, configure GPS Port;
call:fork() to daemonize;
while *satellites_tracked* < 4 **do**
  get GPGGA data from Navigation Device;
end
call:geolocationcell a $\leftarrow$ *ReverseGeocode*(*lat*, *lon*);
call:$H_a \leftarrow$ *SHA1(geolocationcell a)*;
call:$H_{exp} \leftarrow$ *Rebuild_Chain*(H_a, H_{def});
call:$H_{cur} \leftarrow$ *TPM_ReadPCR*(15);
if $H_{cur} \neq H_{exp}$ **then**
  call:TPM_Extend (H_a);
end
sleep(time);

Algorithm 1: Location Reporting Daemon

In the first step of the protocol, the *TTP* uses the PCR aggregate of a particular trusted host platform TP_1 to create an asymmetric TPM key pair *SealKey*, which is used to `seal` data to TP_1 in geolocation cell L_A. Consequently, TPM only releases the private *SealKey* if the current PCR values of TP_1 match the ones specified in the PCR aggregate used in the key creation. For performance reasons, a session key K – encrypted with the *SealKey* – is used for data encryption and decryption operations.

The runtime state of the platform with respect to its identity TP_1 and location L_A is represented by PCR `0-7` and PCR `15` respectively. These PCRs are populated with values at platform boot time. While the platform state of TP_1 is reported by the TCG-compliant BIOS and the bootloader in PCR `0-7`, we implement a component D that runs as a daemon process to report the platform *location* L_A to PCR `15`. Component D is built with minimal functionality: read the navigation device port for location data and extend it to PCR `15`. The navigation device is enabled at boot time and immediately starts tracking satellites. After a valid location fix is found and the minimum number of required satellites are located[6], D translates – through reverse geocoding – the location point l_i to the corresponding geolocation cell (i.e. L_a).

Next, the geolocation cell value verification follows. Denote the default value of PCR `15` at

[6]A connection to 3 satellites suffices to provide a location point on Earth but a minimum of 4 satellites provides better accuracy.

boot time by H_{def}[7]. D computes the SHA-1[8] hash (H_a) of L_a and rebuilds the hash chain for PCR 15: H_{def} extended by H_a yields the expected value for PCR 15, denoted $H_{exp} = H_{def}||H_a$. Next, D compares H_{exp} with H_{cur}, the current value in PCR 15. If $H_{exp} \neq H_{cur}$, then H_a supersedes H_{cur} and is extended to PCR 15; otherwise, no action is taken. D runs as a privileged process and performs this operation recurrently. As a result, during the first check PCR 15 will be extended with the hash value H_a of the current geolocation cell L_a and will maintain this value unless the platform is relocated to a different geolocation cell, L_b. In that case, L_b results in a new hash value H_b which supersedes H_{cur} and – according to the procedure above – is extended to PCR 15, making *SealKey* unavailable on the respective platform.

Thus, data encrypted with *SealKey* is only available in plaintext to TP_1 located in L_a, i.e. preventing access to plaintext data to hosts located outside the authorized jurisdictions. *SealKey* remains available in the face of restarts, as long as the host is rebooted in the same, trusted state and geolocation cell. Implementation details are provided in §6.

5.2 Storage Protection Protocol

We propose the following protocol to ensure that data is available for processing and storage in plaintext only on storage hosts deployed in a user-approved jurisdiction. A high-level model of the protocol is depicted in Fig. D.4 and a detailed message flow is presented in Fig. D.3. For the purposes of the protocol, we assume that the user u_1 knows the public key of the TTP. We further assume that u_1 can generate a high-entropy symmetric key K and encrypt own data prior to uploading them to the distributed object storage. Considering the adversary model described in § 3.2, we do not explicitly include data integrity protection in the following protocol. However, if integrity of the data is a requirement, the protocol can be easily extended to include it. Finally, we assume that encrypted data is indistinguishable from random noise and encrypted replicas of data may be stored without any legal consequences in any jurisdiction.

Protocol description follows; corresponding steps of the protocol and the message flow are also presented in Fig. D.3.

1. User u_1 uploads though the API endpoint the encrypted data (denoted by $E(P, \mathsf{K})$), along with a signed user token containing a description of the location policy constraints – represented by a list of geolocation cells, as shown in Listing D.2 – and the symmetric key K, encrypted with TTPs public key. Thus, K can be accessed in plaintext only by the TTP (and u_1 who has generated it).
2. The API endpoint applies the provided location policy to determine the set of hosts that are allowed – according to the policy specified by u_1 in the token – to store (or process) the uploaded data in plaintext. Denote this set by $\mathsf{T}_{\mathsf{J}_i}$.
3. The API endpoint writes $E(P, \mathsf{K})$ to the hosts in $\mathsf{T}_{\mathsf{J}_i}$ and returns a write confirmation to the user;
4. The API endpoint forwards to the TTP the encrypted user token and the list in T_{J_i}.
5. The TTP verifies the authenticity of the user token, performs a remote attestation of the hosts in $\mathsf{T}_{\mathsf{J}_i}$ to verify their software platform state and confirm the claimed location in jurisdiction J_i, decrypts K and seals it to the trusted configuration of the hosts in $\mathsf{T}_{\mathsf{J}_i}$;

[7]The value of PCR 15 after TPM_Startup is 0, as specified in [36].
[8]The choice of SHA-1 is imposed by the TPM v1.2 specifications, see [36]

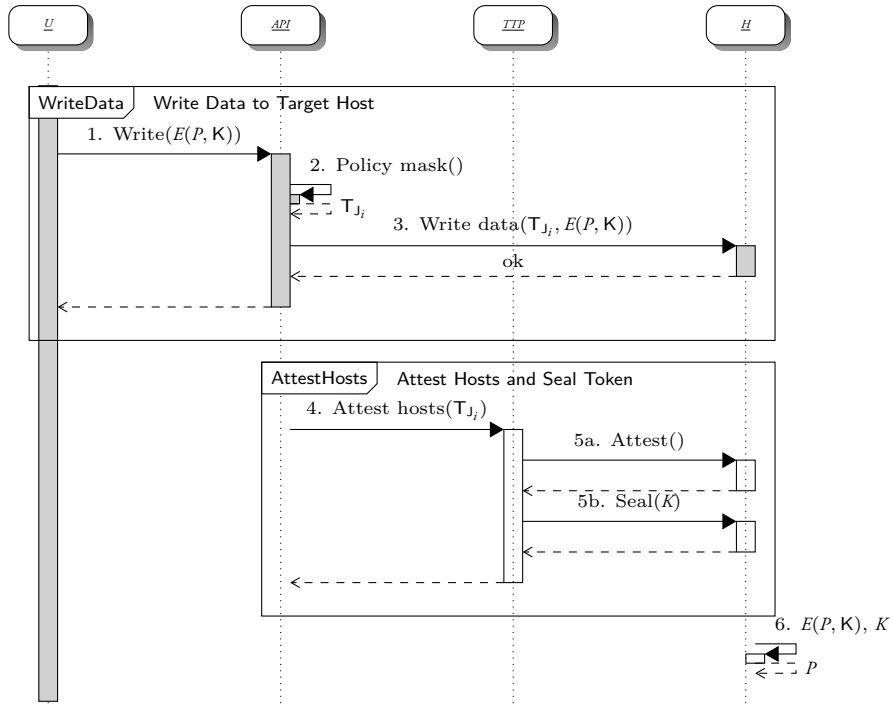


Figure D.3: Protocol message flow: data placement with geolocation cell restrictions.

6. The hosts in T_{J_i} decrypt $E(P, \mathsf{K})$ and use plaintext data (P) for storage or processing.
7. When requested by user or by the replicator component for synchronization, the storage hosts encrypt data with the same key K and send $E(P, \mathsf{K})$ to the requester (for clarity, we have omitted this step from figures D.4, D.3).

Listing D.2: Location policy for a data object

```
<?xml version="1.0" encoding="UTF-8"?>
<LocationPolicy>
    <PolicyID>cred:id</PolicyID>
  <Timestamp>1289123342</Timestamp>
  <GeocellList>
      <Geocell>
      <Location>suburb:name</Location>
      <Location>city:name</Location>
      <Location>county:name</Location>
      <Location>state:name</Location>
      <Location>country:name</Location>
      <Geocell>
  </GeocellList>
</LocationPolicy>
```

Note that user u_1 is free to specify either a single or more geolocation cells. The situation when reverse geocoding maps the location point to a wrong geolocation cell is unlikely, and reoccurring errors can be corrected by redefining the boundaries of the respective geolocation cells.

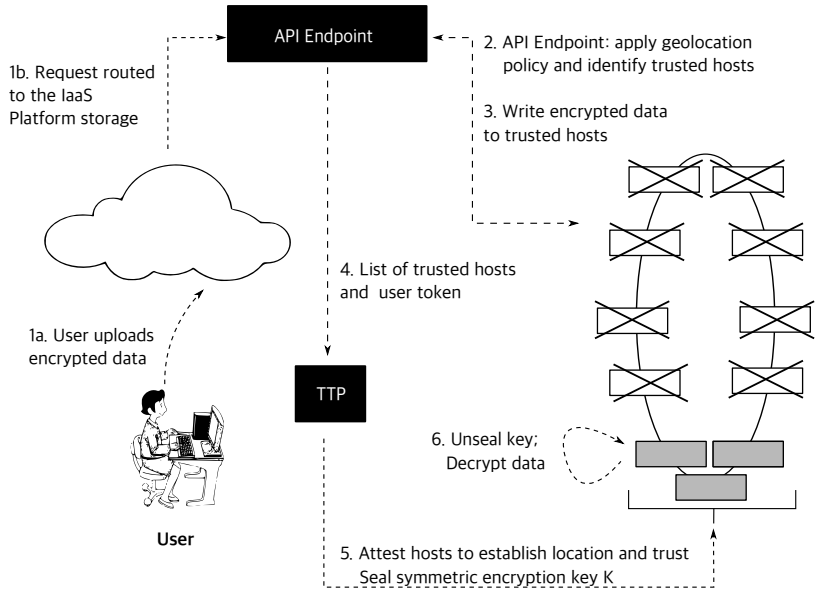


Figure D.4: Process model, data placement with geolocation cell restrictions

6 Prototype Implementation

In order to test the validity of the above protocol, we have implemented it using the Swift distributed object storage [228]. Release 1.9.0 of the Swift distributed object storage introduced support for *global clusters*. This includes capabilities for a separate replication network and read/write affinity configuration, which combined enable Swift to run as a single cluster over a wider geographic area, explicitly addressing our assumption about geographically distributed data stores. In addition, Swift provides capabilities to define storage policies based on certain attributes of the storage hosts. For this prototype, we have implemented a policy functionality and configured our Swift deployment to only store the encrypted data on the hosts in a certain jurisdiction J_i. Hosts with the correct location information were able to unseal the encryption key when needed. A modification of the object storage used the unsealed encryption key to decrypt the data before it was written to the node, thus only storing plaintext data on authorized hosts. However, once a file is *requested* from the respective object storage (by the replicator component for container synchronization, or by the API endpoint to be server to the client), it is encrypted with the same encryption key.

We revisit the requirements outlined in §3.3. In protocol context, we identify two classes of objects: *(i)* objects uploaded or created by the data owner; *(ii)* object replicas created by the storage for operational purposes. The objects in class *(i)* are encrypted by the data owner and the decryption key is made available – according to the above protocol – only to *TP* located in the jurisdiction prescribed by the data owner. Objects in class *(ii)* are *maintained encrypted* and placed on storage hosts at the discretion of the object storage.

For the second requirement, the location policy specified by the object owner enumerates the geolocation cells where objects may be places in plaintext. The TTP will only seal the

encryption key on *TP* that are located in the allowed jurisdiction; the adversary, as defined in the adversary model above, can not force the TTP to seal the encryption key on other hosts.

For the third requirement, note that only storage nodes running on *TP* have access to the plaintext data for storage and processing. All of the data (f_1') derived through processing will be protected with the same symmetric encryption key prior to being copied by the replicators to other hosts in the ring. In this way, f_1' will inherit the same locations policy: only *TP* placed in the jurisdiction specified by the owner of object f_1 will have access to the encryption key.

In our implementation, we have used a `Lenovo Thinkpad T430s` host which in a standard configuration has a navigation device `Ericsson H5321 gw Mobile Broadband GPS` attached to the motherboard on a PCIe slot. Navigation devices are widely deployed on mobile platforms (including laptops), and can be easily added to other types of platforms which do not have them pre-installed. Our test host runs Linux (`CentOS 6.4`) which assigns device name `/dev/ttyACM2` to the navigation device according to the default udev rule[9]. In order to have a persistent device name and to ensure that D reads location data from an authentic source, we define a custom udev rule (shown in listing D.3) – protected by including it in the TCB – and place it in `/etc/udev/rules.d/`, such that whenever the navigation device is switched on, a persistent symbolic link `/dev/navigation_device` is created and D is started.

Listing D.3: 10-navigation-device.rules

```
1  ATTRS{modalias}=="usb:v0B....ip01",
2  ATTRS{interface}=="Ericsson H5321 GPS",
3  SYMLINK+="navigation_device",
4  RUN+=/usr/bin/location_daemon
```

Given the limitations of GPS signal receivers, placing such receivers on individual platforms might not be a viable solution in many cases, such as data centres located underground or placed in sealed metal containers, as described in [225]. As an alternative, we apply the solution described in [230] and presented in Fig. D.2, where a geolocation master server with a dedicated GPS antenna provides signed geolocation cell information to all other hosts in the cluster (e.g. a rack). In this case, the protocol would need to also include the attestation of this dedicated geolocation master. Due to space limitations, we omit the description of these alternative, yet justified scenarios.

For reverse geocoding, we have used the OpenStreet Maps dataset and the API provided by the OpenStreet Maps project[10]. While the size of the complete planet data set is significant (in the range of 400 G), a filtered version containing only country information, produced using a tool such as Osmfilter[11], is significantly smaller – 187 kb – and can be placed on each IP host and included in the TCB.

7 Security Analysis

GPS signal security is an important aspect for the presented solution and is an active research topic [238–240]. Forays into GPS signal security are out of the scope of this paper and for the purposes of the solution we assume that the setup is capable to detect GPS spoofing attempts,

[9] https://www.kernel.org/pub/linux/utils/kernel/hotplug/udev/udev.html
[10] http://planet.openstreetmap.org/, http://nominatim.openstreetmap.org/
[11] http://wiki.openstreetmap.org/wiki/Osmfilter

as described in [238, 240]. However, we consider GPS security at application level, to ensure that daemon D is not tampered and it reads the location data from an interface which is connected to a valid, physical navigation device. The integrity of D is protected by including it in the TCB; moreover, in order to securely and persistently bind the navigation device to a navigation interface, we create a custom `udev rule`, which is also protected by including it in the TCB. Udev rules are loaded by the Linux kernel during the first boot stages and are later used by the udev daemon (i.e. `udevd` which is also a part of TCB) to assign a device name (e.g. `/dev/navigation_device`) when a `uevent` is triggered by that device (e.g. device attached, removed, enabled, disabled, etc.). Including D, `udevd` service and `udev` rules into the TCB prevents their modification under the adversarial model described in (§ 3.2).

When it comes to storage and processing of data in the cloud storage, in § 5 we propose a protocol to both store and process data in plaintext only on trusted hosts in a certain jurisdiction chosen by the user. In both cases, we rely on the remote attestation of hosts (the procedure is described in [131]), expanded with information about the geographical position of the host. The result is, that a host whose platform state has changed through either software modification or change of physical location will not be able to obtain the plaintext version of user data. An adversary in our model might have two goals – obtain the plaintext version of user data or process data in plaintext in a jurisdiction that is not acceptable by the client, in order to reduce operational costs. In both cases, given the physical security of the hosts, confidentiality protection keys will not be made available in case of either a change of the boot aggregate (PCR $\{0-7\}$) or the geographical position of the host PCR$\{15\}$. Here it is worth mentioning that host physical security and change in the geographical position of the platform are not mutually exclusive – considering the example of the modular data centres above, a sealed container with intact platforms may be transferred to a different jurisdiction without affecting the integrity of the platforms.

8 Performance

In our experiments, the average time for a commodity GPS receiver from a "cold" start to acquire at least 4 satellites was 97.5 seconds (the GPS device was reset between measurements using the command `AT*E2RESET`). The GPS device maintained a "hot" start between reboots and thus could acquire at least 4 satellites immediately after initialization[12]. Important to note, according to the protocol, acquiring satellites is only necessary at platform boot and does not affect subsequent data access time, we thus exclude this factor from the following performance assessment.

Given the steps of the protocol and considering that the TPM is a relatively slow device, it is to be expected that the largest contributors to the performance impact are the TPM unseal operation and most importantly the data encryption and decryption operations. The TPM unseal operation to obtain the symmetric encryption key, is also a one-time operation, performed once when the respective storage hosts process the data. Once the decryption key is unsealed, it is maintained in memory and used for decryption and encryption of the respective data. In our experiments, the average duration of the unseal operation was 1.23 s. To evaluate the performance of the protocol, we have measured the execution time of a `PUT` operation using

[12] *Cold Start*: The receiver must download almanac and ephemeris information to achieve a position fix. *Hot Start*: A hot start occurs when a receiver has up-to-date almanac and ephemeris information. `http://www.ni.com/white-paper/7189/en/`

file sizes between 1 and 100 M.

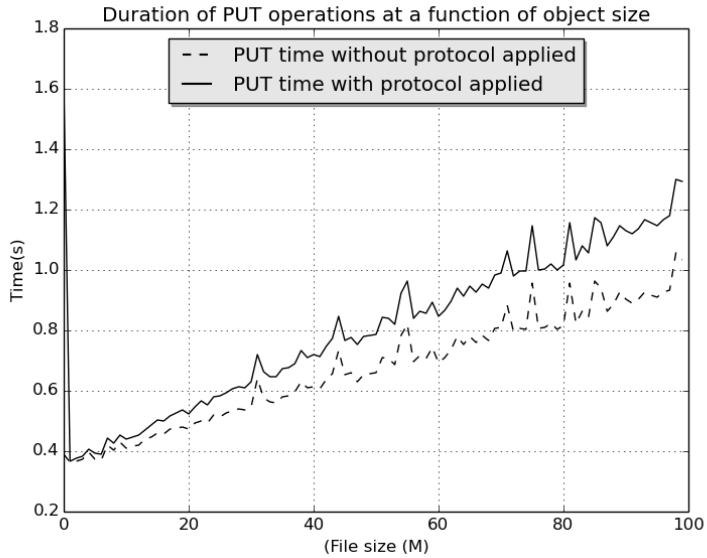


Figure D.5: Performance measurements for PUT operations to Swift distributed object store, with and without the data placement protocol

The results – also presented in Fig. D.5 – showed that encrypting objects prior to writing to the object storage implies a relatively small overhead (∼25%) which increases linearly with the size of the file. The initial spike in the graph is due to the relatively long time to unseal the *SealKey*. As it is clearly visible, this does not affect the following PUT operations. The above results may vary, depending primarily on the configured number of replicas and consistency policy. The optimal configuration is use case and deployment architecture specific, and is thus out of the scope of this paper.

9 Conclusion

In this paper we propose a solution to control the geographic location of plaintext data placed and processed in Infrastructure-as-a-Service deployments. Our analysis of the related work reveals the need to advance beyond verifying that certain data is placed in a certain location. We address this issue and propose several solutions on device, operating system, and object storage platform levels to ensure that data is *only* stored and processed in plaintext in the jurisdiction designated by the data owner. We use trusted computing protocols in order to perform remote attestation of storage and processing hosts, as well as to seal cryptographic material to trusted platforms of the hosts *and their geolocation*, which we obtain from either a commercial off the shelf device on the host motherboard or a dedicated geolocation master node. On the IaaS level, we leverage the trusted state of the platform to decrypt, store and process user data in plaintext only on hosts located in a certain jurisdiction specified by the

user prior to upload. Our performance tests have demonstrated the feasibility of the proposed approach; further improvements in CPU architectures are expected to reduce the overhead induced by the data encryption stages of the algorithm. Future work includes refinement of the proposed geolocation cell model, integration of platform attestation with kernel-level mandatory access control policies as well as minimization and eventual elimination of the TTP, which would allow us to consider stronger adversary models.

Paper E

Towards Secure Multi-tenant Virtualized Networks

Nicolae Paladi and Christian Gehrmann

Abstract

Network virtualization enables multi-tenancy over physical network infrastructure, with a side-effect of increased network complexity. Software-defined networking (SDN) is a novel network architectural model – one where the control plane is separated from the data plane by a standardized API – which aims to reduce the network management overhead. However, as the SDN model itself is evolving, its application to multi-tenant virtualized networks raises multiple security challenges. In this paper, we present a security analysis of SDN-based multi-tenant virtualized networks: we outline the security assumptions applicable to such networks, define the relevant adversarial model, identify the main attack vectors for such network infrastructure deployments and finally synthesize a set of high-level security requirements for SDN-based multi-tenant virtualized networks. This paper sets the foundation for future design of secure SDN-based multi-tenant virtualized networks.

1 Introduction

Rapid development of cloud services made a successful case for virtualization, which allows infrastructure providers to multiplex physical resources and provide complete platform and network resources to multiple tenants.

Multi-tenant cloud infrastructure relies on virtualization of both hosts and network infrastructure. In both cases, system virtualization presents a trade-off between portability and

tenant isolation on one hand, and virtualization overhead on the other hand. For host virtualization, *virtual machines* (VMs) are a widely used approach to enable multi-tenancy in the Infrastructure-as-a-Service cloud model. Similarly, a variety of approaches exist for network virtualization, operating on different levels. Sherwood [241] described five dimensions that must be *sliced* to enable network virtualization: bandwidth, topology, traffic, device CPU and the forwarding tables (also called *forwarding information base*, FIB).

Given the inherently distributed nature of network infrastructure, no single component modification can satisfactorily slice the network across all five dimensions. Multi-tenancy is among the capabilities enabled by virtualization. In the context of infrastructure clouds, we consider the following characteristics of network multi-tenancy:

- A tenant corresponds to a customer using a particular virtual network;
- Tenants may belong to different administrative domains;
- Tenants expect network isolation of their domain;
- Physical resource sharing is fully abstracted, with tenants unaware of other neighbours;
- Tenants may create multiple distinct virtual network instances and topologies.

By enabling network multi-tenancy with strong isolation, network virtualization allows infrastructure providers to multiplex the network infrastructure among network service providers, paving the way for new services and better hardware resource utilization. However, network infrastructure multi-tenancy comes at the cost of increased complexity, leading to higher management costs and new security risks. Software-defined networking (SDN) is a network architectural approach evolved from the "Clean slate" initiative [92], which proposed to decouple the network forwarding functionality from the control and management logic. The initiative aimed to improve network management flexibility based on clear network abstractions: the *management applications*, which expresses the operator goals on a high level; the *network hypervisor*, which implements control program instructions based on a global network view and computes forwarding state for search router/switch; the *network operating system* (NOS), builds the global network view and implements configurations on switches; and finally the *routing and switching equipment*, which forwards packets as instructed. This paved the way for the wide-scale use of commodity hardware for network infrastructure, flexible software implementation of network functionality and new network virtualization abstractions. The SDN architectural approach continues to be a work in progress. Despite a large body of contributions to the SDN architecture ([159, 165, 174, 242–244]), network operating systems ([98, 99, 158, 245]) and the communication between the data plane and the control plane (also referred to as "southbound API", [246]), SDN continues to evolve, requiring further attention to aspects such as security, scalability and policy enforcement.

1.1 Contribution

In this paper, we review the security challenges for multi-tenant virtualized network infrastructure based on the SDN architecture. We briefly describe the application of the SDN approach to the multi-tenant virtualized network infrastructure. We introduce an adversarial model suitable for multi-tenant virtualized network infrastructure using the SDN architecture and identify a set of relevant attack vectors. Finally, we present a list of security requirements towards SDN-based multi-tenant virtualized network infrastructures.

1.2 Organization

The paper is organized as follows: in Section 2 we present the related work; next, we introduce the adversarial model for SDN-based multi-tenant virtualized network infrastructures in Section 3, followed by a description of the relevant attack vectors in Section 4. We continue by listing the requirements for SDN-based multi-tenant virtualized network infrastructures in Section 5 and conclude in Section 6.

2 Related Work

In this section, we provide an overview of related work on secure SDN architectures.

In [174], the authors presented Ethane (Figure E.1), an enterprise network architecture which allows network managers to control the network using a unified interface. Reflecting the identified enterprise network characteristics, the Ethane network consists of commodity switches and one or multiple controllers. The former are responsible for maintaining the FIB and contain a local switch manager to communicate with the controller. The latter handles host registration and authentication, tracks network bindings, verifies permissions and grants access, as well as enforces resource limits on the managed flows. In addition, the authors described a high-level policy definition language for network management policies.

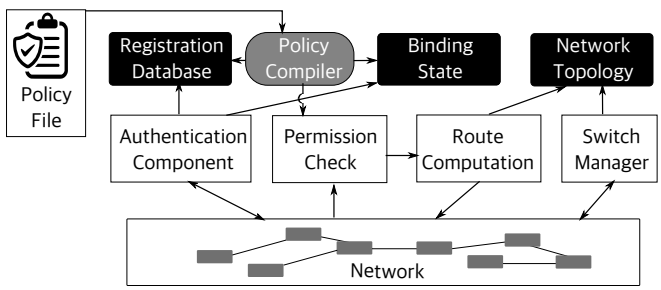


Figure E.1: The main components in Ethane [174]

NOX was introduced in [98], as a network operating system which presents network management programs with a *centralized* programming model and a global view of the system state. This allows network management programs to rely on simpler graph processing algorithms to compute the shortest paths and to operate with higher-level abstractions, such as users and host names, rather than MAC and IP addresses. NOX consists of several distinct *controller* processes operating on a global network view built by NOX based on the process communication with the data plane. The global view is based on switch topology, user location, connected network components (e.g. hosts and middleboxes), as well as bindings between the names and addresses. Controller processes use the global view to make management decisions, which are later implemented on the routing and switching equipment over the OpenFlow API [246].

In [158], the authors presented FortNOX, a software extension for role-based authorization and security constraints enforcement for the NOX OpenFlow Controller. FortNOX detects rule conflicts, i.e. situations when candidate OpenFlow rules modify network flows specified by existing rules, and takes appropriate actions, depending on the authorization of the rule

requestor. Role-based authentication is used to determine the security authorization of each rule producer, enforcing the principle of least privilege to ensure integrity of the mediation process. FortNOX consists of four components: a role-based source authentication module to validate signatures for each flow rule insertion request; a conflict analyzer to evaluate each new flow rule against existing active flow rules; a state table manager to track the current flow rules; and finally a flow rule time-out callback interface to update the aggregate flow table upon rule expiration. For conflict resolution, FortNOX converts all flow rules into 'Alias Reduced Rules', allowing to perform a rule set conflict evaluation. FortNOX includes a security directive translator ifor the *block, deny, allow, redirect, quarantine, undo, constrain,* and *info* directives; such directives are used for high-level threat mitigation which are in turn translated to flow rules to handle suspicious traffic. FortNOX has been implemented and evaluated as an extension to NOX and found to introduce an average overhead of less than 7 ms for evaluating a candidate flow rule against 1000 existing flow rules.

