CODE BASED AUTHORIZATION IN PUBLIC CLOUDS

by

Yan Zhai

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Science)

at the

UNIVERSITY OF WISCONSIN–MADISON

2018

Date of final oral examination: 11/26/2018

The dissertation is approved by the following members of the Final Oral Committee:
Aditya Akella, Professor, Computer Science
Jeffrey Chase, Professor, Computer Science
Loris D'Antoni, Assistant Professor, Computer Science
Kassem Fawaz, Assistant Professor, Electrical and Computer Engineering
Michael Swift, Professor, Computer Science

ABSTRACT

Authorization systems are the last level of protection in production environments. Authorizers, i.e. authorizing services such as a storage gateway, make access decisions based on requester-provided credentials and admin-specified policies, in order to protect the intent of asset owners.

Central to authorizations are authenticated user identities, such as a registered username or a public name from a certificate signed by a trusted authority. However, knowing only the user identity is not enough in the current cloud computing paradigm. As data sharing demands across different organizations prevail in public clouds, current authorization systems become problematic due to their coarse granularity, inflexibility and lacking of defense in depth.

This dissertation presents a new authorization framework Latte on public clouds. It integrates another equally important factor, code identity, into authorization systems. Security of a system depends ultimately on its code identity, i.e., what programs run and how they are configured. For example, an authorizer would risk less to approve access if it knows a request is sent from a program with up-to-date security patches. Using code identities in authorization can largely enhance trust between mutually distrustful organizations, and help public clouds to become healthy and collaborative ecosystems.

However, effectively accessing and leveraging trustworthy program identities in authorization is fundamentally hard. There are four factors: validity of acquired code identities, the dependencies of programs in the form of software stack and distributed components, the difficulties of interpreting and managing code identities for parties who neither produce nor deploy actual programs, and lastly the compatibility with widely used systems.

Latte addresses these challenges based on a few practical assumptions. It securely and compatibly provides code identities for authorization. With Latte, an authorizing party can flexibly

associate a target code identity with a wide array of trust anchors, such as hardened source code, secure configurations, and useful security assertions from trusted auditors or security tools. These anchors allows the authorizing party to continuously monitor and evaluate risks to use a third party service with negligible overhead.

The research includes three parts. The first is CQSTR, an IaaS platform for secure data sharing between cloud tenants. CQSTR proposes a missing abstraction, the *cloud containers*, for cloud tenants to securely share data with each other. With CQSTR, authorizers can obtain code identities of virtual machines, and prove they are properly isolated to avoid data leakage.

The second part is TapCon, an attesting container manager that provides source-based attestation and network-based authentication for containers on a trusted cloud platform incorporating new features for code attestation. TapCon allows a third party to verify that an attested container is running specific code bound securely to an identified source repository. It shows how to use attested code identity as a basis for access control, and provides a basis to trust containers from a different party.

The third part is a logical framework Latte, which incorporates generic code based authorization in clouds. Latte addresses the code dependency problem using logical attestation, and overcomes the difficulties of code identity management with source based attestation and a practical endorser model. An authorizer of Latte can accurately describe in first order logic about requirements on a requester, such as what source repository it comes from, and what specific security configurations it must have. Latte demonstrates a practical and compatible way to integrate code identities into current cloud authorization systems. It enables new use cases such as joint data mining, in which two data owners agree on a safe analytics program that protects the privacy of their inputs, and then ensure that only the designated program can access their data.

To my wife: Wenxuan, my evenstar.

ACKNOWLEDGMENTS

How time flies! It's hard to imagine it has been six years since my arrival at Madison, and now I am already standing at the last mile of this journey. The road of Ph.D is full of thorns and rocks, but it is on this harsh expedition that I met and worked with so many brilliant people. I learned a lot from them, I received their warm help, and I want to thank all of them at this special moment.

I would like to sincerely thank my advisor Professor Michael Swift first. Someone has great talent in his career, someone has deep love about his work, and Mike has both. His broad knowledge and sharp insights in computer systems and architectures have enlightened me time by time during those most difficult moments in my research. His passion in researches has inspired me to enjoy the process of solving tough problems. And he is such a great and patient advisor that, I can always find him and seek for some good advices about the confusions in my work. It was those countless times of discussions with him that finally made my way through. I learned so much from him about being a good researcher. I really appreciate every aspects that he has helped me during my Ph.D life.

I would also like to thank two major collaborators Professor Jeffrey Chase and Qiang Cao from Duke University. The vision of the software trust in this dissertation originally came from Jeff. He offered indispensible help in my research, from ideas, designs, writings to presentations. Qiang is another major contributor to TapCon and Latte projects. He offered great help in the