In [159] the authors presented 'Fresco', an OpenFlow security application development framework, which facilitates rapid prototyping of composable detection and mitigation modules. Such modules represent elementary building blocks of Fresco and contain the following interfaces: *input, output, event, parameter,* and *action*. The main functions provided by Fresco are script to module translation, database management, event management and instance execution. Implementation of policies defined by composable applications is ensured by the Security Enforcement Kernel built into the fabric of the network operating system. Furthermore, the paper contains a description of a collection of Fresco modules, several composed security applications and a performance evaluation of Fresco. Fresco can be integrated as a security extension module into other network operating systems – such as NOX.

Finally, in "Rosemary: A Robust, Secure, and High-Performance Network Operating System" [99], the authors present a network operating system focusing on network resilience in the presence of faulty or malicious applications by creating sandboxed environments for network applications. Sandboxing (also called 'micro-NOS') is achieved by launching each application in a separate process context with access to all of the libraries that the application requires. Each Micro-NOS also contains a resource monitor to supervise the applications and operates within the permission structure of 'Rosemary' network operating system. In turn, the 'Rosemary' network operating system is an application running on a commodity Linux distribution. The isolation offered by the 'micro-NOS' allows to improve robustness, such that faulty or malicious applications are prevented from crashing the entire network operating system. Furthermore, the paper also aims to address security aspects in order to prevent malicious network applications from accessing internal data structures of other network applications. This is achieved by implementing an 'AppZone' sandbox, where privileged system calls made by a network application are interposed and verified by the sandbox framework. To avoid the declared '20-30%' performance overhead, the authors recommend two optimisations called 'request pipelining' and 'trusted execution'. The latter in essence removes the sandbox isolation, allowing the application to run as a kernel process; however, this makes the security advantages of 'Rosemary' less evident. Finally, the authors present the performance evaluation results, which show that the 'Rosemary' network operating system performs roughly on par with the NOX [98] approach on a 1G link and can perform on par with NOX on a 10G link with the 'trusted execution' optimisation in place. This contribution highlights the need for improved security in the architectures of the proposed network operating system. However, one major drawback of the proposed approach is that it ignores distribution aspects, despite significant progresses in distributed network operating system design [245] and the demonstrated need for physical distribution of the control plane [247].

Kreutz et al presented a list of seven attack vectors identified in SDNs [155]: **a.** Forged or faked traffic flows; **b.** Attacks on vulnerabilities in switches; **c.** Attacks on control plane communications; **d.** Attacks on and vulnerabilities in network controllers; **e.** Lack of mechanisms to ensure trust between the controller and management applications; **f.** Attacks on and vulnerabilities in administrative stations; **g.** Lack of trusted resources for forensics and remediation. However, only part of the above attack vectors are exclusively relevant to SDN networks. In this paper, we focus on attack vectors specific for SDN deployments.

The SDN network architectural model abstracts the complexity of network virtualization; however, in adopting it we must revisit the set of network virtualization mechanisms, identify the most relevant attack vectors, define a relevant adversarial model and outline a set of security requirements towards SDN-based multi-tenant virtualized networks.

3 System and Threat Model

In this section, we introduce the SDN system model followed by an example scenario. We next define an adversarial model for SDN-based multi-tenant virtualized networks.

3.1 SDN system model

A conceptual model of the SDN architecture is depicted in Figure E.2, and described below based on the SDN architectural model presented in [165].

- The *data plane* contains both hardware and software routing equipment. This component implements the routing policies that satisfy the goals of the network administrator. It lacks decision logic and is optimized for forwarding speed. Packets that do not match any policy are either discarded or communicated to the control plane through the southbound API.
- *Southbound API* is a vendor-agnostic set of instructions implemented by the routing equipment on the data plane. It allows bi-directional communication between the data and the control planes.
- *Control plane* is a logically distributed abstraction layer that transforms high-level network operator goals into discrete routing policies based on a global network view. It contains a distributed *network operating system*, which builds and maintains the global network view as well as communicates with the equipment on the data plane. The control plane also includes the *network hypervisor*, which multiplexes the available network resources among multiple users with distinct virtual network topologies.
- *Management applications* are used by network administrators to express their network configuration goals using a set of high-level comments. They could also include software-based network management components such as firewalls, intrusion detection systems, traffic shapers, etc.

The above SDN system model will be used as a basis for the adversarial model and threat analysis in the subsequent sections.

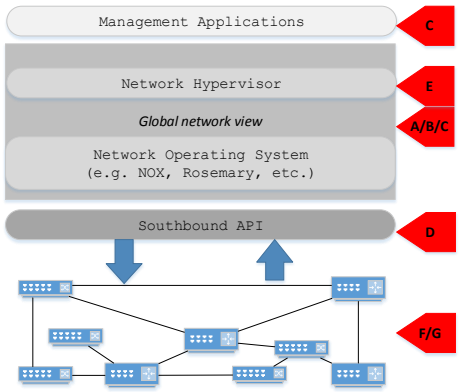


Figure E.2: SDN system model. Letters mark attack vectors, presented in Section 4.

3.2 SDN Multi-tenant Network Scenario

As an example scenario, consider the cloud network infrastructure model, where tenants purchase virtual network resources (e.g. bandwidth, switching capabilities) from network infrastructure providers. Tenants are represented by the *Virtual Topologies* layer in Figure E.2. The parameters of the virtual network resources agreed upon by the network infrastructure provider and tenants are defined by a Quality of Service (QoS) agreement.

In a shared, multi-tenant virtual network environment, the tenants only have a view of the network infrastructure limited to their administrative domain and are not aware about the presence of other tenants. Furthermore, tenants do not have direct access to the configuration of the underlying data plane infrastructure and instead administer their own network domain through routing policies, which are then interposed by the network hypervisor and communicated through the southbound API to the data plane. Enabled by the capabilities of the SDN architectural model, tenants launch their own network management applications (as defined in Section 3.1). Such network management applications can be made available both by the network infrastructure provider and by third party application providers.

3.3 Adversarial model assumptions

We present our assumptions regarding SDN-based multi-tenant virtualized networks in the presence of an adversary.

Assumption of hardware integrity Recent media revelations have raised the issue of hardware tampering en route to deployment sites [2, 213]. We assume that the cloud provider has taken necessary technical and non-technical measures to prevent such hardware tampering.

Assumption of physical security We assume physical security of the data centres where the network infrastructure is deployed. This assumption holds both when the network infrastructure provider owns and manages the respective data center (as in the case of Amazon Web Services, Google Compute Engine, Microsoft Azure, etc.) and when the network infrastructure provider utilizes the capacity of a data center operated by a third party (e.g. CloudSigma), since physical security can be observed, enforced and verified through known best practices by third-party organizations. This assumption is important for building higher-level hardware and software security guarantees for the components of the network infrastructure.

Assumption of cryptographic security We assume symmetric and public-key encryption schemes are semantically secure and that the adversary cannot obtain the plain text of encrypted messages. We also assume the signature scheme is unforgeable, and that the message authentication code algorithm correctly verifies message integrity and authenticity. Finally, we assume that the adversary, with a high probability, cannot predict the output of a pseudorandom function.

3.4 Adversary capabilities

We next describe specific capabilities for adversaries (denoted by $\mathcal{ADV}$). We adopt the Yao-Dolev threat model [211], such that the adversary can overhear, intercept, and synthesize any message and is only limited by the constraints of the employed cryptographic methods. Furthermore, we assume that the adversary can analyse the traffic patterns in the network through passive attacks and may disrupt or degrade network connectivity to achieve its goals, such as e.g. force the sender and the receiver to choose a less secure form of communication. While we prioritise adversaries aiming to compromise the confidentiality and integrity of data in network infrastructure deployments, we also aim to limit the capabilities of attackers to mount Denial-of-Service attacks to disrupt connectivity. We denote this as $\mathcal{ADV}$ *A*.

We define two additional complementary adversarial types. Acting as a tenant (e.g. through impersonation), the adversary obtains new capabilities in addition to the ones described above; we define this as $\mathcal{ADV}$ B. In this case the adversary is able to perform the following actions using valid network tenant credentials[1]:

1. Send valid tenant packages with an arbitrary content and frequency to the components it can reach;
2. Attempt to impersonate other tenants;
3. Install arbitrary management applications and issue arbitrary policies within its network domain;
4. Use the cryptographic material at its disposal to attempt to decrypt intercepted network traffic that is sent and received by other tenants.

Furthermore, the adversary may manage to take over one of the SDN controller components or some control of the network operating system. We denote this as $\mathcal{ADV}$ C; it will be able to perform the following actions:

[1]Related adversary capabilities are defined in: `https://tools.ietf.org/html/draft-ietf-nvo3-security-requirements-04`

1. Affect the network communication of the SDN-based infrastructure by sending valid packets with an arbitrary content and frequency to all reachable network components;
2. Attempt to impersonate network infrastructure components;
3. Issue malicious policies aiming to either monitor, distort or disrupt network traffic;
4. Use the cryptographic material at its disposal to attempt to decrypt intercepted network traffic that is sent and received by other network infrastructure components.

4 Attack Vectors

We review the attack vectors relevant for SDN-based multi-tenant virtualized networks considering the adversarial models presented in Section 3.

4.1 Vulnerabilities in the control plane

Along with ease of network administration, a central control plane introduces a primary attack target for an adversary motivated to take control of the network. Taking over the control plane component in the SDN architecture allows the adversary to obtain full control of the network communication, different from traditional networks where communication control is distributed throughout various network components. Possible solutions include splitting the controller into several domains or distributing the control plane over several hosts, such that issued policies are verified on a different component before deployment.

4.2 Attacks on control plane communications

To manipulate network policies, the adversary may attempt to spoof the control plane communication (both among the components of a distributed controller and between management applications, controller and data plane). Possible solutions include enforcing authenticated and encrypted communication between all of the control plane components, as well as secure enrolment mechanism for management applications and data plane devices.

4.3 Lack of a trust chain between the management applications and the data plane

While the effort on defining the SDN architecture is still in progress, it is clear that management applications belong to a different security domain than the network operating system, and can be launched by malicious administrators or issue conflicting policies. Both *detecting* and *preventing* malicious policy deviations is challenging: a tenant can only observe the traffic after a change has been applied, but cannot obtain and examine snapshots of the data plane FIB; similarly, there is no mechanism to establish a trust chain between tenant commands and entries in the FIB. Possible solutions can be adapted from the ones employed – with varying success – on platform operating systems: verification of code origin and information flow control; however, such mechanisms do not satisfy malicious policy detection requirements.

4.4 Attacks on policies and rules in programmable networks

Even if the integrity of policies remains intact, the adversary may issue malicious policies that modify or disable the effect of legitimate policies already in place (specifically in the scenario with such network management applications implement functionality of network middleboxes). This type of attack is difficult to detect and prevent, since the malicious policies might be indistinguishable from legitimate ones up to the point when the combined policy is deployed (furthermore, it requires a robust definition of a "malicious policy"). Possible solutions are to establish policy hierarchies and perform policy integration verification against some predetermined invariants prior to deployment, to ensure that the resulting modifications remain within the basic policy framework. As policy updates may occur interactively in response to changing network patterns, both static analysis of policies and a pre-deployment simulation may be required.

4.5 Resource limit violations

A malicious tenant may deploy network management applications that exploit vulnerabilities in network service isolation in order to gain network resources beyond the allocated quota defined in the QoS agreement. Possible solutions include adding network operating system capabilities for fine-grained monitoring of management applications to prevent resource overallocation. This in turn requires a well-defined network resource model based on clear definitions of network resources and their respective capacities.

4.6 Attacks on virtual switches and network gateways

As pointed out in Section 3.4, an adversary that controls a virtual network infrastructure component (such as a virtual switch) can attempt to impersonate other virtual network infrastructure components, spoof traffic and negatively affect tenant isolation. Possible solutions include integrity verification of virtual network infrastructure components and protecting the cryptographic secrets necessary for network access using a hardware root of trust.

4.7 Weak bandwidth isolation as attack vehicle

One of the consequences of NIC virtualization is a weakening of QoS guarantees, since most NIC virtualization implementations do not support guaranteed bandwidth [248]. While this does not directly affect data integrity and confidentiality, manipulating bandwidth allocation between tenants sharing a resource can be used in order to force a policy change (e.g. trigger a more permissive policy that is activated when the available bandwidth falls below a certain threshold). Possible solutions include widespread proliferation of bandwidth isolation techniques such as described in [249], as well as including the effects of bandwidth changes into network policy security testing.

5 Security Requirements

In this section, we outline a set of initial security requirements for SDN-based multi-tenant virtualized networks, based on the adversarial model described in Section 3 and the attack vectors in Section 4:

1. **4.1:** The SDN control plane must implement an access control model that limits the effects that vulnerabilities in controllers can have on tenant domains. This can prevent an adversary from simultaneously gaining control over the functionality of the SDN controller component at all privilege levels and in all roles.
2. **4.1:** A dedicated entity must verify the policies to be implemented by the SDN control plane before deployment.
3. **4.2:** All communication between control plane components must the authenticated, and a secure enrolment mechanism for management applications and data plane devices must be in place.
4. **4.3:** A mechanism must be in place to offer traceability and non-repudiation for all configuration commands and policies issued by network management applications.
5. **4.4:** A mechanism must be in place to enforce strong network policy isolation, such that the effects of policies in a certain tenant domain have no effect on other domains. Furthermore, the infrastructure provider must be able to enforce strict boundaries on the effects of policies within tenant domains.
6. **4.4:** New network management policies must run through an integration verification engine prior to deployment, to minimize or exclude the effect of malicious policies on the network configuration.
7. **4.5:** A mechanism must be in place to ensure that network management applications do not allocate resources beyond the assigned quota. To do this, the NOS may apply advanced policing mechanisms – e.g. based on existing extensions, such as in [158] – that keep fine-grained tracking of management applications resource utilization and prevent them from making over-allocations.
8. **4.6:** Integrity of virtual network components must be verified prior to deployment and the cryptographic material required for their network access must be protected with a hardware root of trust.
9. **4.7:** Policy-based routing decisions must not be affected by vulnerabilities in bandwith isolation between tenants. To clarify, consider a network setup with two types of paths: low-bandwidth, low-cost, low-security permanent paths (*type-A* paths) and high-bandwidth, high-cost, high-security switched paths (*type-B* paths).

   Consider further that a legitimate tenant has configured a policy to distribute different types of traffic (low-value and high-value traffic) among the *type-A* and *type-B* paths respectively. An adversary capable of modifying the bandwidth allocated to the paths of the legitimate tenant should not succeed in redirecting high-value traffic through *type-A* paths.
10. **4.7:** Software and hardware network components must offer equally strong bandwidth isolation properties. In the current networks, the data plane components include both software switches and routers deployed on commodity platforms and specialized hardware equipment implemented using application-specific integrated circuits. As pointed out in [248], software-based data plane components lack many of the features currently implemented in specialized hardware switches and routers. Strong bandwidth isolation is one of the features which must be improved in the software implementations.

We plan to address in our forthcoming work the above security requirements towards SDN-based multi-tenant virtualized networks.

6 Conclusion

Integration of large-scale multi-tenant virtualized network infrastructure with the SDN architectural model presents a set of unsolved security challenges. In this paper, we perform a high-level analysis of SDN-based multi-tenant virtualized networks. We present three security assumptions about such networks, that are necessary for defining a secure virtualized network infrastructure. Based on these assumptions, we define the relevant adversarial model and identify the main attack vectors for such network infrastructure deployments. Finally, based on the defined adversarial model and resulting attack vectors, we outlined a set of high-level security requirements for SDN-based multi-tenant virtualized networks. The adversarial model, attack vector analysis and high-level security requirements defined in this paper serve as an initial input towards future design work of secure and trusted SDN-based multi-tenant virtualized networks.

Paper F

TruSDN: Bootstrapping Trust in Cloud Network Infrastructure

Nicolae Paladi and Christian Gehrmann

Abstract

Software-Defined Networking (SDN) is a novel architectural model for cloud network infrastructure, improving resource utilization, scalability and administration. SDN deployments increasingly rely on virtual switches executing on commodity operating systems with large code bases, which are prime targets for adversaries attacking the network infrastructure. We describe and implement TruSDN, a framework for bootstrapping trust in SDN infrastructure using Intel Software Guard Extensions (SGX), allowing to securely deploy SDN components and protect communication between network endpoints. We introduce *ephemeral flow-specific pre-shared keys* and propose a novel defense against *cuckoo attacks* on SGX enclaves. TruSDN is secure under a powerful adversary model, with a minor performance overhead.

1 Introduction

Renewed and widespread interest in virtualization – along with proliferation of cloud computing – has spurred a series of innovations, allowing cloud service providers to deliver on-demand compute, storage and network resources for highly dynamic workloads. Consequently, more hardware and virtual components are added to already large networks, complicating network management. To help address this, SDN emerged as a novel network architecture model. Separation of the *data* and *control* planes is its core principle, allowing network operators to

implement high-level configuration goals by interacting with a single *network controller*, rather than configuring discrete network components. The controller applies the configuration to the *network edge*, i.e. to its global view of the data plane [98]. Data and control plane separation in SDN challenges network infrastructure security best practices evolved in the decades since packet-switched digital network communication gained popularity [155], [250].

In the cloud infrastructure model, SDN allows tenants to configure complex topologies with rich network functionality, managed by a network controller. The availability of a global view of the data plane enables advanced controller capabilities – from pre-calculating optimized traffic routing to managing applications that replace hardware middleboxes. However, these capabilities also make the controller a valuable attack target: once compromised, it yields the adversary complete control over the network [251]. The global view itself is security sensitive: an adversary capable of impersonating network components may distort a controller's global view and influence network-wide routing policies [252].

Virtual switches are another category of security sensitive components in SDN deployments. They execute on commodity operating systems (OS) and are often assigned the same trust level and privileges as hardware switches – specialized network components with compact embedded software [253] or application-specific integrated circuits. Commodity OS are likely to contain security flaws which can be exploited to compromise virtual switches. For example, their configuration can be modified to disobey the protocol, breach network isolation and reroute traffic to a malicious destination or compromise other network edge elements through lateral attacks. Such risks are accentuated by the extensive control a cloud provider has over the infrastructure of its tenants.

Security and isolation of tenant infrastructure can be strengthened by confining select SDN components to trusted execution environments (TEE) and attesting their integrity before provisioning security-sensitive data. TEEs with strong security guarantees can be built using SGX, a set of recently introduced extensions to the x86 instruction set architecture and related hardware [50,51]. Earlier work used SGX to protect computation in cloud environments, by executing modified OS instances in SGX enclaves [61] or a data processing framework in a set of SGX enclaves [63]. However, while both of the above efforts *highlighted* the need to secure network communication, they did not *address* it.

1.1 Contribution

This paper makes the following contributions:

- We present TruSDN, a framework to bootstrap trust in SDN infrastructure.
- We introduce flow-specific pre-shared keys for communication protection.
- We propose a defense against cuckoo attacks [62], based on properties of the enhanced privacy ID (EPID) scheme [55] used for remote enclave attestation.
- We describe the implementation and a performance evaluation of TruSDN.

1.2 Organization

We introduce the system model in Section 2, describe the adversary model in Section 3 and the design of TruSDN in Section 4. In Section 5 we provide a security analysis, describe the

prototype implementation and performance evaluation in Section 6 and review the related work in Section 7. We discuss future work in Section 8 and conclude in Section 9.

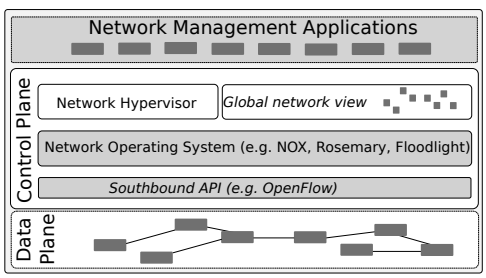
2 System Model

In this section we describe the SDN architectural model and the SDN deployment layers. Furthermore, we describe the use of TEEs based on Intel SGX.

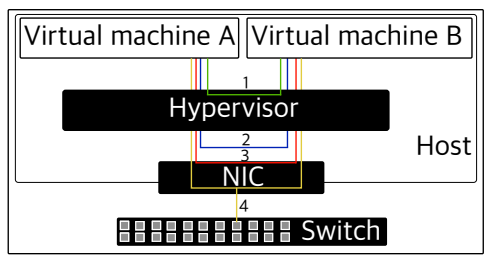
2.1 Software Defined Networking

In this paper we target SDN in infrastructure cloud deployments. The system model follows the architecture presented in [165] and depicted in Figure F.1a.

The *data plane* includes hardware and software switch implementations. *Software switching* is used in cloud deployments due to its scalability and configuration flexibility. Figure F.1b illustrates the software switching approaches for communication between two collocated endpoints. In a typical switch implementation, its kernel-space component is optimized for forwarding performance, lacks decision logic and only forwards packets matching rules in its *forwarding information base* (FIB) [254]. The FIB comprises packet forwarding rules deployed to satisfy network administrator goals. Mismatching packets are discarded or redirected to the *control plane* through the *southbound API*. While the data plane uses complementary functionality of both virtual and physical switches, the role of the latter is often reduced to routing IP-tunneled traffic between hypervisors [95]. In this paper we do *not* address control of hardware switches and traffic routing between hosts; we assume that the physical network provides uniform capacity across hosts, based on e.g. equal-cost multi-path routing [255], such that if multiple equal-cost routes to the same destination exist, they can be discovered and used to provide load balancing among redundant paths. Overlay networks – e.g. VLANs or GRE [256] – are used for communication between endpoints. In this work, we focus exclusively on software switching and use the term "switch" to denote a virtual, software implementation. We refer to hardware switch implementations as "hardware switches".



(a) The SDN architectural model.



(b) Communication paths between collocated endpoints: (1) virtual switch; (2) host-local, e.g. native bridging; (3) virtual queues in the network interface card; (4) external switch, i.e. "hairpin switching".

In the *control plane*, high-level network operator goals are translated into discrete routing policies based on the *global network view*, i.e. a graph representation of the virtual network topology. The main component of a control plane is the *network controller*, which we define as follows:

Definition 2.1. Network Controller (NC) is a logically centralized component that manages network communication in a given deployment by updating the FIB with specific forwarding rules. The NC compiles forwarding rules based on three inputs: the dynamic global network view, the high-level configuration goals of the network operator, and the output of the network management applications.

The NC is typically implemented as part of a logically centralized *network OS*, which builds and maintains the global network view and may include a *network hypervisor*, to multiplex network resources among distinct virtual network deployments.

Southbound API is a set of vendor-agnostic instructions for communication between data and control planes. It is often limited to flow-based traffic control of the data plane, with management done through a configuration database [95].

Network operators use *network management applications* (NMAs), e.g. firewalls, traffic shapers, etc., to configure the network using high-level commands.

2.2 Deployment layers

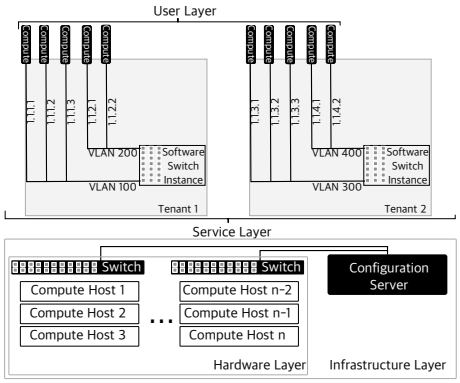
We next describe the deployment layers of SDN infrastructure (Figure F.2a).

The *hardware layer* includes infrastructure for data transfer, processing and storage and is comprised of network hardware (including hardware switches and communication channels), hardware server platforms and data storage.

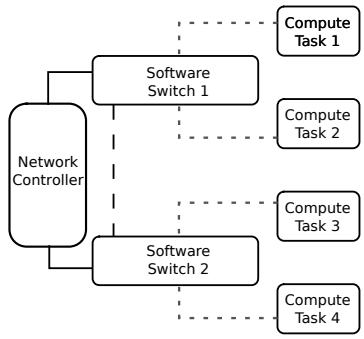
The *infrastructure layer* includes software components for virtualization and resource provisioning to infrastructure users, referred to as *tenants*. For network resources, this layer includes the network hypervisor, which creates *network slices* by multiplexing physical network infrastructure between tenants. Infrastructure providers expose a *slice* (i.e. a quota) of network resources to the tenants.

The *service layer* includes components controlled by tenants. Network components operated by tenants are grouped into *network domains*, comprising the virtual network resources and topologies that logically belong to the same organizational unit and network slice, and perform related tasks or provide a common service. The *network hypervisor* ensures that a tenant's control plane can only control switches in its own slice. Within their slice, tenants have exhaustive creation, destruction and configuration privileges over components, such as instances of switches, the NC, NMAs and network domains. We define three *logical* communication segments (Figure F.2b): between the network controller and switches (α segments); among the switches on each host (β segments); between host-local switches and network endpoints (γ segments).

The *user layer* includes endpoint consumers of network services, e.g. virtualization guests, containers and applications in a network domain.



(a) Deployment Layers.



(b) Logical communication segments: α: between the network controller and switches; β: among the switches on each host; γ between host-local switches and network endpoints.

2.3 Trusted Execution Environments

The proposed solution relies on TEEs that *both* provide strong isolation and allow remote code and data integrity attestation. Such a TEE can be created using Intel SGX enclaves (introduced in [50, 51]) during OS runtime and relies for its security on a trusted computing base (TCB) of code and data loaded at build time, processor firmware and processor hardware. At build time, the CPU measures the loaded code, data and memory page layout. At initialization time, the CPU produces a final measurement, after which the enclave becomes immutable and cannot be externally modified. The CPU maintains the measurement throughout the enclave's lifetime to later assert the integrity of the enclave contents. Processor firmware is the root of trust (ROT) of an enclave. It prevents access to the enclave's memory segment by either the platform OS, other enclaves, or other external agents. Enclaves operate in a separate memory region inaccessible to non-enclave processes, called the enclave page cache (EPC). Multiple mutually distrusting enclaves can operate on the platform. The processor enforces separation of memory access among enclaves based on the layout in the *EPC map*. Program execution within an enclave is transparent to both the underlying OS and other enclaves.

Remote attestation allows an enclave to provide integrity guarantees of its contents [50]. For this, the platform produces an attestation assertion with information about the identity of the enclave and details of its state (e.g. the mode of the software environment, associated data, and a cryptographic binding to the platform TCB making the assertion). For *intra-platform attestation* (i.e. between enclaves on the same platform), the reporting enclave (*reporter*) invokes the `EREPORT` instruction to create a `REPORT` structure with the assertion and calculate a message authentication code (MAC), using a *report key*, known only to the target enclave (*target*) and the CPU. The structure contains a `user data` field, where the reporter can store a hash of the auxiliary data provided. The target recomputes the MAC with its report key to verify the authenticity of the structure, and compares the hash in the `user data` with the hash of the auxiliary data, to verify its integrity. Enclaves then use the auxiliary data to establish a secure communication channel. For *inter-platform attestation* the remote verifier first sends a challenge to the enclave platform, where the challenge is complemented with the indentity of a *quoting enclave* (QE) and forwarded to the reporter, which appends the challenge response

to the `REPORT` and attests itself to the QE. The QE verifies the structure, signs it with a platform-specific key using the *enhanced privacy ID group signature scheme* (EPID) [55] and returns it to the verifier, to check the authenticity of the signature and the report itself [50]. The use of the EPID scheme is part of the SGX implementation and allows to maintain the privacy of the platform which hosts the enclave.

3 Adversary Model

We now describe the adopted adversary model, as well as the core security assumptions on which we base our design. The adversary model we adopt can be described by the capabilities of the adversary at the *network* and *platform* levels respectively (overview in Table F.1).

3.1 Network infrastructure

For SDN infrastructure, we adopt the adversary model introduced in [211] and extended with SDN-specific attack vectors in [250]. We assume a powerful adversary (*Adv*), which controls the cloud deployment network infrastructure; it can intercept, record, forge, drop and replay any message on the network, and is only limited by the constraints of the employed cryptographic methods. Particularly, the *Adv* may forge messages that do not match any of the rules installed in the FIB. Furthermore the *Adv* may create own instances of switches and launch Sybil attacks [257] and launch other types of topology poisoning attacks [252] to distort the global network view. Finally, *Adv* can store arbitrary quantities of intercepted communication and attempt its decryption with encryption keys intercepted or leaked at a later point. It can analyze the traffic patterns in the network through passive probing and may disrupt or degrade network connectivity to achieve its goals. We explicitly exclude Denial-of-Service attacks on the SDN infrastructure.

Table F.1: Summary of the *Adv* capabilities in relation to the adversary model.

Type	*Network*	*Platform*
Included	Intercept, record, forge, drop, replay messages; Analyze the traffic patterns; Disrupt or degrade network connectivity; Launch topology poisoning attacks	Control non-processor hardware; Control software stack OS, hypervisor; Pause execution; Deploy arbitrary software components; "Cuckoo attack": Forward function calls to compromised SGX enclaves; Return arbitrary values to system calls
Not included, mitigations known		Side-channels: cache-collision, controlled channel; Attacks on shielded execution;
Excplicitly excluded	Denial-of-Service (DoS) attacks	Side-channels: power analysis; DoS attacks

3.2 Platform

For platform security, we consider a powerful adversary, similar to [61, 63], that may control the entire software stack in the cloud provider's infrastructure.

On the hardware level, we assume the processor is correctly implemented and remains uncompromised; furthermore, we assume a reliable and secure source of random numbers (which can be provided by the CPU). *Adv* has full control over the remaining hardware, including memory, I/O devices, periferials, etc. Similarly, *Adv* fully controls the software stack, including the platform OS and the hypervisor. This implies that *Adv* may pause indefinitely the execution of the code in the TEE and return arbitrary values in response to OS system calls. However, a deployment orchestrator and NC execute under tenant control, on a fully trusted platform and software stack. We exclude side-channel attacks. While some side-channel attacks – e.g. timing, cache-collision, controlled channel attacks – can be mitigated through software modification [56], preventing other side-channel attacks – such as power analysis – requires hardware modifications. An *Adv* with advanced capabilities may leverage its full control over the OS to utilize the class of known attacks on shielded execution; while we do not address such attacks, they have known countermeasures [60, 61].

SGX, similar to other trusted computing solutions, is vulnerable to *cuckoo attacks* [62]. In one attack scenario, malware on the target platform forwards the messages intended for the *local* SGX enclave (SGX_L^E) to a remote enclave under *Adv*'s physical control (*malicious* enclave, SGX_M^E). Having physical access to SGX_M^E, *Adv* can apply hardware attacks to violate its security guarantees. As a result, *Adv* controls all communication between the verifier and SGX_L^E, with access to an oracle that provides all of the answers a benign SGX^E would, but without its expected security properties.

Briefly, the adversary model for platform security largely matches the remote administrator capabilities of an infrastructure cloud provider.

4 Solution Description

In this section we present TruSDN, a framework for bootstrapping trust in SDN deployments. Its goal is to allow tenants to securely deploy computing tasks and create virtualized network infrastructure deployments, given the adversary model defined in Section 3. To satisfy this goal, the framework must satisfy the following set of requirements:

- *Authentication:* communication in the domain must the authenticated, and a secure enrollment mechanism for data plane components must be in place.
- *Topology integrity:* the NC must be protected from network components that attempt to distort the global network view.
- *Component integrity:* integrity of switches must be attested prior to enrollment and the cryptographic material required for their network access must be protected with a hardware ROT.
- *Confidentiality protection of domain secrets:* network domain secrets – such as VPN session keys – should not be revealed to the *Adv*.
- *Protected network communication:* network communication in the tenant domain must be confidentiality and integrity protected.

4.1 TruSDN overview

We begin by introducing the building blocks of TruSDN (Figure F.3).

Trusted Execution Environments: TruSDN uses TEEs that guarantee secure execution in the given adversary model, assuming the CPU and executed code are correctly implemented.

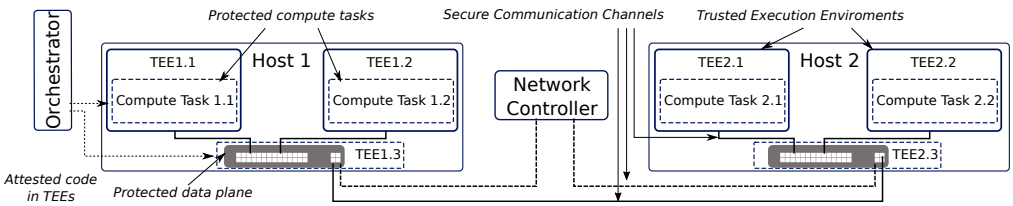


Figure F.3: Illustration of core building blocks of TruSDN.

Protected Compute Tasks: *Security sensitive* compute tasks (CT) are deployed in TEEs. Such tasks include all operations that tenants aim to protect from the *Adv.* However, CTs rely on the untrusted OS for I/O and support functionality.

Protected Data Plane: Switches are deployed in TEEs – they route traffic between CTs according to forwarding rules communicated through secure channels and maintained in the FIB. The FIB of the switches, and the key material necessary to establish the secure channels are stored in TEEs.

Attested code in TEEs: An orchestrator under tenant control attests the TEEs during network infrastructure deployment, to ensure integrity of the deployed code and data before keys or key material are provisioned to the respective TEE.

In a typical deployment scenario, the tenant invokes an orchestrator to deploy a switch *bootstrap application* on the hosts in the tenant's domain. The bootstrap application invokes a host-local SGX driver to build an SGX enclave containing a switch. Next, the orchestrator attests the created enclave (as described in Section 2.3) prior to enrolling the switch with the NC. The orchestrator uses the enclave's public key from the attestation quote to securely transfer the enclave-specific integrity and confidentiality protection session keys used to establish a protected communication channel between the NC and the TEE. Finally, the NC communicates any remaining security-sensitive payload to the created TEE, e.g. the initial FIB. Next, CTs are deployed in TEEs on the host and the switch forwards packets between the CTs, matching them against the rules in the FIB. Mismatching packets are forwarded to the NC, which may update the FIB with new rules. For clarity, we assume the orchestrator and NC are collocated on a platform under tenant control and view both as a single component, further referred to as "NC".

Secure Communication: TruSDN protects the communication between CTs, between switches and the NC, as well as among the switches, in the above adversary model. Communication security is ensured using confidentiality and integrity protection keys provisioned to authenticated network components and endpoints executing in TEEs. Furthermore, TruSDN leverages SDN principles to introduce a novel mechanism – per-flow communication protection using ephemeral flow-specific pre-shared keys (PSKs).

4.2 Cryptographic Primitives

We now define the cryptographic primitives and notations used in the remainder of this paper. We denote by $\{0,1\}^n$ the set of all binary strings of length n, and by $\{0,1\}^*$ the set of all finite binary strings. In a set U, we refer to the i^{th} element as u_i, and use the following notation for cryptographic operations:

- Given an arbitrary message $m \in \{0,1\}^*$, we denote by $c = \mathsf{Enc}(K, m)$ a symmetric encryption of m using the secret key $K \in \{0,1\}^*$. The corresponding symmetric decryption operation is $m = \mathsf{Dec}(K, c) = \mathsf{Dec}(K, \mathsf{Enc}(K, m))$.
- We denote by pk/sk a public/private key pair for a public key encryption scheme. We denote by $c = \mathsf{Enc}_{\mathsf{pk}}(m)$ the encryption of message m with the public key pk, and the decryption by $m = \mathsf{Dec}_{\mathsf{sk}}(c) = \mathsf{Dec}_{\mathsf{sk}}(\mathsf{Enc}_{\mathsf{pk}}(m))$.
- We denote a digital signature over a message m by $\sigma = \mathsf{Sign}_{\mathsf{sk}}(m)$ and the corresponding verification of a digital signature by $\nu = \mathsf{Verify}_{\mathsf{pk}}(m, \sigma)$, where $\nu = 1$ if the signature is valid and $\nu = 0$ otherwise.
- We denote a Message Authentication Code (MAC) using a secret key K over a message m by $\mu = \mathsf{MAC}(K, m)$.

We next describe key sharing and communication protection mechanisms on the identified logical segments. Table F.2 summarizes the keys used by TruSDN.

Table F.2: Summary of keys used in the TruSDN framework.

Key	*Created by*	*Access*	*Usage*
K_i^{α}	NC	NC, switch	Enclave-specific session, segment α
K_j^{β}	NC	NC , switch	Domain-specific session, segment β
K'	NC	NC, switch	Ephemeral session key
K''	NC	NC, switch	Ephemeral MAC key
EK_i^{pk}	switch	public	Public key of the switch enclave
EK_i^{sk}	switch	switch	Private key of the switch enclave
CK_i^{pk}	CT	public	Public key of the compute task
CK_i^{sk}	CT	CT	Private key of the compute task
QE^{pk}	vendor	public	Public key of the quoting enclave
QE^{sk}	vendor	vendor, QE	Private key of the quoting enclave
SK_{ij}^{γ}	NC	NC, CT_i, CT_j	Ephemeral flow-specific pre-shared key

4.3 SDN Trust Bootstrapping and Secure Communication

The first step in deploying a TruSDN infrastructure is to launch a set of trusted switches for connectivity and topology building. The NC requests the creation of switch enclaves to

deploy switches in TEEs on hosts in its domain. Switches are deployed based on parameters provided by the NC in plaintext (application code and configuration). Next, the NC *attests* the integrity of switch enclaves and only *enrolls* the successfully attested ones (Figure F.4). A TEE E_i is attested following the protocol introduced in [50]. With TruSDN however, the reporter generates an enclave-specific public-private keypair and submits its public key EK_i^{pk} along with the attestation data; a hash of the public key is stored in the `user data` field. The switch enclave is only enrolled to the global network view if its reported state matches the one expected by NC.

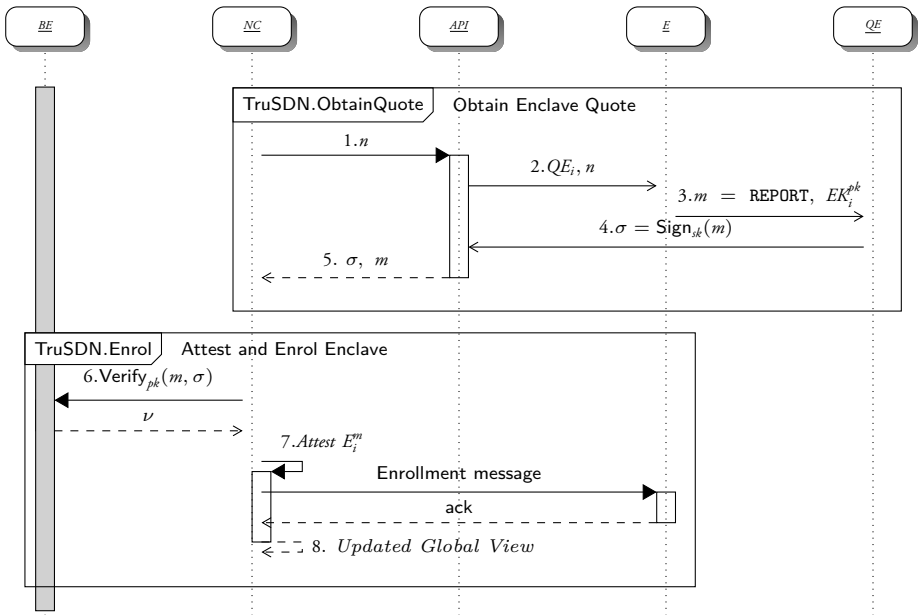


Figure F.4: TruSDN enclave attestation and enrollment: (1.) Random nonce n is (2.) supplemented with the host QE identity; (3.) Quote m produced by the enclave is (4.) signed by the QE. (6.) The verifier checks the signature of the QE, (7.) attests the integrity of the enclave and (8.) only enrolls the enclave upon success. *BE*: back-end.

Having attested enclave E_i, NC communicates an Enrollment message (Table F.3) with the enclave-specific pre-shared key K_i^{α} and domain-specific pre-shared key K_j^{β}, encrypted with an ephemeral key K_i'. Switches within a domain use K_j^{β} to protect communication on β segments. The *NC* appends a MAC of the message calculated with K_i'' and encrypts the keys K_i', K_i'' with EK_i^{pk}.

Once switches are deployed and enrolled, tenants may configure the network topology using the NC to update the switch FIBs. Communication on α segments – e.g. FIB updates or unmatched packets forwarded to the NC – is protected using the session key K_i^{α} (e.g. using TLS [215]), which never leaves the TEE.

Similarly, a secure channel is established among the switches within the same domain, using the pre-shared key K_j^{β}, to protect communication between switches on different hosts (e.g. TEEs 1.2 and 2.3 in Figure F.3). K_j^{β} never leaves the TEEs, has a limited validity time and

Table F.3: Enrollment message sent by the NC upon switch enrollment.

$m = \mathsf{Enc}(K_i', (K_i^{\alpha}, K_j^{\beta}))$	$\mu = \mathsf{MAC}(K_i'', m)$	$\mathsf{Enc}(EK_i^{pk}, (K_i', K_i''))$

is periodically redeployed by the NC. On β segments, traffic may traverse multiple hardware switches, forwarded to the host over tunnels deployed on top of a standard routing protocol (e.g. [255]).

Next, the tenant may deploy CTs in TEEs and attest their integrity using the very same scheme and principles as for the switch deployment described above. The CTs and the network controller use the Enrollment message to establish a secure communication channel (e.g. TLS).

Once the NC has deployed and attested the TEEs with switches and CTs, intra-host communication (i.e. between two CT enclaves on the same host) is straightforward (Figure F.5): when a packet m sent from C1 (e.g. a TLS `ClientHello` message) reaches the local host *switch A*, it attempts to match m against a FIB entry; if no suitable flow rule f is present, the switch forwards $\mathsf{Enc}(K_A^{\alpha}, m)$ to NC, which processes the packet, generates and deploys on the CTs C1, C2 a flow-specific pre-shared key SK_{12}^{γ} and finally updates the switch FIB with f, after which steps 2 and 3 are ignored; once the FIB is updated, the switch forwards m to C2, which continues the message exchange and uses SK_{12}^{γ} to protect the communication with C1, using e.g. TLS with a PSK ciphersuite [215].

Communication between CTs C1 and C3 deployed on distinct hosts is similar, with the only notable difference that the NC updates the FIB of the local switches on both hosts where C1, C3 are deployed.

In the above scenarios TruSDN leverages two aspects of the SDN model – **(1)** the deployment has a central authority (the NC) and **(2)** the first packet of a flow is forwarded to the central authority – to deliver on demand ephemeral PSKs to communication endpoints. This allows to relax the need for high-quality entropy being available to CTs (a known issue in virtualized environments [258]). Furthermore, this approach ensures communication security without compromising packet visibility – having control over the keys used to protect communication between the CTs allows the NC to maintain fine-grained insight into the traffic.

4.4 Preventing Cuckoo Attacks

To prevent cuckoo attacks [62], we propose a solution that leverages cryptographic properties of the EPID scheme used by the QE [55] and the *SIGn and Message Authentication* (SIGMA) protocol [259], which are both part of the Intel SGX implementation. The EPID scheme supports two signature modes: *fully anonymous mode* – the verifier cannot associate a given signature with a particular member of the group; *pseudo-anonymous mode* – the verifier can determine whether it has verified the platform previously. The unlinkability property distinguished in the two modes depends on the chosen *base*. A signature includes a pseudonym B^{f}, where B is the base chosen for a signature and revealed during the signature; f is unique per member and private. For a *random base* R, the pseudonym is R^{f} – in this case the signatures are unlinkable. For a *name base*, the pseudonym is N^{f}, where N is the name of the verifier – in this case the signatures remain unlinkable for *different* verifiers, while signatures with a common N can be linked. For privacy reasons, the EPID scheme currently implemented in

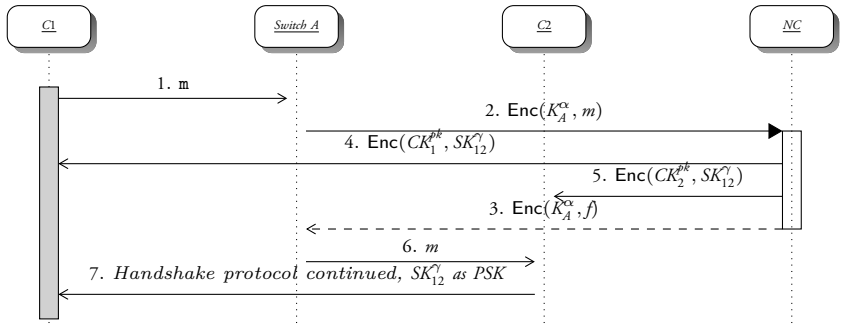


Figure F.5: Intra-host communication with TruSDN.

Intel SGX accepts *name base* pseudonyms only from verifiers authorized by the EPID authority [260], which is done by provisioning qualified verifiers with an X.509 certificate – e.g. an intermediate certification authority (CA) certificate – signed by the EPID authority acting as root CA.

We propose the following algorithm to prevent cuckoo attacks. At deployment time, the EPID authority issues, to an authorized verifier V_P, an intermediate CA verifier certificate for the platforms in the cloud provider's data center. Next, V_P attests its platforms following the SIGMA protocol and publishes a list of resulting platform EPID signatures and the signature name base, $\mathsf{B}_\mathsf{P}^\mathsf{N}$. To guard against cuckoo attacks, tenants first request V_P to issue an X.509 certificate and enable them to become *authorized verifiers*. Next, tenants choose the same pseudonym base $\mathsf{B}_\mathsf{P}^\mathsf{N}$ (and a private f), follow the SIGMA protocol, and verify that the resulting signature is linkable to a signature in the published list. The cloud provider has multiple tools to protect platform privacy and prevent untrusted tenants from fingerprinting the platform infrastructure, e.g. limiting the validity of issued certificates, changing the name base, etc. Considering that the EPID scheme is currently not implemented in the SGX emulation software we used for prototyping, we intend to describe the implementation of the above algorithm in a follow-up report.

5 Security Analysis

In this section we analyze the security properties of the proposed framework in the adversary model described in Section 3. On the network level, many of the *Adv* capabilities are thwarted by first authenticating the switches deployed on the data plane, as well as the network edge (i.e. the compute tasks that generate or receive the network traffic), in combination with confidentiality and integrity protection of the traffic on the three identified segments. Authenticating the network components prevents topology poisoning attacks (a countermeasure mentioned in [252]), while confidentiality and integrity protection of all of the network traffic in the deployment prevents the *Adv* from either learning the contents of the exchanged packets or successfully forging packets. The *Adv* may in this case still intercept and record messages. However, collecting encrypted traffic does not yield the *Adv* any more information about the contents of the exchanged packets. Similarly, the *Adv* does not gain an advantage by simply dropping or replaying messages, since these actions would at most simply reduce the channel

capacity (as would the ability of the *Adv* to disrupt network connectivity). Finally, the proposed framework does not prevent the *Adv* from analyzing the traffic patterns and does not prevent it from fingerprinting the components of the deployment, making it vulnerable to rule scanning and denial of service attacks. While the goals of TruSDN did not include this, such traffic analysis could be prevented using anti-fingerprinting techniques, as proposed in [261].

On the platform level, the security of the proposed framework relies to a large extent on the security properties of Intel SGX enclaves. This allows to protect the execution of switches and network edge components deployed in TEEs from the capabilities of an *Adv* controlling non-processor hardware, the software stack of the OS and the hypervisor. Similarly, pausing execution of switches executing in TEEs, while possible, would have no further effect than degrading network connectivity, already discussed above. While the *Adv* may attempt to deploy own arbitrary components on the data plane or the network edge in order to launch Sybill attacks, the integrity of such components would not be successfully attested, unless they are identical to legitimate components, which are assumed to be executing correctly – rendering Sybill behavior impossible. The *Adv* is prevented from launching cuckoo attacks by enabling tenants to verify the platforms, as described in Section 4.4. As presented in Table F.1, several relevant classes of attacks are not addressed by TruSDN, but have known mitigations, namely cache-collision, controlled channel and attacks on shielded execution (addressed in [56, 63]). The capability of the *Adv* to return arbitrary values to system calls, while not addressed in this work, can be mitigated by a validation component as described in [61].

6 Implementation and Evaluation

We now describe the implementation and evaluation of TruSDN.

6.1 TruSDN Implementation

The TruSDN prototype deployment follows the design presented in Section 4 and is illustrated in Figure F.6. *Host 1* and *Host 2* are instances of Ubuntu OS 15.04. In each instance, we deployed Linux Containers[1], similarly based on Ubuntu OS 15.04. Containers create an environment with own process and network space, implemented using *namespaces*, with a distinct user ID, network stack, mount points, file systems, processes, inter-process communication, and hostname. We chose containers to facilitate prototype implementation, using their lightweight process isolation. Containers are part of the untrusted OS and this implementation choice is orthogonal to the security of TruSDN. Compute tasks are deployed in TEEs created using SGX enclaves (Figure F.6): *enclaves* E1, E2, E4, E5 are placed respectively within containers C1, C2, C3, C4. The switches are deployed in TEEs created using SGX enclaves (enclaves E3, E6 in Figure F.6).

Considering that platforms with hardware and software support for SGX were not publicly available at the time of writing, we used *OpenSGX* [184] to emulate the TEEs. It is a software SGX emulator and a platform for SGX development, implemented using binary translation of QEMU and emulating Intel SGX hardware components at instruction level. It includes emulated hardware and OS components, enclave program loader, the OpenSGX user libraries, debugging and performance monitoring support. The emulator allows to implement, debug,

[1] Linux Containers Project Website: `https://linuxcontainers.org/`

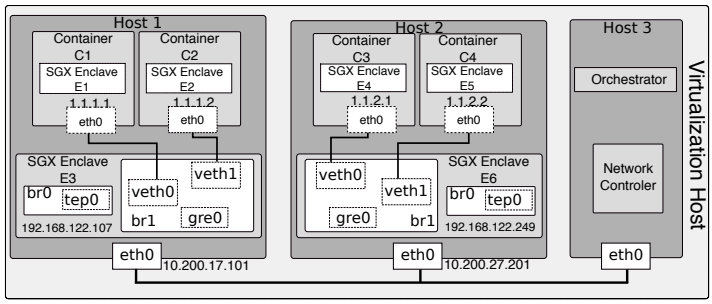


Figure F.6: Prototype deployment of TruSDN

and evaluate SGX applications, but does not support binary compatibility with Intel SGX. Furthermore, OpenSGX does not implement all instructions, e.g. debugging instructions. While OpenSGX does not provide security gurantees, it allows us to obtain performance estimates for the proposed approach. We used *mbedTLS*[2] v1.3.11 (distributed with the emulator) for attestation of the SGX enclaves. We used OpenSSL v1.0.2d (distributed with the emulator) to set up protected communication channels between the CT enclaves and the local switches, and among switches within the same domain.

An SDN network controller is deployed in a third instance (*Host 3*). We used the *Ryu*[3] SDN framework, due to its flexibility and versatile APIs.

6.2 TruSDN Evaluation

We now analyze the performance impact, present evaluation results and discuss aspects that cannot be measured with the current prototype.

Sources of Performance Impact

TruSDN introduces several potential sources of performance impact (Table F.4). We distinguish between *transient* performance overhead, which occurs occasionally (e.g. TLS key negotiation) and *continuous* performance overhead, present throughout the infrastructure operation. We *do not* consider the one-time cost of infrastructure deployment, e.g. provisioning the software, attesting TEEs and enrolling the components.

Measured Performance Impact

To evaluate the performance impact, we measured the footprint of establishing TLS sessions on α and γ segments. We used *iperf*, *openssl s_time* and an own Ryu application (Table F.5).

[2]mbed TLS project website `https://tls.mbed.org/`
[3]Ryu SDN framework: `https://osrg.github.io/ryu/`

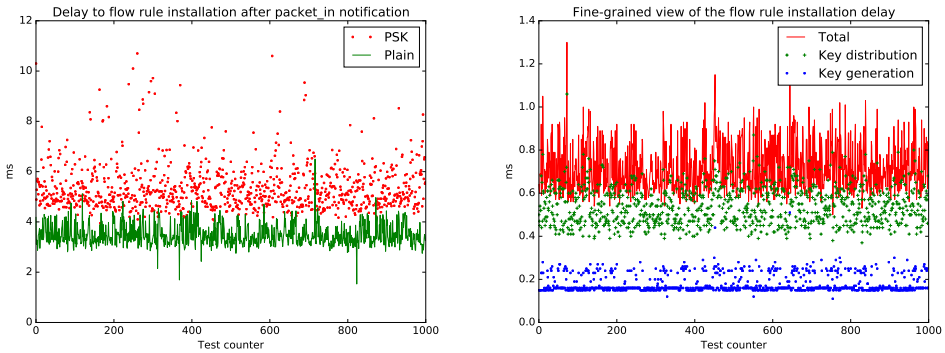
Table F.4: Sources and types of performance overhead in TruSDN

Source	*Type*	*Clarification*
TLS negotiation all segments	transient	Negotiate session keys for TLS
PSK distribution	transient	Distribute PSK for γ segments
TLS protection all segments	continuous	Overhead induced by TLS
Compute task execution in TEEs	continuous	Overhead induced by TEE
Switch execution in TEEs	continuous	Overhead induced by TEE

TLS overhead on the α segment: We measured the round-trip latency of packets sent in plaintext and with TLS, over 1000 tests, each request sending messages of 100 bytes with the 80 bit OpenFlow header. Furthermore, we measured the data transfer rates for plaintext and TLS communication. Use of TLS increased total transfer time by 14.2% and reduced the transfer rate by 15.98%.

Delay on γ segment As mentioned above, the first packet of the flow is intercepted by the switch and forwarded to the NC in a `packet_in` message [96]. At this point the NC processes the flow and installs a flow rule on the switch. TruSDN *extends* this procedure by generating and distributing to the communicating CTs a pre-shared key, to be used for communication protection. Since this must be done prior to both forwarding the message to the destination CT and installing the flow rule, generating and distributing the PSK would normally delay the installation of the flow rule and increase the latency of the *first* packet (all subsequent packets are forwarded according to the flow rule). To measure the introduced delay, we have sequentially established 1000 TLS sessions between compute tasks C1 and C2 (according to Figure F.6). After each TLS session, we flushed the installed flow rules (with `ovs-ofctl del-flows br0`), which resulted in a `packet_in` message upon each new session. The latency of the first packet is shown in Figure F.7a, and compared against the latency of a first packet without the TruSDN extension.

The induced delay is primarily caused by two operations performed by the NC: *generating* a 256-bit PSK and distributing it to the CTs. Figure F.7b displays a fine-grained picture of the induced delay. Key generation lasted on average 0.178 ms, while key distribution on average 0.54 ms (Table F.5). We remind that the test environment is fully virtualized and posit that overhead of key generation can be reduced in a production environment, either by using pre-generated keys or with specialized hardware (e.g. crypto processors). In our tests, the duration of establishing a TLS session with ephemeral flow-specific pre-shared keys using the PSK-AES256-CBC-SHA cipher suite was 2.41% *less* compared to the use of e.g. ECDH-RSA-AES128-SHA256. Thus, TruSDN enables flexible use of pre-shared keys, which in turn reduces the duration of the TLS handshake, by avoiding expensive public key cryptographic operations [262]. Moreover, it reduces the CPU utilization for key derivation in CTs, at the cost of a minimal flow rule installation delay. The above approach may be applicable to other protocols. For example, none of the differences between the datagram TLS (DTLS) and TLS protocols specified in [263] indicate that the above approach is incompatible with DTLS. We leave further investigation for future work.



(a) Coarse-grained view

(b) Fine-grained view

Figure F.7: Performance evaluation of TruSDN

Unmeasured Performance Overhead

Implementing TEEs with OpenSGX limits the level of detail when it comes to performance evaluation, since: **(a)** the OpenSGX emulator *is not* binary compatible with Intel SGX [184]; **(b)** in its current version[4] and unlike Intel's description of SGX [50], OpenSGX has yet to implement support multithreaded applications[5]. Thus, a fully accurate measurement on TruSDN performance cannot be done until Intel SGX hardware and software is made available. However, we believe our experiments yield a fair picture of the expected performance impact.

7 Related work

Adversary models: Kreutz et al. presented a list of attack vectors in SDN [155] (forged traffic flows, vulnerabilities in switches and NCs, lack of trust establishment mechanisms, etc.). However, only part of the described attack vectors are exclusively relevant to SDN networks and no specific solutions are proposed. Work in [250] introduced an adversary model, attack vectors, and security requirements towards multi-tenant SDN infrastructure, highlighting the need to limit the effect of NC vulnerabilities, protect internal SDN communication, verify integrity of SDN components prior to enrollment, and enforce policy and quota isolation. TruSDN addresses several of the attack vectors described in [155, 250].

Secure SDN controllers: The "NOX" network OS [98] presents NMAs with a centralized programming model, allowing to operate with higher-level abstractions and apply graph processing algorithms to compute paths. It consists of several controller processes which use the global view for network management decisions and update switch FIBs over the OpenFlow API [246]. FortNOX [158] extends NOX with role-based authorization (RBA) and enforcement of security constraints. It translates high-level threats into flow rules to handle suspi-

[4]Commit e0713c7 on `https://github.com/sslab-gatech/opensgx`

[5]Issue #34 on `https://github.com/sslab-gatech/opensgx/issues/34`

Table F.5: Summary of performance evaluation of TruSDN

Data	*Minimum*	*Maximum*	*Mean*	*Median*	*Stddev*
Total transfer time, ms	0.4	1.1	0.66	0.7	0.07
Total transfer time w. TruSDN, ms	0.5	7.1	0.8	0.8	0.22
TruSDN overhead, **total transfer time**			**21.2%**	**14.2%**	
Transfer rate, bytes per second	1225	2095	1595	1583	98.07
Transfer rate w. TruSDN, bytes per second	919	1589	1338	1330	64.86
TruSDN overhead, **transfer rate**			**16.11%**	**15.98%**	
First packet latency γ	1.53	6.50	3.48	3.38	0.42
First packet latency γ w. TruSDN	3.35	10.7	5.37	5.14	0.93
TruSDN overhead, **first packet latency**			**54.31%**	**52.07%**	
TLS handshake, ms	36.53	77.72	67.97	67.48	7.42
TLS handshake w. TruSDN, ms	52.35	76.44	67.15	66.53	3.93
TruSDN overhead, **TLS handshake**			**-2.21%**	**-2.41%**	
Key generation NC, ms	0.11	0.51	0.178	0.16	0.04
Key distribution γ, ms	0.37	1.06	0.54	0.53	0.08
Key total γ, ms	0.50	1.30	0.71	0.7	0.11

cious traffic as well as detects rule conflicts, resolves them depending on the authorization of the rule requestor and enforces least privilege authorization. Neither NOX nor FortNOX address malicious network components and Sybill attacks, addressed by TruSDN. "Rosemary" NOS [99] uses NMA sandboxing to improve network resilience, by launching each NMA in a separate process context with access to the required libraries, along with a resource monitor to supervise NMA compliance. It does not address data plane security; TruSDN complements it and creates a foundation for trusted deployment of a secure NOS. TopoGuard [252] detects network topology poisoning and mitigates this through port property management, network edge probing and verification of topology updates. TruSDN complements this by verifying the *integrity* of switches prior to enrollment into the topology.

Software Guard Extensions: SGX was introduced in [51] with a description of the software model, extensions to the x86 ISA and hardware modifications for isolated execution; work in [50] described CPU based attestation. SGX-based solutions in a cloud setting are first described in [61, 63]. "Haven"[61] is a modified version of Windows 8 OS ported to an SGX enclave, evaluated with Apache Web Server and SQL Server using synthetic data sets. It includes a mechanism to protect the enclave from a malicious kernel and a semantically secure data store protecting data and file metadata confidentiality against malicious hosts. TruSDN protects network communication for a similar adversary model. While we deploy compute tasks in SGX enclave-based TEEs, the work in [61] is largely complementary, and similar "Haven"-like OSs could be used.

"VC3" [63] is a Map-Reduce deployment using SGX enclaves. *Map* and *reduce* functions are compiled into private (encrypted) code and public code implementing key exchange and job execution protocols. Code is initialized in enclaves and attested by the users. Public code performs the key exchange, decrypts the private code and runs the job execution protocol. To defend against cuckoo attacks, *cloud quoting enclaves* are created on *each* platform in the cloud provider data centers, to "countersign" quotes produced by the QE. The approach is largely complementary to protecting communication between CTs with TruSDN. However, the proposed defense against cuckoo attacks increases the complexity of the attestation protocol and does not prevent *Adv* from exploiting a compromised cloud QE outside of the physically secure datacenter perimeter. Instead, the approach described in Section 4.4 leverages the

cryptographic properties of EPID scheme, without modifying the attestation protocol.

8 Future Work

Along with security guarantees, the use of Intel SGX imposes limitations on TruSDN. Further performance evaluation may be done once software and hardware support for Intel SGX becomes available; moreover, we note several security limitations. Controlled-channel attacks [56] are a novel type of side-channel attacks allowing the OS to extract data from protected applications. They were successfully applied to "Haven" [61] and TruSDN could also be vulnerable; however, we explicitly excluded such attacks from the adversary model. Known mitigations are: rewriting applications to decouple memory access patterns from sensitive data, prohibiting paging by the OS, or obfuscating memory access patterns [56]. Another limitation stems from the reliance on the platform vendor, which could leak QE^{sk}, to create a "deniable back-door" and allow *person-in-the-middle* attacks on attestation [264]. This challenge remains unaddressed.

In future work we aim to integrate TruSDN with other approaches to cloud infrastructure security, such as in [265], to provide a complete framework for secure cloud infrastructure deployments in the given adversarial model.

9 Conclusion

We described, implemented and evaluated TruSDN – a framework for bootstrapping trust in SDN infrastructure. It isolates network endpoints and switches in SGX enclaves, remotely attests their integrity, and establishes secure communication channels. We leveraged the principles of SDN to introduce *ephemeral flow-specific PSK* distributed at flow creation, which reduce the overhead of key derivation and reduce the total time to establish protected channels, at the cost of a minor delay in the flow rule installation. Finally, we leveraged the properties of the EPID scheme to propose an improved approach to prevent cuckoo attacks.

Paper G

SDN Access Control for the Masses

Nicolae Paladi and Christian Gehrmann

Abstract

The evolution of *Software-Defined Networking* has so far been predominantly geared towards defining and refining the abstractions on the forwarding and control planes. However, despite a maturing south-bound interface and a range of proposed network operating systems, the network management application layer is yet to be specified and standardized. It has currently poorly defined – if any – access control mechanisms that could be exposed to network applications. Available mechanisms allow only rudimentary control and lack procedures to partition resource access across multiple dimensions. We address this by extending the software-defined networking north-bound interface to provide control over shared resources to key stakeholders of network infrastructure: network providers, operators and application developers. We introduce a taxonomy of software-defined networking access models, describe a comprehensive design for software-defined networking access control and implement the proposed solution as an extension of the ONOS network controller intent framework.

1 Introduction

In recent years, research focus on software-defined networking (SDN) has shifted from the maturing forwarding plane abstraction [95, 266] and corresponding south-bound application programming interfaces (APIs) [246, 267], towards richer network control, management and functionality, as well as initial attempts to define a north-bound API. However, the field has advanced unevenly. Currently available north-bound APIs introduce new security risks [172],

and the lack of *network resource access control* is one vivid example of this. Security research in SDN has focused on managing access using topology-specific, low-level resources – such as switch ports and fine-grained bit-matching of packet flows. While this approach is valid and necessary, it does not meaningfully address the needs of either network application *developers* (who create applications operating with network functionality on a higher level of abstraction) or network *operators* (who must consider the security and resource management implications of deploying such applications). As a result, developers face a choice between two undesirable alternatives: assume during development that applications have full, exclusive, and continuous access to deployment resources, or develop custom applications with in-built awareness of the network topology. However, development perils do not end there: in either case network applications operate with *low-level* device resources, mostly unfamiliar to system developers. Finally, at deployment time, operators lack an overview of the resource access granted to network applications once they have been deployed.

Recent efforts aim to provide a declarative paradigm for implementation - independent interaction between network service consumers and providers [268]. However, such efforts do not currently include any access control mechanisms beyond specifying service resource constraints, mostly aimed towards satisfying Quality-Of-Service network provider policies. Several important prior contributions to SDN access control [172,175,269] do not provide the necessary abstraction level to allow exposing access control functionality to potentially malicious SDN applications. We address the above issues by introducing a novel taxonomy of access models for SDN infrastructure resources and by describing and implementing a North-bound Access Control API (NACA) enforcement mechanism for SDN deployments.

For developers, NACA brings simple, clear and usable tools to declare resource requirements of their network applications.

For operators, the API provides the tools to (1) obtain an overview of the access to network resources provided to the applications deployed on the SDN infrastructure and (2) assess the security implications of deploying network applications, considering the resource access they require. This fills the gap in the available tools for managing access control to device resources in SDN deployments and helps answer common questions – such as "Which applications can read the complete topology of the network?", or "Which applications can modify network flows and in which ways?" – that are currently difficult to answer in a given SDN deployment; (3) limit the extent of application access to network resources through resource-specific policies.

To implement the API on the control plane[1], we build our approach on best practices from the fields of operating system security and programming language security – such as code signing for origin verification; code integrity verification; capability-based access control, use of a reference monitor, etc. – to enforce access restrictions to SDN infrastructure resources. Furthermore, we propose leveraging recent developments in execution isolation in order to ensure the robustness of the deployment in the face of a powerful adversary.

In a nutshell, the proposed approach is as follows. We adopt the common access control convention and identify two types of entities: *subjects* – network applications[2], often referred to as "Virtual Network Function" (VNFs), and *objects* – network resources. NACA *access masks* are policies that describe the objects that a certain subject can access, as well as the types of and constraints on actions that can be performed on the object attributes. Examples of *types*

[1] In the remainder of this paper, we use the terms *forwarding plane, control plane, management plane, application plane* as defined in [270].

[2] And implicitly their users

of actions on objects are reading statistics, modifying configuration state, or subscribing to notifications. Examples of *object attributes* are geographical placement of objects, temporal limitations, execution environment visibility, etc. Resource access *requirements* declared by a subject are combined with operator policies to define a distinct immutable *access mask* that persists throughout the lifetime of subject instances. Next, we reliably *tag* requests issued by subjects with their assigned access mask and use a *reference monitor* to ensure that access to the *network information base* is only granted to requests satisfying the constraints of the access mask. In contrast to previous work in this area, NACA does not depend on the use of a particular south-bound-API. It operates entirely on the north-bound API and the implementation on the controller plane. Our contribution is as follows:

- We introduce a taxonomy of access models for network infrastructure resources;
- We introduce a North-bound Access Control API for network infrastructure operators and application developers;
- We propose a novel access control enforcement mechanism for network resources in the software-defined networking model;
- We describe an implementation of the proposed API on the control plane;
- We demonstrate the feasibility of the solution through an integration with an open source SDN controller platform.

The remainder of the paper is organized as follows: in Section 2 we introduce the system and adversary model; next, in Section 3 we describe the north-bound access control API and its internals, followed in Section 4 by a detailed description of the implementation with the ONOS SDN platform and the evaluation results in Section 5. We review the related work in Section 6, outline future work in Section 7 and conclude in Section 8.

2 System and Adversary model

2.1 System model

The software-defined networking model aims to separate the network *forwarding plane* – i.e. the collection of network devices responsible for forwarding traffic – from the control plane – i.e. a collection of functions controlling network devices, defining the network topology and network connectivity policies [165, 270]. Figure G.1 illustrates a high-level architecture of the software-defined networking model.

The forwarding plane includes hardware and software *switches*. Early SDN models envisioned switches that are optimized for forwarding performance, lack decision logic and only forward packets matching *flow definitions* – i.e. packet forwarding rules – in their *forwarding information base* [254]. Later contributions delegate more functional responsibility to switches, while maintaining the capability to selectively upstream packets (or packet data) to controllers [97]. Mismatching packets are discarded or redirected to the *control plane* through the *south-bound API* – a set of vendor-agnostic instructions for communication between forwarding and control planes; this API is often limited to flow-based *traffic control* of the forwarding plane, while *management* of the forwarding plane is done through a configuration database [95].

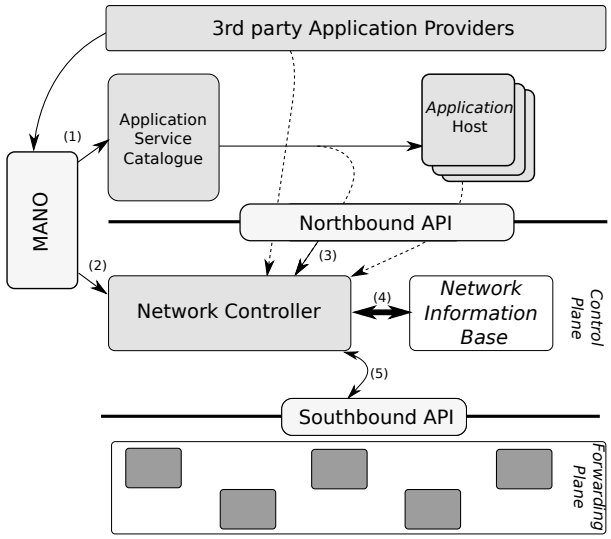


Figure G.1: The SDN architectural model: (1) populate service catalogue; (2) deploy applications; (3) applications interact with network controller and (4) query the NIB; (5) Network controller performs configuration actions on forwarding plane.

On the *control plane*, network operator goals are translated into discrete routing policies based on the *global network view*, e.g. a graph representation of the network topology. A core element is the *network controller* – a logically centralized component that manages network communication in a deployment by updating the FIB with specific forwarding rules. The network controller compiles forwarding rules based on three inputs: the (dynamic) global network view, the configuration goals of the network operator, and the output of the installed *network applications*. The network view built by the controller is maintained in a *network information base* (NIB) [245]. This may either include the entire network topology or a *slice* of it (e.g. in multi-tenant deployments).

The NIB describes all resources of the SDN deployment reachable by the controller. *We use a broad definition of the term resources*, to encompass software components used to achieve network communication goals (e.g. virtual or physical switches), *information* about such network components, and interactions involving them. From a network application point of view, we distinguish three resource categories: *device resources*, e.g. forwarding plane components; *data resources*, e.g. network topology, flow statistics, forwarding logic; and *control resources*, e.g. management policies (Figure G.2).

Operators use network management applications to implement network functionality using high-level commands. Network applications – also known as "middleboxes" – often appear as hardware components in traditional networks; however, alternatives such as VNFs – e.g. software implementations of firewalls, traffic shapers, etc. – are better suited for dynamic SDN deployments and hence are becoming increasingly popular. Applications communicate with the network controller and are used for network management, based on operator-defined policies and network state.

There is currently no single widely adopted interface between applications and network control-

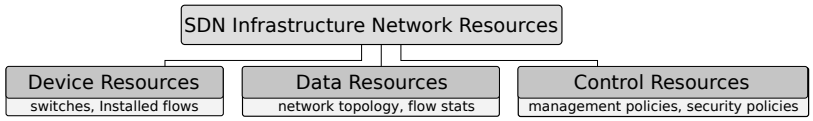


Figure G.2: SDN infrastructure network resources.

lers (i.e. a "north-bound API"). Multiple distinct implementation-specific interfaces are used by network controllers [93, 94]. We distinguish three emerging network application deployment models:

1. *Locally installed applications*, developed in-house or deployed through e.g. "SDN App Stores" [271].
2. *Managed applications*, operating in an "Software-as-a-Service" model, i.e. on the premises of a network function provider [272, 273].
3. *Hybrid applications*, where a back-end executing on the application provider premises interacts with a front-end on the network provider infrastructure.

To facilitate function isolation, scalability and deployment flexibility, applications are commonly deployed as virtualized components, in e.g. virtual machines or containers. We define *candidate applications* as the applications available for deployment from the service catalogue, which can contain both complete application images for locally installed applications or configuration definitions in the managed applications.

A *management and network orchestration* (MANO) component monitors the SDN infrastructure and takes actions to ensure availability and satisfy performance requirements. Such actions include component creation, deployment, migration and destruction. A candidate application becomes an *installed application* once it is granted *access* to the SDN resources described in the NIB.

We define *access by an application to SDN resources* as the capability to execute commands on device resources or modify their state; create, read or write control resources and data resources. Applications exercise access through *requests* submitted to the controller over the north-bound API; the requests are further compiled into a limited number of queries by the network controller and submitted to the network information base. A network controller can only issue queries following a corresponding request by a network application.

2.2 Adversary model

We next describe the adversary model (illustrated in Figure G.3), along with core security assumptions on which we base our design. The adversary (*Adv*) controls the applications installed on the network slice and can request access to arbitrary device resources. Furthermore, the *Adv* can collude several applications to achieve a defined purpose, e.g. take over the device resources allocated to benign applications (provoking a Denial-of-Service attack on them). It can intercept, record, forge, drop and replay any message on its network slice and is only limited by the constraints of the employed cryptographic methods. Furthermore, it can analyze network traffic patterns through passive probing and may disrupt or degrade network connectivity to achieve its goals. The adversary can craft malicious packets to exploit

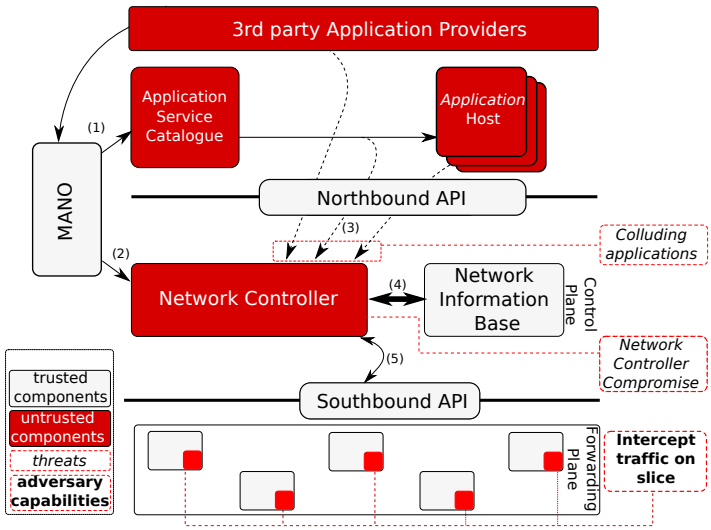


Figure G.3: Adversary capabilities

vulnerabilities in the request processing functionality of the network controller: an adversary may use a network application to submit malicious requests to trigger escalation of resource access permissions for the respective application. However, the *Adv* can only interact with the NIB through queries produced by the network controller, based on requests issued by installed network applications. Briefly, the capabilities of the adversary are similar to the ones of a malicious application provider or operator, whose applications are installed on a network slice in a "Network-as-a-Service" provisioning model.

2.3 Notation and Cryptographic Primitives

The set of all binary strings of length n is denoted by $\{0,1\}^n$, and the set of all finite binary strings as $\{0,1\}^*$. Given a set U, we refer to the i^{th} element as u_i. Additionally, we use the following notations for cryptographic operations:

- For an arbitrary message $m \in \{0,1\}^*$, we denote by $c = \mathsf{Enc}(K, m)$ a symmetric encryption of m using the secret key $K \in \{0,1\}^*$. The corresponding symmetric decryption operation is denoted by $m = \mathsf{Dec}(K, c) = \mathsf{Dec}(K, \mathsf{Enc}(K, m))$.
- We denote by pk/sk a public/private key pair for a public key encryption scheme. Encryption of message m under the public key pk is denoted by $c = \mathsf{Enc}_{\mathsf{pk}}(m)$[3] and the corresponding decryption operation by $m = \mathsf{Dec}_{\mathsf{sk}}(c) = \mathsf{Dec}_{\mathsf{sk}}(\mathsf{Enc}_{\mathsf{pk}}(m))$.
- A digital signature over a message m is denoted by $\sigma = \mathsf{Sign}_{\mathsf{sk}}(m)$. The corresponding verification operation for a digital signature is denoted by $b = \mathsf{Verify}_{\mathsf{pk}}(m, \sigma)$, where $b = 1$ if the signature is valid and $b = 0$ otherwise.
- A Message Authentication Code (MAC) using a secret key K over a message m is denoted by $\mu = \mathsf{MAC}(K, m)$.

[3] Alternative notation used for clarity is $\{m\}_{\mathsf{pk}}$.

3 Taking Control Over Network Resources

Decoupling abstraction layers is a core benefit of the SDN model. It allows to combine solutions from distinct providers across the abstraction layers of a network infrastructure while maintaining encapsulation. This also applies to the interface between the application layer and the rest of the SDN deployment: on the one hand, application developers are often oblivious to packet switching details and network functionality internals. On the other hand, operators may want to withhold details of their SDN deployments from potentially malicious applications and only allow them to interact with the SDN infrastructure through a restricted policy interface.

3.1 Access Classification Scheme

Network applications require a variety of network resources to fulfill their functional requirements. In some cases, they may require temporary *exclusive* access, e.g. for atomic updates to the NIB [274]. However, they seldom – if ever – require *complete* and *indefinite* access to *all* resources. Furthermore, operators may wish to limit the access of applications to SDN resources. For example, a passive network intrusion detection function may only need "read" access to device resources, control resources and data resources, but it need not be able to modify the network state; in addition, the network operator may intend to limit the access of the intrusion-detection application to traffic from endpoints located in a certain jurisdiction. Similar to other domains where multiple parties access distributed resources (e.g. radio spectrum access [275]), we foresee multiple models of managing access to SDN resources. Based on an extensive review of existing literature on software-defined networking [156,245,254,274,276], as well as earlier classifications of access to resources in distributed systems [275,277], we next propose a taxonomy of SDN resource access models (see Figure G.4). While this scheme covers the access models and use cases identified in the reviewed literature, it can be extended with novel and emerging access models in the future.

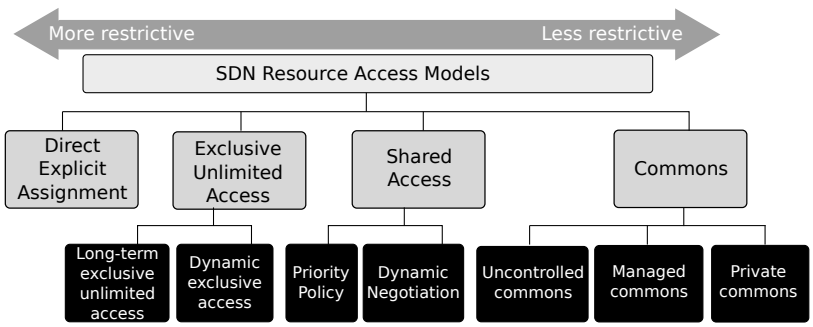


Figure G.4: Taxonomy of SDN resource access models.

Direct Explicit Assignment

In this model, the operator explicitly prescribes the SDN resources each application can access, the actions that can be performed on the resources, the duration of the granted access

and other (potentially resource-specific) attributes. This approach is suitable in cases when applications must have guaranteed access to certain resources or vice-versa, when particularly limited or security-sensitive SDN resources must be explicitly assigned to certain applications to be accessible. Direct explicit assignment might not be applicable to either *all* applications or SDN resources in the deployment. The direct assignment model is the most restrictive in the proposed taxonomy.

Exclusive Unlimited Access

In this model, the operator allows the application exclusive, unlimited access to listed SDN resources under defined constraints – e.g. geographical placement, jurisdiction, duration, etc; it encompasses two variants:

- **Long-term exclusive unlimited access:** an installed application A has unfettered access to SDN resources until its termination and cannot delegate its access permissions to a different application;
- **Dynamic exclusive access:** the installed application A has unfettered access to SDN resources. However it can delegate its permissions to other installed applications (e.g. B and C). All permissions for applications A, B, C are revoked once application A is terminated.

Shared Access

In this model, the access to SDN resources allocated to an installed application A is *shared* with one or more installed applications (e.g. B and C). This model is further detailed into the following two variants:

- **Priority policy:** in this case, if applications B and C may attempt to access the SDN resources allocated to A, their requests will *always* be denied in case of a conflicting request from A.
- **Dynamic negotiation:** application A *may* issue requests with varying priorities when prompted by other installed applications B and C. In this case, higher-priority requests from applications B and C would be accepted. Implementation of this variant may involve intricate details on dynamic access negotiation between applications.

Commons

The network operator may apply this *least* restrictive model to applications which require access to the same pool of SDN resources and have the same trust level. This model contains three variants:

- **Uncontrolled commons:** installed applications compete unrestricted for access to the allocated SDN resources. Conflict resolution mechanisms (as proposed in [172, 278]) can be used to prioritize conflicting requests. Lack of effective conflict resolution can impair the functionality of this variant.

- **Managed commons:** installed applications compete for access to the allocated SDN resources; in case of request conflicts, applications negotiate access using peer-to-peer protocols. Compared to the uncontrolled commons variant, this reduces the conflict resolution overhead, at the cost of increased communication between installed applications.
- **Private commons:** this variant includes elements of the *dynamic exclusive access* variant described above. Installed application A with exclusive unlimited access to a set of SDN resources (and delegation permissions) may allow access for other applications using one of the commons variants described above.

Once one or more SDN resource access models have been selected, the resource access control is expressed through policies, as described below.

3.2 Policies for infrastructure management

Kephart et al [279] describe three types of infrastructure management policies: *goal policies*, *utility function policies* and *action policies*. Goal policies and utility function policies are most suitable to specify enterprise business objectives and Service Level Agreements. In a datacenter scenario, a goal policy example may be *"Response time of Gold Class should be less than 100ms"*, while a utility function policy can be *"Maximize the sum of Gold and Silver Classes"*. In practice, such high-level commands are transformed into action policies, often in the form of '`IF (Condition) THEN (Action)`', e.g. '`IF (Gold_Class.Response_Time > 100 ms) THEN (increase CPU by 5%)`'. Considering the widely adopted SDN south-bound API protocols, it is clear that they are unsuitable for expressing goal policies or utility function policies; it is equally clear that formulating action policies requires intimate knowledge of the SDN deployment. Instead, functional details of SDN deployments can be encapsulated and selectively exposed to network applications through a north-bound interface (NBI).

The *north-bound interface intent framework* (Intent NBI) [268] proposed by the Open Networking Foundation and implemented in the ONOS intent framework [178], adopts this approach. Intent NBI aims to separate consumer and provider system implementations and simplify consumer-originated requests to provider systems. This is realized through non-prescriptive and composable requests, independent of network operator implementations and internal policies [268]. The non-prescriptive property allows the network controller the largest degrees of freedom in fulfilling service requests and thus facilitates conflict resolution, since intent requests do not specify which resources providers must allocate to specific services. Implementation independence is supported by a mapping mechanism, to provide the bridge between application and controller frame-of-reference terms. Figure G.5 presents a high-level view of the intent NBI approach to translating the policies – expressed in consumer application terms – to configurations – expressed in network controller terms.

The intent NBI currently lacks a mechanism to encapsulate the details of access control over network resources, in order to shield them from installed network applications. We extend the Intent NBI with a north-bound access control API (NACA) that exposes such higher-level abstractions to applications and implements them on the control plane. NACA allows to both produce policy-defined network slices and complement the intent framework with additional attributes, such as physical resource visibility, execution environment access, concurrency, etc. It is flexible enough to accommodate the various access models of the taxonomy introduced in §3.1.

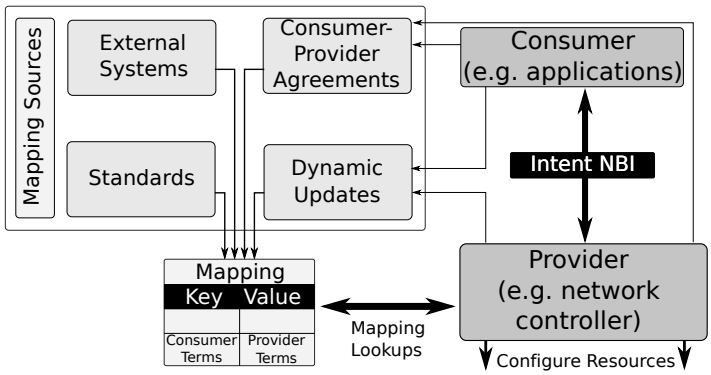


Figure G.5: Intent NBI high-level architecture illustrating the mechanism of frame-of-reference term mapping for intents.

We introduce NACA by first describing its concepts and underlying mechanisms. We next delve into the underlying implementation by describing: the north-bound access control API, which extends the Intent NBI introduced above and allows applications to declare the network resources and type of access necessary for their functionality; a mechanism to reliably tag application requests with their respective access mask; finally, a reference monitor, which ensures that illegal queries issued by applications are detected and discarded even in the event of a network controller compromise.

3.3 Scalable Access Control for SDN Resources

Beyond managing access to network resources, the centralized NIB introduced in the SDN model allows to limit access to resources using *partial views* of the system according to various dimensions, such as geographical or logical placement of resources, visibility of the underlying execution platform, etc.

This approach differs from both network *slicing* and from *network virtualization*. In [241, 276], FlowVisor defines slices along any combination of ten packet header fields, including physical, link, network, and transport layers; furthermore, such slices can be defined with negation ("all packets but TCP packets with dst port 80"), unions ("ethertype is ARP or IP dst address is 255.255.255.255"), or intersections ("netblock 192.168/16 and IP protocol is TCP"). Network virtualization decouples virtual topologies from the physical infrastructure, without exposing the mappings; instead, tenants only see their virtual networks [280]. However, neither slicing nor network virtualization can support the rich variety of access models introduced in §3.1.

We propose dividing the flowspace, on the north-bound API level, according to: (1) the resource access requests of the subjects and (2) *access masks*, defined – per subject – by the MANO component. Subjects (i.e. the installed applications) declare the resource access requests in a deployment manifest (§3.4). Access masks describe limitations to resource access on a higher abstraction level and can depend on the attributes of the network resources themselves, of the environment where they execute, or on the state of the SDN deployment (§3.4). Finally, limitations described by the access masks are enforced by NACA on the controller platform (§3.4).

This approach – first outlined in [250] – is based on a combination of earlier introduced access control approaches, such as capability-based access (CBA) [281, 282], attribute-based access control (ABAC) [283] and policy-based access control (PBAC) [284]. CBA is a subject-oriented approach, where a *capability* represents an unforgeable token used to access a resource. Subjects store their capabilities as sets of pairs $(x_i, \{R\})$, where x represents a resource and $\{R\}$ represents the set of access rights to the resource granted to the subject. PBAC allows flexible management of access rules using policies – expressed as sets of rules combined to decide authorization and determine authorization level – and can be seen as a standardization of ABAC for governance-oriented structures [285]. Restricting user access to resources using an access control API has been introduced earlier [175, 286], as well as limiting access to data based on higher-level attributes (e.g. based on geographical location [287]). However, to the best of our knowledge *there is currently no support for access control based on higher-level attributes for SDN controllers.*

We describe the framework allowing attributes to be used as input for access masks that limit access to network resources. We *do not* aim to provide an exhaustive list of resource attributes that can serve as input for access masks, since attributes are resource- and implementation-specific. The rationale behind the proposed approach is based on the portability, performance, elasticity and (multiple) security requirements formulated for VNFs [288]. We start by defining a resource attribute in the context of NACA:

Definition 3.1. A *resource attribute* is a property of the SDN resource that can be used to describe access constraints on the respective resource. The values of a collection of resource attributes can determine the type and scope of access to an SDN resource granted to a subject.

Table G.1 outlines several examples of attributes (beyond trivial ones such as e.g. instance name or identifier). Recall the example introduced by Kephart [279] and described above. The network resource attributes presented in Table G.1 can be used to create access restrictions on the goal policy level, without specifying details about either the deployment itself or the various protocols that it uses for internal communication among the components.

Table G.1: Example attributes and clarifications

Placement	Geogr./logical placement of accessible resources
Scope	Aggregate vs domain-specific access
Time	Continuous updates vs discrete updates
Jurisdiction	Jurisdictional placement of accessible resources
Physical resource visibility	Visibility of underlying execution environment
Execution environment access	Direct vs mediated access to physical resources
Resource modification types	Read state vs modify state
Concurrency	Exclusive (locking) or non-exclusive access
Authority delegation	Ability to delegate access capabilities

In a typical workflow, network operators define for each available resource R (and based on its attributes) a set of *resource access rules* (G.1).

$$R_i = \{rar_i^1, ..., \ rar_i^n\}; \ R_j = \{rar_j^1, ..., \ rar_j^n\}; ...; \ R_m = \{rar_m^1, ..., \ rar_m^n\} \tag{G.1}$$

The resource access rules contain values for the relevant attributes (e.g. as in Table G.1) of each resource.

A candidate application declares through an *application deployment manifest* the network resources it requires for its functionality. Besides a structured enumeration of the required

resources, the manifest optionally contains the types of actions to be performed. The set of requested resources must be a subset of available resources advertised by the network controller (G.2).

$$\{r_i, r_j, ..., r_m\} \in AvailableResources \tag{G.2}$$

The resource enumeration in the deployment manifest (DM) consists of a list of tuples, where each tuple contains a requested resource and a set of actions on the resource that the application requires for its functionality. This has the form $\langle resource, actions \rangle$, as shown in (G.4):

$$DM = \{\langle r_i, \{a_1, a_2, ..., a_n\}\rangle, \langle r_j, \{a_1, a_2, ..., a_n\}\rangle, ..., \langle r_m, \{a_1, a_2, ..., a_n\}\rangle\} \tag{G.3}$$

The actions listed in the deployment manifest (G.3) are resource and implementation specific: Ferguson et al. describe two types of actions – *read* and *write* [278]; Klaedtke et al. propose an expanded action set, including permissions for reading statistics (`stat`), requesting information about an object (`config_read`), modifying the state of an object (`config_mod`), as well as for subscription permissions (`subscr`) [172]. Investigation of a comprehensive taxonomy of *actions* applicable to SDN resources is out of the scope of this paper and left for future work.

The MANO component computes the access mask. First, it selects the resource access rules applicable to the requested resources and relevant to the application and builds an operator policy set (G.4):

$$OP = \{\langle rar_i^1, ..., rar_i^n\rangle, \langle rar_j^1, ..., rar_j^n\rangle, ..., \langle rar_m^1, ..., rar_m^n\rangle\} \tag{G.4}$$

Next, it applies a function f_{AM} to map each element in the operator policy set OP to at most one element in DM (G.5). Note that depending on the number and scope of the resource access rules, f_{AM} is either a surjective, bijective or injective mapping of resource access rules to requested resources.

$$f_{AM} : OP \longrightarrow DM \tag{G.5}$$

The resulting access mask AM is a three element tuple of the form $\langle resource, actions, resourceAccessRules \rangle$ (G.6). Note that there is no requirement that resource access rules describe all possible attributes of a resource.

$$AM = \{\langle r_i, \{a_1, ..., a_n\}, \{rar_i^1, ..., rar_i^n\}\rangle, ..., \langle r_m, \{a_1 ... a_n\}, \{rar_m^1, ..., rar_m^n\}\rangle\} \tag{G.6}$$

The MANO component maintains a dictionary of installed applications and their resource masks. For each new installed application, conflict detection is done by recursively checking matching resources, matching resource access rules and finally matching resource attribute values. Applications with conflicting resource requests can only be installed once the conflict is resolved. This approach to specifying access masks allows to implement any of the resource access models described in the taxonomy introduced in §3.1.

The resulting access mask is communicated to the *Request Tagger*, which – in combination with the *Reference Monitor* – ensures that the respective application cannot access resources outside the access mask, as described below.

3.4 NACA Internals

To implement support for capability-based access control policies in NACA, we have applied the Policy Core Information Model (PCIM) IETF specification [289], which describes an object-oriented information model for representing policy information. The architecture of NACA is aligned with the PCIM approach. The flow of intents issued by installed applications is illustrated in Figure G.6. The depicted components can be directly mapped to the elements of

the SDN architectural model (see Figure G.1), with the exception of *Request Tagger* (RT) and Reference Monitor (RM), both introduced as supporting mechanisms for NACA. Figure G.6 also depicts the trust relationship between the included components, according to the adversary model in §2.2: the MANO component, *Request Tagger*, *Reference Monitor* and the NIB are considered to be trusted, while the application(s) and network controller are untrusted.

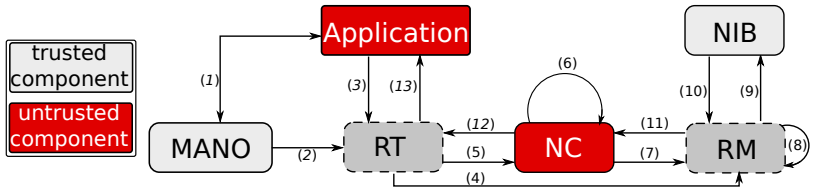


Figure G.6: NACA communication model. *Components:* MANO: management and orchestration component; RT: request tagger; NC: network controller; RM: reference monitor; NIB: network information base. *Communication steps:* (1) MANO deploys application and computes the access mask; (2) Send access masks of installed applications to RT; (3) submit application request; (4) compute the tag and an alias, communicate them to RM; (5) forward the request, along with the access mask and alias, to the NC; (6) compile request to set of queries; (7) forward compiled queries reference monitor; if verification successful (8), forward to NIB (9); return result (10-13).

In a vanilla approach, application intents are forwarded to the network controller which maps them to its internal frame of reference (recall Figure G.5), compiles them to a set of configuration instructions and applies the instructions on the network information base. We next motivate the introduction of additional components and describe their functionality.

North-bound Access Control API

Network applications vary significantly in their intended functionality. While software development best-practices emphasize data and function encapsulation [290], application developers cannot be expected to either develop applications that efficiently use limited SDN resources or have an understanding of resource partitioning within any particular SDN deployment. On the other hand, it is reasonable to expect that application developers are interested in explicitly specifying the complete set of *sufficient* resources for correct application functionality. This may include elements of all resource categories introduced in §2.1 – device, control, and data resources.

A network operator uses the deployment manifest (among other parameters) to decide whether a certain application should be installed or included in the application service catalog. Listing G.1 shows an example deployment manifest in a notation based on the standardized OASIS eXtensible Access Control Markup Language (XACML) Version 3.0 [291] (pruned for clarity and brevity).

Listing G.1: Deployment manifest fragment

```
<AnyOf>
 <AllOf>
  <Match MatchId="string-equal">
   <AttributeValue>dataplane topology</
        AttributeValue>
   <AttributeDesignator AttributeId = "resource-
        id" Category="resource"/>
  </Match>
 </AllOf>
</AnyOf>
<AnyOf>
<AllOf>
  <Match MatchId="string-equal">
    <AttributeValue>read</AttributeValue>
    <AttributeDesignator AttributeId = "action-id
        " Category="action"/>
  </Match>
</AllOf>
<AllOf>
  <Match MatchId="string-equal">
    <AttributeValue>modify</AttributeValue>
    <AttributeDesignator AttributeId = "action-id"
        Category="action"/>
  </Match>
</AllOf>
</AnyOf>
```

Listing G.2: Resource mask fragment based on deployment manifest

```
<AnyOf>
 <AllOf>
  <Match MatchId="string-equal">
  <AttributeValue> dataplane topology </AttributeValue>
    <AttributeDesignator AttributeId="resource-id"
        Category="resource"></AttributeDesignator>
  </Match>
  <Match MatchId="string-equal">
  <AttributeValue DataType="string">region-A</
        AttributeValue>
    <AttributeDesignator AttributeId="jurisdiction"
        Category="resource"></AttributeDesignator>
  </Match>
  <Match MatchId="string-equal">
  <AttributeValue DataType="string">dataplane topology</
        AttributeValue>
    <AttributeDesignator AttributeId="resource-id"
        Category="resource"></AttributeDesignator>
  </Match>
 </AllOf>
</AnyOf>
<AnyOf>
 <AllOf>
  <Match MatchId="string-equal">
   <AttributeValue DataType="string">read</AttributeValue>
        <AttributeDesignator AttributeId="action-id
            " Category="action">
   </AttributeDesignator>
  </Match>
 </AllOf>
</AnyOf>
```

Once installed, an application containing in its deployment manifest the fragment from Listing G.1 can read and modify the topology on the data plane. While the application requires read permissions to the network topology, revealing the *entire* topology might be unacceptable to the network operator. Instead, the operator may consider allowing the application to access only a *restricted subset* of the topology and present the rest as a black box. Therefore, the operator uses the MANO component to apply a simple resource access rule to the resource "`dataplane-topology`". The rule comprises only one attribute – jurisdiction – with the value "`region-A`". Once applied, the resource access rule reduces the visibility of the topology for the application exclusively to the selected region, while the rest of the topology is not reported.

To represent access masks, we extend the OASIS XACML notation [291] (see Listing G.2). The resource mask example in Listing G.2 allows an application to access the topology of the SDN deployment; however, it is only limited to the resources located in region "A". The MANO component communicates the access masks for each installed application to the *Request Tagger*.

Request tagging

Application requests must be *authenticated* and *tagged* with an access mask prior to reaching the network controller. While such functionality can be implemented by the network controller, we have chosen a modular approach for the following reasons: first, authentication and tagging are generic functions that can be implemented independently from a deployment-specific network controller; second, this approach allows to minimize the network controller modifications, required to implement NACA; finally, the network controller is a high-value, high-risk target which may contain API vulnerabilities which can be exploited by untrusted applications; moreover, query parsing is a common attack vector [292] which could be used to corrupt the network controller. The reasons above motivate the introduction of a *Request Tag-*

ger – a pre-processing component implementing access control on the north-bound interface, resilient to a potentially compromised network controller (Figure G.6).

At deployment time, the MANO component communicates to the *Request Tagger* a set of tuples describing the installed applications and their access mask; the set of tuples is updated for every new installed application. We consider the *Request Tagger* a trusted component (see discussion in §4.2) and assume the integrity of such messages can be ensured and reliably verified. Application requests on the north-bound interface must contain a reliably verified application identifier, the set of requested resources (i.e. elements included in the application deployment manifest), and an intent that can be compiled into one or more implementable queries to the NIB. An incoming request is pre-processed by the *Request Tagger*, which:

1. verifies the authenticity of the request;
2. matches the application instance – identified by ($\mathsf{App_i}$) – with one of the access masks ($\mathsf{AM_i}$) earlier communicated by the MANO component;
3. assigns a unique identifier to the request ($\mathsf{Req_i^j}$);
4. tags the request with the identified access mask.

For step 4, the *Request Tagger* constructs a tag (G.7) which is a MAC over the following elements: identifier of the requesting application ($\mathsf{App_i}$); unique identity of the request ($\mathsf{Req_i^j}$); application access mask $\mathsf{AM_i}$; monotonically increasing sequence counter n. The MAC value is computed using a shared key K distributed by the MANO component at deployment time to the *Request Tagger* and *Reference Monitor* (see Figure G.7).

$$= \mathsf{MAC}(\mathsf{K}, (\mathsf{App_i}, \mathsf{Req_i^j}, \mathsf{AM_i}, \mathsf{n})) \tag{G.7}$$

Next, the application request is forwarded to the network controller without further processing, along with $\mathsf{AM_i}$ – the application access mask. The network controller processes the request, applies operator-defined policies and compiles a request $\mathsf{Req_i^j}$ into a set of discrete queries $\{Q_i^{j1}..Q_i^{jn}\}$. Note that the compiled queries contain the identifier of the requesting application ($\mathsf{App_i}$) and the unique request identifier ($\mathsf{Req_i^j}$). Internals of network controller processing are out of the scope of this description and can be found in [100, 178].

Access Reference Monitor

The *queries* produced by the network controller describe discrete changes made to e.g. the configuration of network elements or flow tables on the forwarding plane. However, a network controller containing API vulnerabilities, implementation bugs or maliciously modified through an attack [293] may produce queries that invoke resources outside of the application's access mask. To address this, we introduce a *Reference Monitor* as a discrete component (similar to the approach described in §3.4). It validates the network controller output (i.e. queries ready for execution on the NIB) and ensures that illegal application queries fail to reach the NIB.

Definition 3.2. *Illegal application queries* are those that on behalf of an application invoke resources outside of its access mask.

The workflow of the *Reference Monitor* is as follows. At deployment time, the MANO component pre-seeds the *Reference Monitor* with the key material (e.g. public key infrastructure

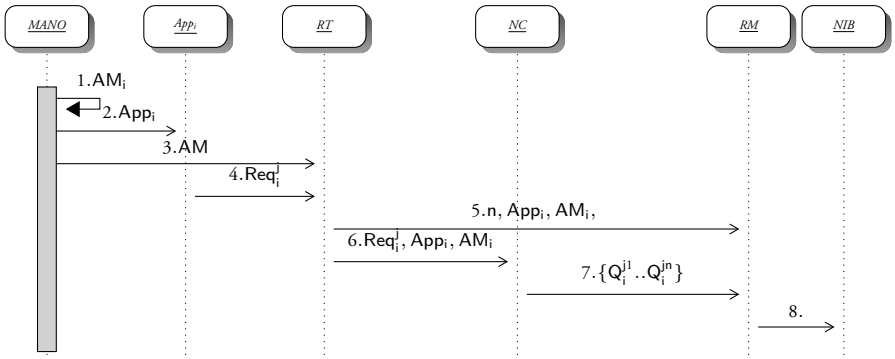


Figure G.7: Message flow in NACA (we assume communication over a secure channel). *Components*: MANO: management and orchestration component; RT: request tagger; NC: network controller; RM: reference monitor; NIB: network information base. *Flow*: (1) MANO computes access mask and (2) deploys the app; (3) MANO distributes to *Request Tagger* the access mask for $\mathsf{App_i}$; once the application $\mathsf{App_i}$ issues a request (4), RT computes a tag and communicates it to the RM (5), prior to forwarding the request to the NC where the request is processed and compiled into a set of queries (6), forwarded to the RM (7). RM verifies the queries, computes a MAC over each query with a nonce – producing $\{\{\mathsf{Q_i^{j1}}, \mathsf{u^1}, \mathsf{_i^{j1}}\}..\{\mathsf{Q_i^{jn}}, \mathsf{u^n}, \mathsf{_i^{jn}}\}\}$ and forwards them to the NIB (8).

certificate) required to establish an authenticated, confidentiality and integrity-protected communication channel, as well as a shared key K used for authenticating incoming request tags. This communication channel is maintained throughout the component lifetime. In-transit component protection can be ensured using mechanisms described in [265].

Request matching For each application request, the *Request Tagger* communicates to the *Reference Monitor*, over a secure channel, the tag μ computed according to (G.7), along with the application identifier ($\mathsf{App_i}$), its current access mask ($\mathsf{AM_i}$) and a request counter n. Similarly, the network controller forwards to the *Reference Monitor* for verification all queries, produced from application requests according to the respective access mask.

Upon receiving a set of queries from the network controller, the *Reference Monitor* first verifies the freshness of the monotonically increasing counter n (it is expected that a correctly functioning network controller compiles requests in first-in-first-out order). It next computes a tag μ' according to operation (G.7) and compares the result with the received tag μ; execution only continues if $\mu' = \mu$. Finally, it parses the queries and verifies them against the access mask of the application. This is done by checking that:

- queries exclusively invoke resources enumerated in the access mask;
- resources are invoked according to the actions specified in the access mask.

Query invalidation While duration of request compilation into queries can vary depending on the complexity of the requests, a malicious network controller may delay or reorder output of queries. To prevent the reuse of a more permissive access mask for lower-privileged applications,

we use a *sliding window* for invalidating the computed queries. The approach is as follows: a delayed request $\mathsf{Req}_\mathsf{a}^1$ issued by application App_a can be followed by a limited number of requests $\mathsf{Req}_\mathsf{a}^2 ... \mathsf{Req}_\mathsf{a}^n$ issued by the same application, while a delay of $\mathsf{Req}_\mathsf{a}^{n+1}$ requests from the same application invalidates the entire batch (the limit is configuration-specific); delay of a request $\mathsf{Req}_\mathsf{a}^1$ issued by App_a followed by a request $\mathsf{Req}_\mathsf{b}^1$ issued by App_b, as well as reordering of the requests issued by App_a and App_b, invalidates both requests.

The verification by the *Reference Monitor* adds an essential element of the *direct explicit assignment* resource access model introduced in §3.1: access is only permitted to SDN resources that have been explicitly described and over actions that have been explicitly listed. We exemplify this below.

Recall the access mask fragment in Listing G.2. Example 3.1 shows a query to install a new flow between source IP w, port w1 and destination IP x, port x1 over the user datagram protocol. The *Reference Monitor* would trivially reject this query since it invokes a resource (flow) that is *not* enumerated in the deployment manifest (and hence would *not* be present in the access mask).

Example 3.1. *New flow installation query*
$\langle \mathsf{flow},\ \mathsf{allow},\ \mathsf{srcIP} = \mathsf{w}, \mathsf{dstIP} = \mathsf{x}\ \mathsf{proto} = \mathsf{UDP}, \mathsf{srcPort} = \mathsf{w1}, \mathsf{dstPort} = \mathsf{x1} \rangle$

Depending on the size and complexity of the query set, verifying queries may require multiple interactions between the *Reference Monitor* and NIB to learn the attributes of the invoked resources. In Example 3.2 the application requests a list of logical termination points that bound a node 'j':

Example 3.2. *Application request*
$\langle \mathsf{topo},\ \mathsf{node_j},\ \mathsf{ltpRefList} \rangle$

To verify this query, the *Reference Monitor* must first obtain the topology reference of the respective node ($\mathsf{node_j}$), identify its jurisdiction and compare it with the jurisdiction in the access mask (`region-A` in Listing G.2). If $\mathsf{node_j}$ is located in `region-A`, a list of logical termination points (also located in the same jurisdiction) is returned to the caller. Otherwise the request is denied.

In case of an access mask mismatch – which may indicate an intent compiler bug or a compromise of the network controller – the *Reference Monitor* drops the illegal query and notifies the MANO component to take an appropriate mitigation action. While the access mask verification prevents illegal queries, a compromised network controller may attempt other attacks as well, such as attempt to circumvent the *Reference Monitor* by compiling a request into a set of unintended queries (which nevertheless comply with the access mask). Considering that a vulnerability in the network controller must be exploited (or triggered) through a request from a malicious application, such an attack is prevented by the sliding window invalidating delayed or reordered queries. This prevents the adversary from inserting malicious queries into the set of queries compiled from the request of a benign application. The ordering, origin and privileges of submitted requests – communicated from the *Request Tagger* to the *Request Manager* – provide the information necessary for invalidating queries in case of suspected network controller compromise. Note however that these steps help prevent exploitation of a compromise and do not aim to detect all occurrences of compilation errors. Likewise, the steps above do not prevent malicious actions by code built into the intent compiler. This problem (first formulated in [294]) is out of the current scope.

For valid requests, the *Reference Monitor* computes a MAC over each query and a nonce 'u' using the key $\mathsf{K_{NIB}}$, shared between *Reference Monitor* and NIB (G.8).

$$\{\mu_i^{j1}..\mu_i^{jn}\} = \mathsf{MAC}(\mathsf{K_{NIB}}, (\mathsf{Q_i^{j1}}, \mathsf{u^1}))..\mathsf{MAC}(\mathsf{K_{NIB}}, (\mathsf{Q_i^{jn}}, \mathsf{u^n})) \tag{G.8}$$

Finally, the *Reference Monitor* forwards the queries, nonces and computed MACs to the NIB, which first recomputes the MACs and only processes queries with verified integrity and a fresh nonce.

4 Implementation

We have implemented NACA as an extension to ONOS, a popular open-source SDN controller [100]. ONOS includes an *intent* framework, allowing applications to specify their *network control desires* as policies rather than specific mechanisms. This raises the abstraction level for applications from specifying details about *how* the SDN infrastructure configuration should be updated to achieve certain functional changes (e.g. by updating network flows, instantiating new applications, etc.) to conveying through high-level intents *what* functionality should be enabled. This shift is equivalent to replacing the need to specify network mechanisms acting through OpenFlow operations with tools that are more durable and robust in the face of topology changes. Once created by an application, *intents* become immutable objects communicated to the ONOS core which – when installed – alter the network state.

4.1 Extending the ONOS Intent Framework

In ONOS, intent objects contain the identifier of the issuing application (`ApplicationId`) and are uniquely identifiable through the intent identifier (`IntentId`), generated when the query object is created in the intent framework based on an application request. An ONOS intent is additionally described by the following elements: the required device resources (a collection of `networkResource` objects, similar to the the NACA queries described above); the intent `priority` and `constraints` prescribed by the application (criteria enumerating e.g. packet header fields or patterns describing slices of traffic); instructions describing actions to be applied to a slice of traffic. To implement NACA, we extend the intent framework with two components – `RequestTagger` and `ReferenceMonitor`. Figure G.8 depicts the state transition diagram for compilation of application intents, along with the NACA extensions.

The `RequestTagger` pre-processes intents submitted by applications. It first reliably determines the identity of the issuing application by checking the intent signature and certificate, and assigns a corresponding `ApplicationId`. It next identifies the access mask matching the `ApplicationId`, computes the tag T (see G.9), communicates it to the `Reference Monitor` and transitions to the `InstallRequest` state (transition ❷ in Figure G.8).

$$\mathsf{T} = \mathsf{MAC}(\mathsf{K}, (\mathsf{ApplicationId},\ \mathsf{IntentIdnetworkResource},\ \mathsf{priority},\ \mathsf{AM},\ \mathsf{n})) \tag{G.9}$$

In ONOS, we implemented the `RequestTagger` as a state of the intent framework, invoked directly from the `IntentManager` class. The intent framework compiles requests issued by the applications into queries according to constraints specified in the access mask. Prior to executing the compiled query on the NIB and transitioning to the `Installed` state, the execution flow

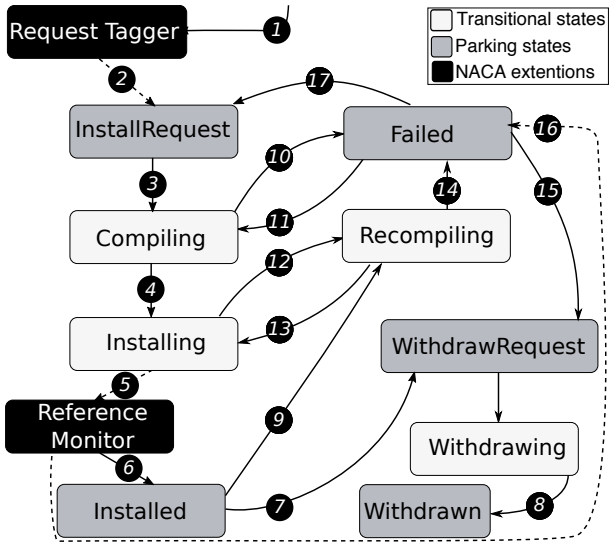


Figure G.8: NACA Extended intent framework state diagram. *Transitions:* (1) application request; (2) tagged query request; (3) submit for compilation; (4) compile succeeded; (5) install succeeded; (6) verify access compliance; (7) withdrawal installed request; (8) remove topo or flow event; (9) add/update topo event; (10) compile failed; (11) retry compile; (12) install failed; (13) retry install; (14) compile failed or same result; (15) withdrawal of failed requests; (16) reference monitor rejected request; (17) retry install intent.

transitions to the `ReferenceMonitor` (transition ❺ in Figure G.8). The `ReferenceMonitor` first verifies tag T by recomputing the MAC:

$$\mathsf{MAC}(\mathsf{K}, (\mathsf{ApplicationId},\ \mathsf{IntentIdnetworkResource},\ \mathsf{priority},\ \mathsf{AM},\ \mathsf{n})) \tag{G.10}$$

It next verifies that the queries fall into the validation sliding window, by checking that all earlier expected queries have been output (sliding window for delayed requests was configured to `n=2`) and that queries have the expected source application and access mask. Finally, it checks that the compiled queries do not violate the access mask assigned by the MANO component, as described in §3.4: the `ReferenceMonitor` parses the query to check if all invoked resources are included in the access mask, if queries follow the limits of the resource attributes, as well as whether the queries invoke only actions allowed by the access mask. A MAC is computed over valid queries and submitted – along with the requests – to the NIB.

Similar to the `RequestTagger` above, the `ReferenceMonitor` is implemented as a state of the ONOS intent framework. Every intent must transition to the `ReferenceMonitor` state before the resulting compiled queries can be applied. Queries submitted directly to the NIB are ignored, as they lack a valid MAC that must be computed by the `ReferenceMonitor`.

The `Constraint` implementation in ONOS allows applications to formulate filters which are applied on the submitted intents. We leverage this implementation to apply the access masks assigned by the MANO component – the `ReferenceMonitor` ensures that the candidate intents comply with the access mask constraints. If an access mask violation is detected, the `ReferenceMonitor` transitions to the `Failed` state (transition ⓰ in Figure G.8).

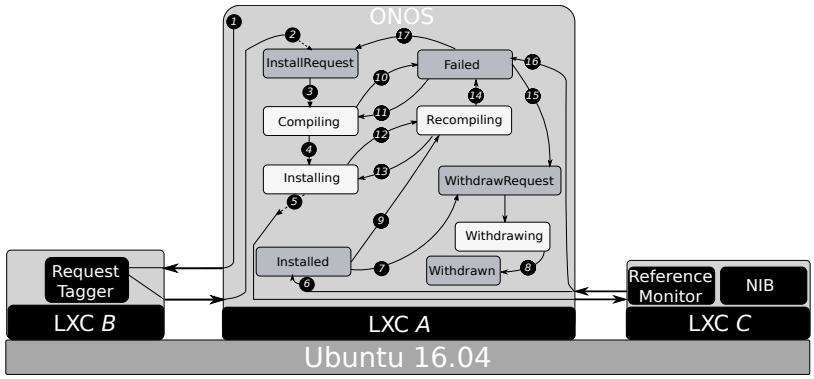


Figure G.9: NACA testbed

4.2 Component Isolation

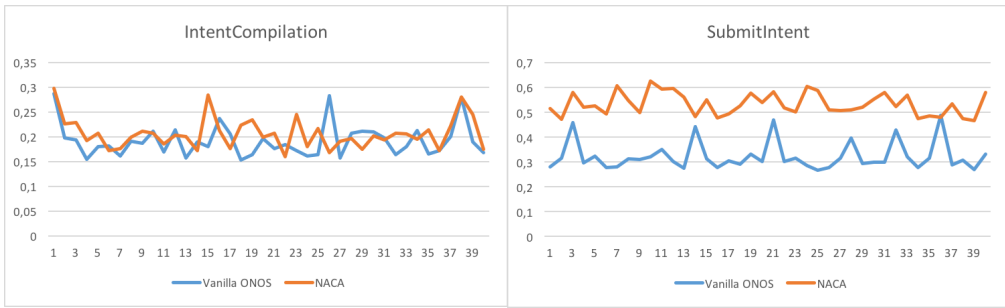
Request Tagger and *Reference Monitor* are of central importance in verifying access of untrusted applications to limited, potentially confidentiality and integrity sensitive resources. To protect them from a malicious network controller, the *Request Tagger* and *Reference Monitor* are executed in a trusted execution environment *isolated* from a potentially malicious underlying operating system [63,116,265]. We adopt the definition of trust from [14], namely "confidence in the integrity of an entity for reliance on that entity to fulfill specific responsibilities". A trusted execution environment can be created using operating system level virtualization [26], platform virtualization [108], or using hardware-assisted isolated execution environments [134] – such as ARM TrustZone [79], Intel SGX enclaves [51] or AMD secure memory encryption [77]. The exact implementation approach depends of multiple factors, such as adversary model, deployment context, acceptable performance penalty or hardware capabilities. In the current implementation, as further described in Section 5, we have chosen operating system level virtualization to separate the trusted components – such as `RequestTagger`, `ReferenceMonitor` and the NIB – from the vulnerable network controller. This approach allows to isolate the process and address spaces of the components, while avoiding excessive overhead.

5 Evaluation

NACA was evaluated in a virtualized testbed, as illustrated in Figure G.9.

In order to limit the influence of system configuration options on the results of the evaluation, we have chosen a lightweight approach to isolation between the components and used Linux Containers [295] to isolate the address and memory spaces of the NACA components. Thus, the modified ONOS controller, `RequestTagger` and `ReferenceMonitor` were deployed in three separate containers (LXC A, LXC B, and LXC C respectively). Note that this deployment choice is not binding and allows for alternative isolation approaches to be used. The testbed containers were deployed on a Ubuntu 16.04 VirtualBox[4] virtual machine, with 1 CPU and 8 GB memory, default paravirtualization interface.

[4]Oracle VirtualBox `https://www.virtualbox.org/wiki/VirtualBox`



(a) `compileIntent`, time in s.

(b) `submitIntent`, time in s.

Figure G.10: NACA performance evaluation.

We evaluated NACA using the intents and test case coverage available in ONOS. Figure G.10a illustrates the performance of intent compilation over 40 test runs. Here the NACA extensions induce only a minor performance impact, namely 9% increase on the median intent compilation time (8% mean increase). Figure G.10b illustrates the performance of intent installation over 40 test runs. Intent installation includes the entire flow from request submission by the application to the queries accepted by the NIB. In this case, the induced performance overhead reflects implementation and deployment decisions (including choice of isolation mechanism and target deployment platform).

6 Related Work

From the origins of SDN, access control and network programambility have received significant attention.

Access control for SDN controllers Casado et al. describe in Ethane [174] an enterprise network architecture allowing network managers to control deployments using system-wide fine-grained policies. Ethane includes two component types: *(1)* 'dumb' switches maintaining flow table entries and communicating with the controller, and *(2)* one or more controllers handling host registration and authentication, tracking network bindings, implementing access control, and enforcing resource limits on the managed flows. The proposed high-level language for Ethane network management policies – despite its shortcomings such as lack of support for dynamic policy updates and assumption of a fixed network topology – has inspired a rich collection of subsequent network policy languages [296–299]. While such network control languages fueled a rapid development of network controller capabilities, they operate on a lower abstraction level than required for network management applications. Several outstanding issues of the Ethane model are broadcast and service discovery, application-layer routing, knowledge about application-layer configuration and potential damage from spoofing Ethernet addresses.

Ferguson et al. [175] rely on *hierarchical* composition of policies to define access to actions performed on traffic flows, as well as to control resource allocation. Furthermore, the framework contains a policy conflict resolution mechanism based on user-defined operators. Conflict

resolution is essential to enable a distributed approach to policy definition, where distinct applications requiring device resources and services define access policies and Quality-of-Service requirements. While this approach brings control over network access to applications using network services, it does not hide the details of device resources and requires from the application knowledge of the network internals. NACA maintains the higher-level control of network access, while abstracting the network internals from the application programmers.

The access control scheme for SDN controllers proposed in [172] accounts for device resources, multiple security requirements, conflicts originating from reconfiguration of network components, and delegation of access permissions. This scheme mimics access control schemes for operating systems, with contextual adjustments. In this case network users are the subjects and the network components are the objects. A follow-up prototype is described in [173].

APIs for Network Configuration The *Pane* controller [278] allows applications to control SDN deployments. It delegates read-write privileges on network configuration to end-users or applications acting on their behalf. The *Pane* API allows three messages types – requests, queries and hints. Requests affect the state of the network following the intention of the application. Applications can issue queries about the state of the network and provide hints to notify the controller about upcoming changes, e.g. in the network resource usage. In this model, principals are included in one or more shares and have complete access control privileges – including recursive delegation of privileges – over the set of network flows in the respective shares. Intra-share resource over-subscription is allowed and is monitored by the controller, which also detects and resolves potential conflicts. NACA takes a different approach by restricting applications to network control privileges explicitly declared, verified and validated at deployment time. Furthermore, it verifies that control messages issued by network applications comply with the allowed control privileges.

A low-level API for dynamic configuration of Quality-of-Service resources in network devices can be implemented through a plugin to the OVSDB protocol [300]. Such low-level configuration allows granular control across three levels: port, switch and network-wide controls. While this allows extensive control over the forwarding plane configuration, the proposed approach does not discuss mechanisms to prevent applications using such an API from monopolizing network control, nor any resolution mechanism in case of conflicting policies.

A Network Overlay Framework [301] addresses the challenge of configuring network deployments according to application requirements. Through its API and programming language, the Network Overlay Framework allows developers to program the network according to the needs of the application. This replaces the "best-effort" packet delivery approach with explicit Quality-of-Service guarantees tailored to a specific application. While the approach has been implemented for the Hadoop data processing framework, it does not consider any multi-application or multi-tennant scenarios. Likewise, it does not describe any resolution mechanism that would be needed in the case of configuration conflicts. Furthermore, this approach allows the forwarding plane to manipulate the configuration of the control plane, disregarding the possibility of malicious or misconfigured applications. This in turn creates network security and safety concerns.

SDN controller functionality can be implemented by extending an existing operating system and leveraging its software ecosystem, i.e. operating system utilities and a distributed file system, as proposed in *yanc* [269]. Here, file I/O is used as a single API for SDN applications, allowing to avoid the restriction to use programming languages mandated by the implementation

of the SDN controller. The authors indicate the possibility of using permissions implemented by the virtual file system layer for fine-grained access control of network resources. *Yanc* could leverage NACA by storing *resource masks* in the extended file attributes and further integrating the NACA API with existing access control policies.

Eden is a framework for enabling end-host network functions [302], assuming a single-domain network deployment, where at least a subset of end-hosts can be trusted, e.g. in datacenters. The design builds on the observation that a large class of network functions feature three key requirements: data-plane computation, data-plane state, and operation on application semantics. It includes a flexible scheme for application-level classification of network traffic based on a custom language as well as a compiler and run-time for action functions – constructs able to access and modify packet classes and endpoint enclaves. In Eden, network applications are "first-class principals", and can classify packets based on application-internal semantics. The classification follows the packet through the end host stack and is used to determine the rules to apply. A similar approach has been used in NACA for access control. However, Eden assumes a single-domain, controlled and trusted environment, and does not feature policy conflict resolution or access control mechanisms.

An early concept of intent-based network abstractions has been introduced in [303], allowing to specify networks as a policy governed service. Such intents describe the functionality of the network – i.e. *what* connectivity a certain application requires, connectivity between endpoints and policies associated with the connectivity – while leaving out the specifics on *how* to implement the required connectivity. This approach is further developed in projects such as "Boulder North-bound Interface (NBI)" [268]. NACA reuses the NBI implementation in the ONOS intent framework and extends the intent concept described in [303] with an API to specify access control constraints on the intents.

Maple [304] enables developers to use programming languages such as Haskell and Java to define network behaviors through centralized algorithms (algorithmic policies). The use of algorithmic policies hides the challenges of implementing high-level policies into sets of rules on distributed individual switches.

In large distributed systems such as SDN deployments configuration updates may lead to undefined network behavior and security vulnerabilities, if applied incorrectly due to long latency, dropped packets or weak consistency. To address this, *event-driven consistent updates* [299] preserve well-defined behaviors when transitioning between configurations in response to events. The approach places strong *locality* requirements towards configuration updates: it allows exclusively configuration changes decidable with *local* (rather than remote) information in a distributed system. NACA allows to control the locality access of network applications to SDN resources (recall *Extent* and *Placement* resource masks in Table G.1), and can be extended to include other access types.

7 Future Work

We aim to extend and improve several aspects of this contribution in future work. One important direction is integration with other controller platforms, in particular *yanc* [269] in order to adapt the NACA intent approach to existing operating system access control mechanisms. Extending NACA with *hints* as in [175] can improve the overall performance of the deployment. Finally, implementing and evaluating the approach using alternative isolation

solutions discussed in §4.2 can yield new insights into trade-offs between security guarantees and performance for network controllers.

8 Conclusion

Design and implementation of a controller-agnostic north-bound interface is the current focus of SDN development. Such an interface will allow network operators to deploy multiple applications on their SDN infrastructure and unlock rich network management functionality. However, the currently available north-bound interfaces do not offer access control mechanisms to allow applications to negotiate network resource access or operators to obtain a comprehensive overview of the resource access of the installed applications. We have addressed this by first introducing a taxonomy of resource access models for SDN infrastructure, along with a network access control API which allow applications to commit at deployment time to a set of resource access requirements, which are then enforced by discrete components on the network controller platform. We described the design, implementation and performance evaluation of the proposed solution.

SICS Dissertation Series

1. Bogumil Hausman, Pruning and Speculative Work in OR-Parallel PROLOG, 1990.
2. Mats Carlsson, Design and Implementation of an OR-Parallel Prolog Engine, 1990.
3. Nabiel A. Elshiewy, Robust Coordinated Reactive Computing in SANDRA, 1990.
4. Dan Sahlin, An Automatic Partial Evaluator for Full Prolog, 1991.
5. Hans A. Hansson, Time and Probability in Formal Design of Distributed Systems, 1991.
6. Peter Sjödin, From LOTOS Specifications to Distributed Implementations, 1991.
7. Roland Karlsson, A High Performance OR-parallel Prolog System, 1992.
8. Erik Hagersten, Toward Scalable Cache Only Memory Architectures, 1992.
9. Lars-Henrik Eriksson, Finitary Partial Inductive Definitions and General Logic, 1993.
10. Mats Björkman, Architectures for High Performance Communication, 1993.
11. Stephen Pink, Measurement, Implementation, and Optimization of Internet Protocols, 1993.
12. Martin Aronsson, GCLA. The Design, Use, and Implementation of a Program Development System, 1993.
13. Christer Samuelsson, Fast Natural-Language Parsing Using Explanation-Based Learning, 1994.
14. Sverker Jansson, AKL – A Multiparadigm Programming Language, 1994.
15. Fredrik Orava, On the Formal Analysis of Telecommunication Protocols, 1994.
16. Torbjörn Keisu, Tree Constraints, 1994.
17. Olof Hagsand, Computer and Communication Support for Interactive Distributed Applications, 1995.
18. Björn Carlsson, Compiling and Executing Finite Domain Constraints, 1995.
19. Per Kreuger, Computational Issues in Calculi of Partial Inductive Definitions, 1995.
20. Annika Waern, Recognising Human Plans: Issues for Plan Recognition in Human-Computer Interaction, 1996.
21. Björn Gambäck, Processing Swedish Sentences: A Unification-Based Grammar and Some Applications, 1997.
22. Klas Orsvärn, Knowledge Modelling with Libraries of Task Decomposition Methods, 1996.
23. Kia Höök, A Glass Box Approach to Adaptive Hypermedia, 1996.
24. Bengt Ahlgren, Improving Computer Communication Performance by Reducing Memory Bandwidth Consumption, 1997.
25. Johan Montelius, Exploiting Fine-grain Parallelism in Concurrent Constraint Languages, 1997.
26. Jussi Karlgren, Stylistic experiments in information retrieval, 2000.
27. Ashley Saulsbury, Attacking Latency Bottlenecks in Distributed Shared Memory Systems, 1999.

28. Kristian Simsarian, Toward Human Robot Collaboration, 2000.
29. Lars-Åke Fredlund, A Framework for Reasoning about Erlang Code, 2001.
30. Thiemo Voigt, Architectures for Service Differentiation in Overloaded Internet Servers, 2002.
31. Fredrik Espinoza, Individual Service Provisioning, 2003.
32. Lars Rasmusson, Network capacity sharing with QoS as a financial derivative pricing problem: algorithms and network design, 2002.
33. Martin Svensson, Defining, Designing and Evaluating Social Navigation, 2003.
34. Joe Armstrong, Making reliable distributed systems in the presence of software errors, 2003.
35. Emmanuel Frécon, DIVE on the Internet, 2004.
36. Rickard Cöster, Algorithms and Representations for Personalised Information Access, 2005.
37. Per Brand, The Design Philosophy of Distributed Programming Systems: the Mozart Experience, 2005.
38. Sameh El-Ansary, Designs and Analyses in Structured Peer-to-Peer Systems, 2005.
39. Erik Klintskog, Generic Distribution Support for Programming Systems, 2005.
40. Markus Bylund, A Design Rationale for Pervasive Computing – User Experience, Contextual Change, and Technical Requirements, 2005.
41. Åsa Rudström, Co-Construction of hybrid spaces, 2005.
42. Babak Sadighi Firozabadi, Decentralised Privilege Management for Access Control, 2005.
43. Marie Sjölinder, Age-related Cognitive Decline and Navigation in Electronic Environments, 2006.
44. Magnus Sahlgren, The Word-Space Model: Using Distributional Analysis to Represent Syntagmatic and Paradigmatic Relations between Words in High-dimensional Vector Spaces, 2006.
45. Ali Ghodsi, Distributed k-ary System: Algorithms for Distributed Hash Tables, 2006.
46. Stina Nylander, Design and Implementation of Multi-Device Services, 2007
47. Adam Dunkels, Programming Memory-Constrained Networked Embedded Systems, 2007
48. Jarmo Laaksolahti, Plot, Spectacle, and Experience: Contributions to the Design and Evaluation of Interactive Storytelling, 2008
49. Daniel Gillblad, On Practical Machine Learning and Data Analysis, 2008
50. Fredrik Olsson, Bootstrapping Named Entity Annotation by Means of Active Machine Learning: a Method for Creating Corpora, 2008
51. Ian Marsh, Quality Aspects of Internet Telephony, 2009
52. Markus Bohlin, A Study of Combinatorial Optimization Problems in Industrial Computer Systems, 2009
53. Petra Sundström, Designing Affective Loop Experiences, 2010
54. Anders Gunnar, Aspects of Proactive Traffic Engineering in IP Networks, 2011
55. Preben Hansen, Task-based Information Seeking and Retrieval in the Patent Domain: Process and Relationships, 2011
56. Fredrik Österlind, Improving Low-Power Wireless Protocols with Timing-Accurate Simulation, 2011
57. Ahmad Al-Shishtawy, Self-Management for Large-Scale Distributed Systems, 2012
58. Henrik Abrahamsson, Network overload avoidance by traffic engineering and content caching, 2012
59. Mattias Rost, Mobility is the Message: Experiment with Mobile Media Sharing, 2013
60. Amir H. Payberah, Live Streaming in P2P and Hybrid P2P-Cloud Environments for the open Internet, 2013
61. Oscar Täckström, Predicting Linguistic Structure with Incomplete and Cross-Lingual Supervision, 2013

62. Cosmin Arad, Programming Model and Protocols for Reconfigurable Distributed Systems, 2013
63. Tallat M. Shafaat, Partition Tolerance and Data Consistency in Structured Overlay Networks, 2013
64. Shahid Raza, Lightweight Security Solutions for the Internet of Things, 2013
65. Mattias Jacobsson, Tinkering with Interactive Materials: Studies, Concepts and Prototypes, 2013
66. Baki Cakici, The Informed Gaze: On the Implications of ICT-Based Surveillance, 2013
67. John Ardelius, On the Performance Analysis of Large Scale, Dynamic, Distributed and Parallel Systems, 2013
68. Fatemeh Rahimian, Gossip-Based Algorithms for Information Dissemination and Graph Clustering, 2014
69. Rebecca Steinert, Probabilistic Fault Management in Networked Systems, 2014
70. Mudassar Alsam, Bringing Visibility in the Clouds: Using Security, Transparency and Assurance Services, 2014
71. Anna Ståhl, Designing for Interactional Empowerment, 2015
72. Pedro Sanches, Health Data: Representation and (In)visibility, 2015
73. Tomas Olsson, A Data-Driven Approach to Remote Fault Diagnosis of Heavy-duty Machines, 2015
74. Nicolas Tsiftes, Storage-Centric System Architectures for Networked, Resource-Constrained Devices, 2016
75. Oliver Schwarz, No Hypervisor Is an Island: System-wide Isolation Guarantees for Low Level Code, 2016
76. Elsa Kosmack Vaara, Exploring the Aesthetics of Felt Time, 2017
77. Nicolae Paladi, Trust But Verify: Trust Establishment Mechanisms in Infrastructure Clouds, 2017

References

[1] G. Greenwald, E. MacAskill, and L. Poitras, "Edward Snowden: the whistleblower behind the NSA surveillance revelations," *The Guardian*, vol. 9, June 2013.

[2] G. Greenwald, "How the NSA tampers with US-made Internet routers," *The Guardian*, vol. 10, May 2014.

[3] P. Heim, "Resetting passwords to keep your files safe," August 2016. [Online; July 2017].

[4] C. A. Ardagna, R. Asal, E. Damiani, and Q. H. Vu, "From security to assurance in the cloud: A survey," *ACM Comput. Surv.*, vol. 48, pp. 2:1–2:50, July 2015.

[5] Y. Nugraha and A. Martin, "Towards the Classification of Confidentiality Capabilities in Trustworthy Service Level Agreements," in *Proc. 2017 IEEE International Conference on Cloud Engineering*, IC2E'17, IEEE, April 2017.

[6] L. J. Bell, D. E. ; LaPadula, "Secure Computer Systems: Mathematical Foundations," Tech. Rep. AD0770768, MITRE CORP BEDFORD MA, November 1973.

[7] K. J. Biba, "Integrity Considerations for Secure Computer Systems," Tech. Rep. ADA039324, MITRE CORP BEDFORD MA, April 1977.

[8] R. Anderson, *Security Engineering*. New Jersey, USA: John Wiley & Sons, 2nd ed. ed., April 2010.

[9] M. Dekker, C. Karsberg, M. Lakka, and D. Liveri, "Auditing Security Measures, An Overview of schemes for auditing security measures," Tech. Rep. TP-03-13-551-EN-N, European Union Agency for Network and Information Security, September 2013.

[10] C. Di Giulio, C. Kamhoua, R. H. Campbell, R. Sprabery, K. Kwiat, and M. N. Bashir, "IT Security and Privacy Standards in Comparison: Improving FedRAMP Authorization for Cloud Service Providers," in *Proc. of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '17, pp. 1090–1099, IEEE, 2017.

[11] L. Taylor, "FedRAMP: History and Future Direction," *IEEE Cloud Computing*, vol. 1, pp. 10–14, September 2014.

[12] M. Copeland, J. Soh, A. Puca, M. Manning, and D. Gollob, *Microsoft Azure: Planning, Deploying, and Managing Your Data center in the Cloud*. Apress, 2015.

[13] L. Badger, T. Grance, R. Patt-Corner, and J. Voas, "Cloud Computing Synopsis and Recommendations," Tech. Rep. SP 800-146, National Institute of Standards and Technology, May 2012.

[14] M. Bursell, A. Dutta, H.-L. Lu, M.-P. Odini, K. Roemer, K. Sood, M. Wong, and P. Wörndle, "Network Functions Virtualisation (NFV), NFV Security, Security and Trust Guidance, v.1.1.1," Tech. Rep. gs nfv-sec 003, European Telecommunications Standards Institute, December 2014.

[15] T. Grandison and M. Sloman, "A survey of trust in internet applications," *IEEE Comm. Surveys & Tutorials*, vol. 3, pp. 2–16, April 2000.

[16] E. J. Domingo, J. T. Nino, A. L. Lemos, M. L. Lemos, R. C. Palacios, and J. M. G. Berbís, "CLOUDIO: A Cloud Computing-Oriented Multi-tenant Architecture for Business Information Systems," in *Proc. 2010 IEEE 3rd International Conference on Cloud Computing*, CLOUD'10, pp. 532–533, IEEE, July 2010.

[17] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds," in *Proc. 16th ACM Conference on Computer and Communications Security*, CCS '09, (New York, NY, USA), pp. 199–212, ACM, November 2009.

[18] J. Somorovsky, M. Heiderich, M. Jensen, J. Schwenk, N. Gruschka, and L. Lo Iacono, "All Your Clouds Are Belong to us: Security Analysis of Cloud Management Interfaces," in *Proc. 3rd ACM Workshop on Cloud Computing Security*, CCSW '11, (New York, NY, USA), pp. 3–14, ACM, October 2011.

[19] F. Rocha and M. Correia, "Lucy in the sky without diamonds: Stealing confidential data in the cloud," in *Proc. 41st International Conference on Dependable Systems and Networks Workshops*, DSNW '11, pp. 129–134, IEEE, July 2011.

[20] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM Side Channels and Their Use to Extract Private Keys," in *Proc. 2012 ACM Conference on Computer and Communications Security*, CCS '12, pp. 305–316, ACM, Octber 2012.

[21] M. Jordon, "Cleaning up dirty disks in the cloud," *Network Security*, vol. 2012, pp. 12 – 15, October 2012.

[22] M. Metheny, *Federal Cloud Computing: The Definitive Guide for Cloud Service Providers*. Cambridge, USA: Syngress - Elsevier, 2017.

[23] O. Diez and A. Silva, "Govcloud: Using cloud computing in public organizations," *IEEE Technology and Society Magazine*, vol. 32, pp. 66–72, March 2013.

[24] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of remote attestation," *International Journal of Information Security*, vol. 10, pp. 63–81, April 2011.

[25] Trusted Computing Group, "Main Specification version 1.1b," Tech. Rep. -, Trusted Computing Group Inc., February 2002.

[26] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors," in *Proc. 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pp. 275–287, ACM, March 2007.

[27] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library Operating Systems for the Cloud," in *Proc. of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pp. 461–472, ACM, March 2013.

[28] A. Bratterud, A. A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum, "IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services," in *Proc. 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, CloudCom '15, pp. 250–257, IEEE, November 2015.

[29] D. C. Latham, "Department of defense trusted computer system evaluation criteria.," Tech. Rep. DoD 5200.28-STD, U.S. Department of Defense, December 1986.

[30] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in distributed systems: Theory and practice," *ACM Transactions of Computing Systems*, vol. 10, pp. 265–310, November 1992.

[31] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson, "The digital distributed system security architecture," in *Proc. 12th National Computer Security Conference*, NISSC '89, (Baltimore, Maryland, USA), pp. 305–319, NIST, 1989.

[32] B. Parno, J. M. McCune, and A. Perrig, "Bootstrapping trust in commodity computers," in *Proc. 2010 IEEE Symposium on Security and Privacy*, SP '10, pp. 414–429, IEEE, May 2010.

[33] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *Proc. 1997 IEEE Symposium on Security and Privacy*, SP '97, pp. 65–71, IEEE, May 1997.

[34] K. Kostiainen, E. Reshetova, J.-E. Ekberg, and N. Asokan, "Old, new, borrowed, blue – a perspective on the evolution of mobile platform security architectures," in *Proc. 1st ACM Conference on Data and Application Security and Privacy*, CODASPY '11, pp. 13–24, ACM, 2011.

[35] J.-E. Ekberg, K. Kostiainen, and N. Asokan, "Trusted execution environments on mobile devices," in *Proc. 2013 ACM SIGSAC conference on Computer & communications security*, CCS '13, pp. 1497–1498, ACM, 2013.

[36] Trusted Computing Group, "TPM Main Specification Level 2 Version 1.2, Revision 116. Parts 1-3," Tech. Rep. 116_01032011, Trusted Computing Group Inc., March 2011.

[37] T. Nyman, J.-E. Ekberg, and N. Asokan, "Citizen Electronic Identities Using TPM 2.0," in *Proc. 4th International Workshop on Trustworthy Embedded Devices*, TrustED '14, pp. 37–48, ACM, 2014.

[38] Trusted Computing Group, "Trusted Platform Module Library Specification, Family "2.0", Level 00, Revision 00.99. Parts 1-4." `http://www.trustedcomputinggroup.org/resources/tpm_library_specification`, October 2014. Accessed: 2017-04-21.

[39] W. Arthur and D. Challener, *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. Berkely, CA, USA: Apress, 2015.

[40] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn, "vTPM: virtualizing the trusted platform module," in *Proc. 15th conference on USENIX Security Symposium*, USENIX Security '06, USENIX, May 2006.

[41] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten, "fTPM: A Software-Only Implementation of a TPM Chip," in *Proc. 25th USENIX Security Symposium*, USENIX Security '16, pp. 841–856, USENIX, 2016.

[42] Trusted Computing Group, "Trusted Platform Module Library Specification, Family "2.0", Level 00, Revision 01.16," Tech. Rep. 120_01102013, Trusted Computing Group Inc., October 2014.

[43] S. W. Smith and S. Weingart, "Building a high-performance, programmable secure coprocessor," *Computer Networks*, vol. 31, pp. 831 – 860, April 1999.

[44] J. Reid, J. M. G. Nieto, E. Dawson, and E. Okamoto, "Privacy and trusted computing," in *Proc. 14th International Workshop on Database and Expert Systems Applications*, pp. 383–388, IEEE, Sept 2003.

[45] S. Balfe, E. Gallery, C. Mitchell, and K. Paterson, "Challenges for Trusted Computing," Tech. Rep. RHUL–MA–2008–14, Royal Holloway, University of London, February 2008.

[46] F. Stumpf, O. Tafreschi, P. Röder, and C. Eckert, "A robust integrity reporting protocol for remote attestation," in *Proc. of the Second Workshop on Advances in Trusted Computing*, WATC'06, 2006.

[47] Intel, "Trusted Execution Technology Software Development Guide," Tech. Rep. 315168-013, Intel Inc., August 2016.

[48] J. Butterworth, C. Kallenberg, X. Kovah, and A. Herzog, "Bios chronomancy: Fixing the core root of trust for measurement," in *Proc. 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pp. 25–36, ACM, November 2013.

[49] Intel, "Intel 64 and ia-32 architectures developer's manual," Tech. Rep. 252046-052, Intel Inc., September 2011.

[50] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Proc. 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, p. 10, ACM, June 2013.

[51] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution," in *Proc. 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pp. 10:1–10:1, ACM, June 2013.

[52] B. C. Xing, M. Shanahan, and R. Leslie-Hurd, "Intel Software Guard Extensions (Intel SGX) Software Support for Dynamic Memory Allocation Inside an Enclave," in *Proc. 2016 Hardware and Architectural Support for Security and Privacy*, HASP '16, pp. 11:1–11:9, ACM, June 2016.

[53] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel Software Guard Extensions (Intel SGX) Support for Dynamic Memory Management Inside an Enclave," in *Proc. 2016 Hardware and Architectural Support for Security and Privacy*, HASP '16, pp. 10:1–10:9, ACM, June 2016.

[54] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4," Tech. Rep. 325462-063US, Intel Inc., July 2017.

[55] E. Brickell and J. Li, "Enhanced Privacy ID: A Direct Anonymous Attestation Scheme with Enhanced Revocation Capabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, pp. 345–360, May 2012.

[56] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *Proc. 2015 IEEE Symposium on Security and Privacy*, SP '15, pp. 640–656, IEEE, May 2015.

[57] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, "AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves," in *Proc. 21st European Symposium on Research in Computer Security*, ESORICS 2016, pp. 440–457, Springer, September 2016.

[58] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using sgx to conceal cache attacks," Tech. Rep. arXiv:1702.08719, Graz University of Technology, March 2017.

[59] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: Sgx cache attacks are practical," Tech. Rep. arXiv:1702.07521, Technical University Darmstadt, February 2017.

[60] S. Checkoway and H. Shacham, "Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface," in *Proc. 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pp. 253–264, ACM, March 2013.

[61] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," *ACM Transactions on Computer Systems*, vol. 33, pp. 8:1–8:26, August 2015.

[62] B. Parno, "Bootstrapping Trust in a "Trusted" Platform," in *Proc. 3rd Conference on Hot Topics in Security*, HOTSEC'08, pp. 9:1–9:6, USENIX, July 2008.

[63] F. Schuster, M. Costa, C. Gkantsidis, M. Peinado, G. Mainar-ruiz, and M. Russinovich, "VC3 : Trustworthy Data Analytics in the Cloud using SGX," in *Proc. 2015 IEEE Symposium on Security and Privacy*, SP '15, pp. 38–54, IEEE, May 2015.

[64] P. Mell and T. Gance, "The NIST Definition of Cloud Computing," Tech. Rep. SP 800-145, National Institute of Standards and Technology, September 2011.

[65] S. A. Baset, "Open Source Cloud Technologies," in *Proc. of the Third ACM Symposium on Cloud Computing*, SoCC '12, pp. 28:1–28:2, ACM, October 2012.

[66] G. von Laszewski, J. Diaz, F. Wang, and G. C. Fox, "Comparison of multiple cloud frameworks," in *Proc. IEEE Fifth International Conference on Cloud Computing*, SoCC '12, pp. 734–741, IEEE, June 2012.

[67] M. Katzer and D. Crawford, *Office 365: Migrating and Managing Your Business in the Cloud*, pp. 1–23. Berkeley, CA: Apress, 2013.

[68] D. Nowak and K. Kurbel, "Understanding the Flexibility of Cloud ERP Software," in *Proc. 5th International Conference on Innovations in Enterprise Information Systems Management and Engineering*, ERP Future 2016, pp. 135–146, Springer International Publishing, November 2017.

[69] B. Kehoe, A. Matsukawa, S. Candido, J. Kuffner, and K. Goldberg, "Cloud-based robot grasping with the google object recognition engine," in *Proc. 2013 IEEE International Conference on Robotics and Automation*, ICRA'13, pp. 4263–4270, May 2013.

[70] J. Dilley, P. Laghate, J. Summers, and T. Devanneaux, "Cloud based firewall system and service," June 2013. US Patent 8,458,769.

[71] R. Mijumbi, J. Serrat, J. L. Gorricho, N. Bouten, F. D. Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys Tutorials*, vol. 18, pp. 236–262, First quarter 2016.

[72] D. Weerasiri, M. C. Barukh, B. Benatallah, Q. Z. Sheng, and R. Ranjan, "A Taxonomy and Survey of Cloud Resource Orchestration Techniques," *ACM Comput. Surv.*, vol. 50, pp. 26:1–26:41, May 2017.

[73] G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *Commun. ACM*, vol. 17, pp. 412–421, July 1974.

[74] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, C. Pu, and Y. Cao, "Who Is Your Neighbor: Net I/O Performance Interference in Virtualized Clouds," *IEEE Transactions on Services Computing*, vol. 6, pp. 314–329, July 2013.

[75] C. Delimitrou and C. Kozyrakis, "Bolt: I Know What You Did Last Summer... In The Cloud," in *Proc. of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pp. 599–613, ACM, April 2017.

[76] Advanced Micro Devices, "Amd64 architecture programmer's manual volume 2: System programming," Tech. Rep. 24593-3.28, AMD, March 2016.

[77] Advanced Micro Devices, "Secure Encrypted Virtualization Key Management," Tech. Rep. 55766-3.01, AMD, August 2016.

[78] J. Goodacre, "The evolution of the arm architecture towards big data and the datacentre," in *Proc. of the 8th Workshop on Virtualization in High-Performance Cloud Computing*, VHPC '13, pp. 4:1–4:1, ACM, November 2013.

[79] ARM, "Building a Secure System using TrustZone Technology," Tech. Rep. PRD29-GENC-009492C, ARM, April 2009.

[80] P. Williams and R. Boivie, "Cpu support for secure executables," in *Proc. 4th International Conference on Trust and Trustworthy Computing*, TRUST '11, pp. 172–187, Springer, June 2011.

[81] P. England and J. Loeser, "Para-Virtualized TPM Sharing," in *Proc. of the 1st International Conference on Trusted Computing and Trust in Information Technologies: Trusted Computing - Challenges and Applications*, Trust '08, pp. 119–132, Springer-Verlag, March 2008.

[82] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling, "Data Sharing Options for Scientific Workflows on Amazon EC2," in *Proc. 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pp. 1–9, November 2010.

[83] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-performance Distributed File System," in *Proc. 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pp. 307–320, USENIX, November 2006.

[84] A. Davies and A. Orsaria, "Scale out with GlusterFS," *Linux Journal*, vol. 2013, pp. 72–82, November 2013.

[85] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," in *Proc. 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pp. 205–220, ACM, October 2007.

[86] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s Hosted Data Serving Platform," *Proc. of the VLDB Endowment*, vol. 1, pp. 1277–1288, August 2008.

[87] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel, *et al.*, "Finding a Needle in Haystack: Facebook's Photo Storage.," in *Proc. 9th USENIX conference on Operating systems design and implementation*, OSDI '10, pp. 46–60, USENIX, October 2010.

[88] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, "Windows azure storage: A highly available cloud storage service with strong consistency," in *Proc. 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 143–157, ACM, 2011.

[89] S. A. Noghabi, S. Subramanian, P. Narayanan, S. Narayanan, G. Holla, M. Zadeh, T. Li, I. Gupta, and R. H. Campbell, "Ambry: LinkedIn's Scalable Geo-Distributed Object Store," in *Proc. 2016 International Conference on Management of Data*, SIGMOD '16, pp. 253–265, ACM, June 2016.

[90] S. Zhang, X. Zhang, and X. Ou, "After We Knew It: Empirical Study and Modeling of Cost-effectiveness of Exploiting Prevalent Known Vulnerabilities Across IaaS Cloud," in *Proc. 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pp. 317–328, ACM, 2014.

[91] Z. Zhang, C. Wu, and D. W. Cheung, "A survey on cloud interoperability: Taxonomies, standards, and practice," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, pp. 13–22, April 2013.

[92] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A clean slate 4D approach to network control and management," *ACM SIGCOMM Computer Communication Review*, vol. 35, pp. 41–54, October 2005.

[93] W. Zhou, L. Li, M. Luo, and W. Chou, "REST API Design Patterns for SDN Northbound API," in *Proc. 28th International Conference on Advanced Information Networking and Applications Workshops*, WAINA '14, pp. 358–365, IEEE, May 2014.

[94] OpenDaylight Project, "OpenDaylight Documentation," Tech. Rep. r.0.3.0, OpenDaylight Foundation, April 2016.

[95] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The Design and Implementation of Open vSwitch," in *Proc. 12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '15, pp. 117–130, USENIX, May 2015.

[96] O. S. Consortium, "OpenFlow switch specification, version 1.3.5," Tech. Rep. ONF TS-023, Open Networking Foundation, March 2015.

[97] R. Bifulco, J. Boite, M. Bouet, and F. Schneider, "Improving SDN with InSPired Switches," in *Proc. Symposium on SDN Research*, SOSR '16, pp. 11:1–11:12, ACM, March 2016.

[98] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, pp. 105–110, July 2008.

[99] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang, "Rosemary: A Robust, Secure, and High-Performance Network Operating System," in *Proc. 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pp. 78–89, ACM, November 2014.

[100] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: Towards an Open, Distributed SDN OS," in *Proc. 3rd Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pp. 1–6, ACM, August 2014.

[101] A. Regalado, "Who Coined 'Cloud Computing'?," *MIT Technology Review*, October 2011. Accessed 2017-04-11.

[102] R. L. Krutz and R. D. Vines, *Cloud Security: A Comprehensive Guide to Secure Cloud Computing*. New Jersey, USA: John Wiley & Sons, August 2010.

[103] B. D. Payne, M. D. P. D. A. Carbone, and W. Lee, "Secure and Flexible Monitoring of Virtual Machines," in *Proc. Twenty-Third Annual Computer Security Applications Conference*, ACSAC '07, pp. 385–397, December 2007.

[104] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection," in *Proc. 2011 IEEE Symposium on Security and Privacy*, SP '11, pp. 297–312, IEEE, May 2011.

[105] D. Birk and C. Wegener, "Technical Issues of Forensic Investigations in Cloud Computing Environments," in *Proc. Sixth IEEE International Workshop on Systematic Approaches to Digital Forensic Engineering*, SADFE '11, pp. 1–10, May 2011.

[106] F. Yao, R. Sprabery, and R. H. Campbell, "CryptVMI: A Flexible and Encrypted Virtual Machine Introspection System in the Cloud," in *Proc. of the 2nd International Workshop on Security in Cloud Computing*, SCC '14, pp. 11–18, ACM, 2014.

[107] F. Zhang, J. Chen, H. Chen, and B. Zang, "CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization," in *Proc. 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 203–216, ACM, October 2011.

[108] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB Reduction and Attestation," in *Proc. 2010 IEEE Symposium on Security and Privacy*, SP '10, pp. 143–158, May 2010.

[109] H. Raj, D. Robinson, T. B. Tariq, P. Engl, S. Saroiu, and A. Wolman, "Credo: Trusted Computing for Guest VMs with a Commodity Hypervisor," Tech. Rep. MSR-TR-2011-130, Microsoft Research, December 2011.

[110] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "MiniBox: A Two-way Sandbox for x86 Native Code," in *Proc. 2014 USENIX Annual Technical Conference*, USENIX ATC '14, pp. 409–420, USENIX, June 2014.

[111] S. Butt, H. A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy, "Self-service Cloud Computing," in *Proc. of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pp. 253–264, ACM, October 2012.

[112] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes," in *Proc. 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pp. 335–350, ACM, October 2007.

[113] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An Execution Infrastructure for TCB Minimization," in *Proc. 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, Eurosys '08, pp. 315–328, ACM, April 2008.

[114] S. Jin, J. Ahn, S. Cha, and J. Huh, "Architectural Support for Secure Virtualization Under a Vulnerable Hypervisor," in *Proc. 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pp. 272–283, ACM, December 2011.

[115] P. Maene, J. Gotzfried, R. de Clercq, T. Muller, F. Freiling, and I. Verbauwhede, "Hardware-Based Trusted Computing Architectures for Isolation and Attestation," *IEEE Transactions on Computers*, vol. PP, pp. 1–1, January 2017.

[116] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in *Proc. 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pp. 689–703, USENIX, November 2016.

[117] A. J. Benjamin, "Improving information storage reliability using a data network," Tech. Rep. MIT/LCS/TM-78, Massachussets Institute of Technology, Cambridge Lab for Computer Science, October 1976.

[118] D. L. Burkes and R. K. Treiber, "Design approaches for real-time transaction processing remote site recovery," in *Proc. 35th IEEE Computer Society International Conference*, Compcon '90, pp. 568–572, IEEE, February 1990.

[119] R. P. King, N. Halim, H. Garcia-Molina, and C. A. Polyzois, "Management of a Remote Backup Copy for Disaster Recovery," *ACM Transactions on Database Systems*, vol. 16, pp. 338–368, May 1991.

[120] Y. Ofek, "Method and apparatus for mirroring data in a remote data storage system," August 1999. Patent Application US5933653A.

[121] J. Reardon, D. Basin, and S. Capkun, "SoK: Secure Data Deletion," in *Proc. 2013 IEEE Symposium on Security and Privacy*, SP '13, pp. 301–315, IEEE, May 2013.

[122] S. Kamara and K. Lauter, "Cryptographic cloud storage," in *14th International Conference on Financial Cryptograpy and Data Security*, FC '10, pp. 136–149, Springer, January 2010.

[123] D. X. Song, D. Wagner, and A. Perrig, "Practical Techniques for Searches on Encrypted Data," in *Proc. 2000 IEEE Symposium on Security and Privacy*, SP '00, pp. 44–55, IEEE, 2000.

[124] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano, "Public Key Encryption with Keyword Search," in *Proc. International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004. Proc.*, Advances in Cryptology - EUROCRYPT 2004, pp. 506–522, Springer Berlin Heidelberg, 2004.

[125] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-based Encryption for Fine-grained Access Control of Encrypted Data," in *Proc. of the 13th ACM Conference on Computer and Communications Security*, CCS '06, (New York, NY, USA), pp. 89–98, ACM, 2006.

[126] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable data possession at untrusted stores," in *Proc. of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pp. 598–609, ACM, 2007.

[127] Y. Tang, P. P. C. Lee, J. C. S. Lui, and R. Perlman, *FADE: Secure Overlay Cloud Storage with File Assured Deletion*, pp. 380–397. SecureComm '10, Springer, September 2010.

[128] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-policy attribute-based encryption," in *Proc. 2007 IEEE Symposium on Security and Privacy*, SP '07, pp. 321–334, IEEE, May 2007.

[129] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu, "Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services," in *Proc. 21st USENIX Security Symposium*, USENIX Security '12, pp. 175–188, USENIX, August 2012.

[130] R. A. Popa, E. Stark, J. Helfer, S. Valdez, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan, "Building Web Applications on Top of Encrypted Data Using Mylar," in *Proc. 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pp. 157–172, USENIX Association, April 2014.

[131] N. Paladi, C. Gehrmann, M. Aslam, and F. Morenius, "Trusted Launch of Virtual Machine Instances in Public IaaS Environments," in *Proc. 15th International Conference on Information Security and Cryptology*, ICISC' 12, pp. 309–323, Springer, December 2012.

[132] N. Paladi, C. Gehrmann, and F. Morenius, "Domain-Based Storage Protection (DBSP) in Public Infrastructure Clouds," in *Proc. 18th Nordic Conference on Secure IT Systems*, NordSec '13, pp. 279–296, Springer, October 2013.

[133] C. Gehrmann, F. Morenius, and N. Paladi, "Procedure For Platform Enforced Storage in Infrastructure Clouds," March 2016. Patent Application WO2014185845 A1.

[134] F. Zhang and H. Zhang, "SoK: A Study of Using Hardware-assisted Isolated Execution Environments for Security," in *Proc. Hardware and Architectural Support for Security and Privacy 2016*, HASP 2016, pp. 3:1–3:8, ACM, June 2016.

[135] L. M. Kaufman, "Data Security in the World of Cloud Computing," *IEEE Security Privacy*, vol. 7, pp. 61–64, July 2009.

[136] A. Chander and U. P. Le, "Breaking the Web: Data Localization vs. the Global Internet," Tech. Rep. No. 378, UC Davis, March 2014.

[137] J. Selby, "Data localization laws: trade barriers or legitimate responses to cybersecurity risks, or both?," *International Journal of Law and Information Technology*, vol. eax010, July 2017.

[138] Australia House of Representatives, "Personally Controlled Electronic Health Records Act 2012," November 2011. [Online, accessed 2017-07-24].

[139] KPMG China, "Overview of China's Cybersecurity Law," February 2017. [Online, accessed 2017-07-24].

[140] German Bundestag, "Entwurf eines Gesetzes zur Einführung einer Speicherpflicht und einer Höchstspeicherfrist für Verkehrsdaten," June 2015. [Online, accessed 2017-07-24].

[141] Ministry of Science and Technology of India, "National Data Sharing and Accessibility Policy," March 2012. [Online, accessed 2017-07-24].

[142] Government of the Republic of Indonesia, "Regulation of the Government of the Republic of Indonesia Number 82 of 2012," October 2012. [Online, accessed 2017-07-24].

[143] State Duma of the Russian Federation, "On making amendments to certain laws of the Russian Federation regarding clarification of the order of processing of personal data in information and telecommunication networks," July 2014. [Online, accessed 2017-07-24].

[144] Defense Acquisition Regulations System, Department of Defense, "Defense Federal Acquisition Regulation Supplement: Network Penetration Reporting and Contracting for Cloud Services," *Federal Register*, vol. 80, pp. 51742–51748, August 2015.

[145] G. J. Watson, R. Safavi-Naini, M. Alimomeni, M. E. Locasto, and S. Narayan, "LoSt: Location Based Storage," in *Proc. 2012 ACM Cloud Computing Security Workshop*, CCSW '12, pp. 59–70, ACM, October 2012.

[146] K. Benson, R. Dowsley, and H. Shacham, "Do You Know Where Your Cloud Files Are?," in *Proc. 3rd ACM Cloud Computing Security Workshop*, CCSW '11, pp. 73–82, ACM, October 2011.

[147] A. Albeshri, C. Boyd, and J. G. Nieto, "GeoProof: Proofs of Geographic Location for Cloud Computing Environment," in *Proc. 32nd International Conference on Distributed Computing Systems Workshops*, ICDCSW '12, pp. 506–514, IEEE, August 2012.

[148] A. Albeshri, C. Boyd, and J. Nieto, "Enhanced GeoProof: improved geographic assurance for data in the cloud," *International Journal of Information Security*, vol. 13, pp. 191–198, April 2013.

[149] Z. N. J. Peterson, M. Gondree, and R. Beverly, "A Position Paper on Data Sovereignty: The Importance of Geolocating Data in the Cloud," in *Proc. 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'11, pp. 9–9, USENIX, June 2011.

[150] M. Gondree and Z. N. Peterson, "Geolocation of Data in the Cloud," in *Proc. 3rd ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pp. 25–36, ACM, February 2013.

[151] E. Banks, M. Bartock, K. Firftal, D. Lemon, K. Scarfone, U. Shetty, M. Souppaya, T. Williams, and R. Yeluri, "Trusted Geolocation in the Cloud: Proof of Concept Implementation," Tech. Rep. NIST 7904, National Institute of Standards and Technology, December 2015.

[152] K. R. Jayaram, D. Safford, U. Sharma, V. Naik, D. Pendarakis, and S. Tao, "Trustworthy Geographically Fenced Hybrid Clouds," in *Proc. 15th International Middleware Conference*, Middleware '14, pp. 37–48, ACM, 2014.

[153] S. Park, J.-J. Won, J. Yoon, K. H. Kim, and T. Han, "A tiny hypervisor-based trusted geolocation framework with minimized TPM operations," *Journal of Systems and Software*, vol. 122, pp. 202 – 214, 2016.

[154] J. Arnold, *Openstack swift: Using, administering, and developing for swift object storage.* " O'Reilly Media, Inc.", 2014.

[155] D. Kreutz, F. Ramos, and P. Verissimo, "Towards secure and dependable software-defined networks," in *Proc. 2nd ACM SIGCOMM workshop on Hot topics in software defined networking*, HotSDN '13, pp. 55–60, ACM, August 2013.

[156] S. Scott-Hayward, S. Natarajan, and S. Sezer, "A survey of security in software defined networks," *IEEE Comm. Surveys & Tutorials*, vol. 18, pp. 623–654, July 2015.

[157] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the Data Plane with Anteater," in *Proc. ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pp. 290–301, ACM, August 2011.

[158] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for OpenFlow networks," in *Proc. 1st Workshop on Hot topics in software defined networks*, HotSDN 12, pp. 121–126, ACM, August 2012.

[159] S. Shin, P. A. Porras, V. Yegneswaran, M. W. Fong, G. Gu, and M. Tyson, "FRESCO: Modular Composable Security Services for Software-Defined Networks," in *Proc. 20th Annual Network & Distributed System Security Symposium*, NDSS '13, Internet Society, February 2013.

[160] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao, "FLOWGUARD: Building Robust Firewalls for Software-defined Networks," in *Proc. 3rd Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pp. 97–102, ACM, August 2014.

[161] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky, "VeriCon: Towards Verifying Controller Programs in Software-defined Networks," in *Proc. 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pp. 282–293, ACM, June 2014.

[162] I. Nikolaevskiy, A. Lukyanenko, and A. Gurtov, "Stem+: Allocating bandwidth fairly to tasks," in *2015 IEEE Conference on Computer Communications Workshops*, INFOCOM WKSHPS '15, pp. 67–68, IEEE, April 2015.

[163] I. Ahmad, S. Namal, M. Ylianttila, and A. Gurtov, "Security in Software Defined Networks: A Survey," *IEEE Communications Surveys Tutorials*, vol. 17, pp. 2317–2346, Fourthquarter 2015.

[164] T. Koponen and A. Gurtov, "Application mobility with Host Identity Protocol," in *In Proc. of NDSS Wireless and Security Workshop*, Internet Society, 2005.

[165] M. Casado, N. Foster, and A. Guha, "Abstractions for Software-defined Networks," *Communications of the ACM*, vol. 57, pp. 86–95, September 2014.

[166] N. Paladi and L. Karlsson, "Safeguarding VNF Credentials with Intel SGX," in *Proceedings of the SIGCOMM Posters and Demos*, SIGCOMM Posters and Demos '17, pp. 144–146, ACM, August 2017.

[167] M. Antikainen, T. Aura, and M. Särelä, *Spook in Your Network: Attacking an SDN with a Compromised OpenFlow Switch*, pp. 229–244. NordSec '14, Springer, October 2014.

[168] K. Thimmaraju, B. Shastry, T. Fiebig, F. Hetzelt, J.-P. Seifert, A. Feldmann, and S. Schmid, "Reigns to the Cloud: Compromising Cloud Systems via the Data Plane," Tech. Rep. arXiv:1610.08717, Technical University of Berlin, October 2016.

[169] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska, "S-NFV: Securing NFV States by Using SGX," in *Proc. 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, SDN-NFV Security '16, pp. 45–48, ACM, March 2016.

[170] S. Kim, J. Han, J. Ha, T. Kim, and D. Han, "Enhancing security and privacy of tor's ecosystem by using trusted execution environments," in *14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '17, pp. 145–161, USENIX, Month 2017.

[171] M. Coughlin, E. Keller, and E. Wustrow, "Trusted Click: Overcoming Security Issues of NFV in the Cloud," in *Proc. ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, SDN-NFVSec '17, pp. 31–36, ACM, March 2017.

[172] F. Klaedtke, G. O. Karame, R. Bifulco, and H. Cui, "Access Control for SDN Controllers," in *Proc. 3rd Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pp. 219–220, ACM, August 2014.

[173] D. Gkounis, F. Klaedtke, R. Bifulco, and G. O. Karame, "Cases for Including a Reference Monitor to SDN," in *Proc. 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pp. 599–600, ACM, August 2016.

[174] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking Control of the Enterprise," in *Proc. 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, pp. 1–12, ACM, August 2007.

[175] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, "Hierarchical Policies for Software Defined Networks," in *Proc. 1st Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pp. 37–42, ACM, August 2012.

[176] D. Yu, "Access control for network management," Tech. Rep. UCAM-CL-TR-898, University of Cambridge, Computer Laboratory, January 2017.

[177] Y. Han, J. Li, D. Hoang, J. H. Yoo, and J. W. K. Hong, "An intent-based network virtualization platform for sdn," in *Proc. 12th International Conference on Network and Service Management*, CNSM '16, pp. 353–358, IEEE, October 2016.

[178] ONOS, "ONOS intent framework." ONOS Project Code Implemenetation, May 2017.

[179] A. K. Sood and R. J. Enbody, "Targeted Cyberattacks: A Superset of Advanced Persistent Threats," *IEEE Security & Privacy*, vol. 11, pp. 54–61, January 2013.

[180] N. Santos, K. P. Gummadi, and R. Rodrigues, "Towards Trusted Cloud Computing," in *Proc. 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, USENIX, June 2009.

[181] N. Paladi, C. Gehrmann, and A. Michalas, "Providing User Security Guarantees in Public Infrastructure Clouds," *IEEE Transactions on Cloud Computing*, vol. PP, pp. 1–1, February 2016.

[182] R. Yeluri and E. Castro-Leon, "The trusted cloud: addressing security and compliance," in *Building the Infrastructure for Cloud Security*, ch. 2, pp. 19–36, Berkely, CA, USA: Apress, 2014.

[183] R. Yeluri and E. Castro-Leon, *Building the Infrastructure for Cloud Security: A Solutions View*. Berkely, CA, USA: Apress, 2014.

[184] P. Jain, S. Desai, S. Kim, M.-W. Shih, J. Lee, C. Choi, Y. Shin, T. Kim, B. B. Kang, and D. Han, "OpenSGX: An Open Platform for SGX Research," in *Proc. 2016 Network and Distributed System Security Symposium*, NDSS '16, Internet Society, February 2016.

[185] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, June 2005.

[186] R. Neisse, D. Holling, and A. Pretschner, "Implementing Trust in Cloud Infrastructures," in *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '11, pp. 524 –533, IEEE, May 2011.

[187] A.-R. Sadeghi, C. Stüble, and M. Winandy, "Property-Based TPM Virtualization," in *International Conference on Information Security*, ISC '08, pp. 31–46, Springer, September 2008.

[188] B. Danev, R. J. Masti, G. O. Karame, and S. Capkun, "Enabling Secure VM-vTPM Migration in Private Clouds," in *Proc. 27th Annual Computer Security Applications Conference*, ACSAC '11, pp. 187–196, ACM, December 2011.

[189] M. Aslam, C. Gehrmann, L. Rasmusson, and M. Björkman, "Securely Launching Virtual Machines on Trustworthy Platforms in a Public Cloud - An Enterprise's Perspective," in *Proc. 2nd International Conference on Cloud Computing and Services Science*, CLOSER '12, pp. 511–521, Springer, April 2012.

[190] M. Aslam, C. Gehrmann, and M. Björkman, "Security and Trust Preserving VM Migrations in Public Clouds," in *11th International Conference on Trust, Security and Privacy in Computing and Communications*, TrustCom '12, pp. 869–876, IEEE, June 2012.

[191] P. Goyal, "Application of a Distributed Security Method to End-2-End Services Security in Independent Heterogeneous Cloud Computing Environments," in *2011 IEEE World Congress on Services*, pp. 379 –384, IEEE, July 2011.

[192] V. Boyko, P. MacKenzie, and S. Patel, "Provably secure password-authenticated key exchange using diffie-hellman," in *Proc. 19th international conference on Theory and application of cryptographic techniques*, EUROCRYPT '00, pp. 156–171, Springer, 2000.

[193] M. Price, "The Paradox of Security in Virtual Environments," *Computer*, vol. 41, pp. 22–28, November 2008.

[194] R. Wojtczuk, J. Rutkowska, and A. Tereshkin, "Attacking intel trusted execution technology," Tech. Rep. -, Invisible Things Lab, August 2008.

[195] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest We Remember: Cold-Boot Attacks on Encryption Keys," *Communications of the ACM*, vol. 52, pp. 91–98, May 2009.

[196] J. Schiffman, T. Moyer, H. Vijayakumar, T. Jaeger, and P. McDaniel, "Seeding Clouds With Trust Anchors," in *Proc. 2010 ACM Workshop on Cloud Computing Security*, CCSW '10, pp. 43–46, ACM, October 2010.

[197] D. Molnar and S. Schechter, "Self Hosting vs . Cloud Hosting : Accounting for the Security Impact of Hosting in the Cloud," in *Proc. of the Ninth Workshop on the Economics of Information Security*, WEIS '10, pp. 1–18, ACM, June 2010.

[198] S. Y. Ko, K. Jeon, and R. Morales, "The HybrEx Model for Confidentiality and Privacy in Cloud Computing," in *Proc. 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'11, pp. 8–8, USENIX, June 2011.

[199] N. Paladi, A. Michalas, and C. Gehrmann, "Domain Based Storage Protection with Secure Access Control for the Cloud," in *Proc. 2014 International Workshop on Security in Cloud Computing*, SCC '14, pp. 35–42, ACM, June 2014.

[200] Top Threats Working Group, "The notorious nine cloud computing top threats 2013," Tech. Rep. -, Cloud Security Alliance, February 2013.

[201] A. Michalas, N. Paladi, and C. Gehrmann, "Security aspects of e-health systems migration to the cloud," in *Proc. 16th International Conference on e-Health Networking, Applications and Services*, HealthCom '14, pp. 212–218, IEEE, October 2014.

[202] B. Bertholon, S. Varrette, and P. Bouvry, "Certicloud: A Novel TPM-based Approach to Ensure Cloud IaaS Security," in *Proc. 2011 IEEE International Conference on Cloud Computing*, CLOUD, pp. 121–130, IEEE, September 2011.

[203] A. Cooper and A. Martin, "Towards a secure, tamper-proof grid platform," in *Proc. 6th IEEE International Symposium on Cluster Computing and the Grid*, CCGrid '06, pp. 372–380, IEEE, May 2006.

[204] W. Wang, Z. Li, R. Owens, and B. Bhargava, "Secure and efficient access to outsourced data," in *Proc. 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, pp. 55–66, ACM, November 2009.

[205] D. Song, E. Shi, I. Fischer, and U. Shankar, "Cloud data protection for the masses," *IEEE Computer*, vol. 45, pp. 39–45, January 2012.

[206] S. Graf, P. Lang, S. A. Hohenadel, and M. Waldvogel, "Versatile key management for secure cloud storage," in *Proc. 31st Symposium on Reliable Distributed Systems*, SRDS '12, pp. 469–474, IEEE, October 2012.

[207] A.-R. Sadeghi and C. Stüble, "Property-based attestation for computing platforms: caring about properties, not mechanisms," in *Proc. 2004 workshop on New security paradigms*, NSPW '04, pp. 67–77, ACM, September 2004.

[208] A. Sahai and B. Waters, "Fuzzy identity-based encryption," in *Proc. 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT '05, pp. 457–473, Springer, 2005.

[209] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in *Proc. 17th International Conference on Financial Cryptography and Data Security*, pp. 258–274, Springer, April 2013.

[210] C. Waldspurger and M. Rosenblum, "I/O virtualization," *Communications of the ACM*, vol. 55, pp. 66–73, January 2012.

[211] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, pp. 198–208, March 1983.

[212] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: a virtual machine-based platform for trusted computing," in *Proc. 19th ACM symposium on Operating Systems Principles*, SOSP '03, pp. 193–206, ACM, October 2003.

[213] S. Goldberg, "Why is it taking so long to secure internet routing?," *Communications of the ACM*, vol. 57, pp. 56–63, October 2014.

[214] B. Parno, J. M. McCune, and A. Perrig, *Bootstrapping Trust in Modern Computers*. Heidelberg, Germany: Springer, 2011.

[215] P. Eronen and H. Tschofenig, "Pre-shared key ciphersuites for transport layer security (TLS)," RFC 4279, RFC Editor, December 2005.

[216] B. Blanchet, "An Efficient Cryptographic Protocol Verifier Based on Prolog Rules," in *Proc. 14th IEEE Workshop on Computer Security Foundations*, CSFW '01, pp. 82–96, IEEE, 2001.

[217] N. Paladi and A. Michalas, ""One of our hosts in another country": Challenges of data geolocation in cloud storage," in *Proc. 4th International Conference on Wireless Communications, Vehicular Technology, Information Theory and Aerospace Electronic Systems*, VITAE '14, pp. 1–6, May 2014.

[218] M. Rezaei, N. S. Moosavi, H. Nemati, and R. Azmi, "TCvisor: A hypervisor level secure storage," in *Proc. 2010 International Conference for Internet Technology and Secured Transactions*, ICITST '10, pp. 1–9, IEEE, November 2010.

[219] M. Waldvogel, G. Caronni, D. Sun, N. Weiler, and B. Plattner, "The VersaKey framework: Versatile group key management," *IEEE Journal on Selected Areas in Communications*, vol. 17, pp. 1614–1631, September 1999.

[220] O. Goldreich, S. Goldwasser, and S. Micali, "How to construct random functions," *Journal of the ACM*, vol. 33, pp. 792–807, August 1986.

[221] M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro, "Reducing TCB Size by Using Untrusted Components: Small Kernels Versus Virtual-machine Monitors," in *Proc. 11th Workshop on ACM SIGOPS European Workshop*, EW '11, ACM, 2004.

[222] D. Dhar and J. H.-e. Ho, "Stem cell issue: stem cell research policies around the world," *The Yale journal of biology and medicine*, vol. 82, pp. 113–115, September 2009.

[223] A. Juels and A. Oprea, "New Approaches to Security and Availability for Cloud Data," *Communications of the ACM*, vol. 56, pp. 64–73, February 2013.

[224] D. Desai, "Beyond Location: Data Security in the 21st Century," *Communications of the ACM*, vol. 56, pp. 34–36, January 2013.

[225] J. Hamilton, "Architecture for modular data centers," *arXiv preprint cs/0612110*, 2006.

[226] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers," in *Proc. ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pp. 63–74, ACM, August 2009.

[227] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat, "Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers," in *Proc. ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pp. 339–350, ACM, August 2010.

[228] S. Toor, R. Toebbicke, M. Z. Resines, and S. Holmgren, "Investigating an open source cloud storage infrastructure for CERN-specific data analysis," in *7th International Conference on Networking, Architecture and Storage*, NAS, pp. 84–88, IEEE, June 2012.

[229] C. Krauß and V. Fusenig, "Using Trusted Platform Modules for Location Assurance in Cloud Networking," in *Proc. 7th International Conference on Network and System Security*, NSS '13, pp. 109–121, Springer, June 2013.

[230] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's Globally Distributed Database," *ACM Transactions on Computer Systems*, vol. 31, pp. 8:1–8:22, August 2013.

[231] J. Pearsall and P. Hanks, *The new Oxford dictionary of English*. Clarendon Press, 1998.

[232] C. Gentry, "Computing arbitrary functions of encrypted data," *Communications of the ACM*, vol. 53, pp. 97–105, March 2010.

[233] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *ACM Transactions on Computer Systems*, vol. 26, pp. 4:1–4:26, June 2008.

[234] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional Storage for Geo-replicated Systems," in *Proc. 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 385–400, ACM, 2011.

[235] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making Geo-replicated Systems Fast As Possible, Consistent when Necessary," in *Proc. 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pp. 265–278, USENIX, October 2012.

[236] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger Semantics for Low-latency Geo-replicated Storage," in *Proc. 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, pp. 313–328, USENIX, April 2013.

[237] D. W. Goldberg, J. P. Wilson, and C. A. Knoblock, "From text to geographic coordinates: the current state of geocoding," *URISA journal*, vol. 19, pp. 33–46, January 2007.

[238] C. J. Wullems, "A Spoofing Detection Method for Civilian L1 GPS and the E1-B Galileo Safety of Life Service," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 48, pp. 2849–2864, October 2012.

[239] B. W. O'Hanlon, M. L. Psiaki, J. A. Bhatti, D. P. Shepard, and T. E. Humphreys, "Real-Time GPS Spoofing Detection via Correlation of Encrypted Signals," *Journal of the Institute of Navigation*, vol. 60, pp. 267–278, December 2013.

[240] M. L. Psiaki, B. W. O'Hanlon, J. A. Bhatti, D. P. Shepard, and T. E. Humphreys, "GPS spoofing detection via dual-receiver correlation of military signals," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 49, pp. 2250–2267, 2013.

[241] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," Tech. Rep. OPENFLOW-TR-2009-1, OpenFlow Switch Consortium, October 2009.

[242] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker, "SANE: A Protection Architecture for Enterprise Networks," in *Proc. 15th USENIX Security Symposium*, USENIX Security '06, p. 137–151, USENIX, 2006.

[243] A. K. Nayak, A. Reimers, N. Feamster, and R. Clark, "Resonance: dynamic access control for enterprise networks," in *Proc. 1st ACM workshop on Research on enterprise networking*, WREN '09, pp. 11–18, ACM, August 2009.

[244] H. Song, "Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane," in *Proc. 2nd ACM SIGCOMM workshop on Hot topics in software defined networking*, HotSDN '13, pp. 127–132, ACM, August 2013.

[245] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A Distributed Control Platform for Large-scale Production Networks," in *Proc. 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, pp. 351–364, USENIX, October 2010.

[246] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, pp. 69–74, April 2008.

[247] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined wan," *ACM SIGCOMM Computer Communication Review*, vol. 43, pp. 3–14, August 2013.

[248] A. Wang, M. Iyer, R. Dutta, G. N. Rouskas, and I. Baldine, "Network Virtualization: Technologies, Perspectives, and Frontiers," *Journal of Lightwave Technology*, vol. 31, pp. 523–537, August 2013.

[249] S. Tripathi, N. Droux, T. Srinivasan, K. Belgaied, and V. Iyer, "Crossbow: A Vertically Integrated QoS Stack," in *Proc. 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, pp. 45–54, ACM, August 2009.

[250] N. Paladi, "Towards Secure SDN Policy Management," in *Proc. 8th International Conference on Utility and Cloud Computing*, UCC '15, pp. 607–611, December 2015.

[251] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran, "Securing the software-defined network control layer," in *Proc. 2015 Network and Distributed System Security Symposium*, NDSS '15, Internet Society, February 2015.

[252] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures," in *Proc. 2015 Network and Distributed System Security Symposium*, NDSS '15, Internet Society, February 2015.

[253] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying Middlebox Policy Enforcement Using SDN," in *Proc. ACM Conference of the Special Interest Group on Data Communication*, SIGCOMM '13, (New York, NY, USA), pp. 27–38, ACM, 2013.

[254] T. D. Nadeau and K. Gray, *SDN: Software Defined Networks*. Sebastopol, California, USA: O'Reilly Media, Inc., 2013.

[255] C. Hopps, "Analysis of an Equal-Cost Multi-Path Algorithm," RFC 2992, RFC Editor, November 2000.

[256] D. Farinacci, P. Traina, S. Hanks, and T. Li, "Generic routing encapsulation (GRE)," RFC 2992, RFC Editor, March 2000.

[257] J. R. Douceur, "The Sybil Attack," in *Proc. 1st International Workshop on Peer-to-Peer Systems*, IPTPS '02, pp. 251–260, Springer, March 2002.

[258] T. Ristenpart and S. Yilek, "When Good Randomness Goes Bad: Virtual Machine Reset Vulnerabilities and Hedging Deployed Cryptography," in *Proc. 2010 Network and Distributed System Security Symposium*, NDSS '10, Internet Society, February 2010.

[259] J. Walker and J. Li, "Key exchange with anonymous authentication using DAA-SIGMA protocol," in *Proc. 2nd International Conference on Trusted Systems*, INTRUST '11, pp. 108–127, Springer, December 2011.

[260] X. Ruan, *Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. Berkely, CA, USA: Apress, 1st ed., 2014.

[261] R. Bifulco, H. Cui, G. O. Karame, and F. Klaedtke, "Fingerprinting Software-Defined Networks," in *Proc. 23rd International Conference on Network Protocols*, ICNP '15, pp. 453–459, IEEE, November 2015.

[262] F.-C. Kuo, H. Tschofenig, F. Meyer, and X. Fu, "Comparison studies between pre-shared and public key exchange mechanisms for transport layer security," in *Proc. 25th IEEE International Conference on Computer Communications*, INFOCOM '06, pp. 1–6, IEEE, April 2006.

[263] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2," RFC 6347, RFC Editor, January 2012.

[264] J. Rutkowska, "Thoughts on Intel's upcoming Software Guard Extensions (Part 2)," 2013. [Online; March 2016].

[265] N. Paladi and C. Gehrmann, "TruSDN: Bootstrapping Trust in Cloud Network Infrastructure," in *Proc. 12th International Conference on Security and Privacy in Communication Networks*, SecureComm'16, pp. 104–124, Springer, October 2016.

[266] A. Doria, J. H. Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, R. Gopal, and J. Halpern, "Forwarding and control element separation (ForCES) Protocol Specification," RFC 5810, RFC Editor, March 2010.

[267] J. Halpern and J. H. Salim, "Forwarding and control element separation (ForCES) Forwarding Element Model," RFC 5812, RFC Editor, March 2010.

[268] C. Janz, N. Davis, D. Hood, M. Lemay, D. Lenrow, L. Fengka, F. Schneider, J. Strassner, and A. Veitch, "Intent NBI – Definition and Principles," Tech. Rep. ONF TR-523, Open Networking Foundation, October 2016.

[269] M. Monaco, O. Michel, and E. Keller, "Applying Operating System Principles to SDN Controller Design," in *Proc. 12th ACM Workshop on Hot Topics in Networks*, HotNets-XII, pp. 2:1–2:7, ACM, November 2013.

[270] E. Haleplidis, K. Patras, K. Pentikousis, S. Denazis, J. Hadi, S. Mojatatu, M. D., and O. Koufopavlou, "Forwarding and control element separation (ForCES) Protocol Specification," RFC 7426, RFC Editor, January 2015.

[271] S. Scott-Hayward, "Design and deployment of secure, robust, and resilient SDN controllers," in *Proc. 1st IEEE Conference on Network Softwarization*, NetSoft '15, pp. 1–5, IEEE, April 2015.

[272] A. R. Matthews and A. Naveed, "Distributed virtual system to support managed, network-based services," September 2010. Patent Grant US7389358 B1.

[273] Network Functions Virtualisation ETSI Industry Specification Group, "Network Functions Virtualisation (NFV); Management and Orchestration; Report on Architectural Options," Tech. Rep. DGS/NFV-IFA009, European Telecommunications Standards Institute, July 2016.

[274] L. Schiff, S. Schmid, and P. Kuznetsov, "In-Band Synchronization for Distributed SDN Control Planes," *ACM SIGCOMM Computer Communication Review*, vol. 46, pp. 37–43, January 2016.

[275] M. M. Buddhikot, "Understanding Dynamic Spectrum Access: Models,Taxonomy and Challenges," in *Proc. 2nd IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks*, DySPAN '07, pp. 649–663, IEEE, April 2007.

[276] R. Sherwood, M. Chan, A. Covington, G. Gibb, M. Flajslik, N. Handigol, T.-Y. Huang, P. Kazemian, M. Kobayashi, J. Naous, S. Seetharaman, D. Underhill, T. Yabe, K.-K. Yap, Y. Yiakoumis, H. Zeng, G. Appenzeller, R. Johari, N. McKeown, and G. Parulkar, "Carving Research Slices out of Your Production Networks with OpenFlow," *ACM SIGCOMM Computer Communication Review*, vol. 40, pp. 129–130, January 2010.

[277] T. L. Casavant and J. G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Transactions on Software Engineering*, vol. 14, pp. 141–154, Feb 1988.

[278] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, "Participatory Networking: An API for Application Control of SDNs," *ACM SIGCOMM Computer Communication Review*, vol. 43, pp. 327–338, August 2013.

[279] J. O. Kephart and W. E. Walsh, "An artificial intelligence perspective on autonomic computing policies," in *Proc. 5th IEEE International Workshop on Policies for Distributed Systems and Networks*, POLICY '04, pp. 3–12, IEEE, June 2004.

[280] D. Drutskoy, E. Keller, and J. Rexford, "Scalable Network Virtualization in Software-Defined Networksd," *IEEE Internet Computing*, vol. 17, pp. 20–27, March 2013.

[281] R. S. Fabry, "Capability-based Addressing," *Communications of the ACM*, vol. 17, pp. 403–412, July 1974.

[282] H. M. Levy, *Capability-based computer systems.* Newton, MA, USA: Butterworth-Heinemann, 1st ed., 1984.

[283] V. C. Hu, D. Ferraiolo, R. Kuhn, A. R. Friedman, A. J. Lang, M. M. Cogdell, A. Schnitzer, K. Sandlin, R. Miller, K. Scarfone, *et al.*, "Guide to attribute based access control (ABAC) definition and considerations," Tech. Rep. SP 800-162, National Institute of Standards and Technology, January 2014.

[284] W. Han and C. Lei, "A survey on policy languages in network and security management," *Computer Networks*, vol. 56, pp. 477 – 489, January 2012.

[285] Y. Bertrand, M. Blay-Fornarino, K. Boudaoud, and M. Riveill, "Generation of Transmission Control Rules Compliant with Existing Access Control Policies," in *Proc. 11th EAI International Conference onSecurity and Privacy in Communication Networks*, SecureComm '15, pp. 438–455, Springer, October 2015.

[286] R. W. Brown, R. Keller, M. S. Medin, and D. Temkin, "Method and system for restricting access to user resources," May 2004. Patent Grant US9038145 B2.

[287] G. A. Piccionelli and T. R. Rittmaster, "System and process for limiting distribution of information on a communication network based on geographic location," September 2010. Patent Grant US8352601 B2.

[288] J. Ingatius, P. Woerndle, A. Khan, M. Brenner, D. Clarke, M. Flauw, N. Khan, O. Le Grand, D. McDysan, O. Kazuaki, P. Yegani, and V. Young, "Network Function Virtualization (VNF) Virtualization Requirements," Tech. Rep. DGS/NFV-0012, European Telecommunications Standards Institute, October 2014.

[289] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen, "Policy Core Information Model–Version 1 Specification," RFC 3060, RFC Editor, February 2001.

[290] J. Greenfield and K. Short, "Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools," in *Proc. 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pp. 16–27, ACM, October 2003.

[291] OASIS XACML Technical Committee, "eXtensible access control markup language (XACML) Version 3.0," Tech. Rep. xacml-3.0-core-spec-cs-01-en, OASIS, August 2010.

[292] C. Späth, C. Mainka, V. Mladenov, and J. Schwenk, "SoK: XML Parser Vulnerabilities," in *Proc. 10th USENIX Conference on Offensive Technologies*, WOOT'16, pp. 141–154, USENIX, August 2016.

[293] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "SPHINX: Detecting Security Attacks in Software-Defined Networks.," in *Proc. 2015 Network and Distributed System Security Symposium*, NDSS '15, Internet Society, February 2015.

[294] K. Thompson, "Reflections on Trusting Trust," *Communications of the ACM*, vol. 27, pp. 761–763, August 1984.

[295] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Proc. 2015 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '15, pp. 171–172, IEEE, March 2015.

[296] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A Network Programming Language," in *Proc. 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pp. 279–291, ACM, September 2011.

[297] A. Voellmy, H. Kim, and N. Feamster, "Procera: A Language for High-level Reactive Network Control," in *Proc. 1st Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, ACM, August 2012.

[298] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing Software-defined Networks," in *Proc. 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, pp. 1–14, USENIX, August 2013.

[299] J. McClurg, H. Hojjat, N. Foster, and P. Černý, "Event-driven Network Programming," in *Proc. 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'16, pp. 369–385, ACM, June 2016.

[300] C. Caba and J. Soler, "APIs for QoS configuration in Software Defined Networks," in *Proc. 1st IEEE Conference on Network Softwarizationx*, NetSoft '15, pp. 1–5, IEEE, April 2015.

[301] C. Trois, M. Martinello, L. C. E. de Bona, and M. D. Del Fabro, "From Software Defined Network to Network Defined for Software," in *Proc. 30th Annual ACM Symposium on Applied Computing*, SAC '15, pp. 665–668, ACM, April 2015.

[302] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea, "Enabling End-Host Network Functions," in *Proc. ACM Conference of Special Interest Group on Data Communication*, SIGCOMM '15, pp. 493–507, ACM, August 2015.

[303] R. Cohen, K. Barabash, B. Rochwerger, L. Schour, D. Crisan, R. Birke, C. Minkenberg, M. Gusat, R. Recio, and V. Jain, "An intent-based approach for network virtualization," in *Proc. 2013 IFIP/IEEE International Symposium on Integrated Network Management*, IM '13, pp. 42–50, IEEE, May 2013.

[304] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple: Simplifying SDN Programming Using Algorithmic Policies," in *Proc. ACM Conference of the Special Interest Group on Data Communication*, SIGCOMM '13, pp. 87–98, ACM, August 2013.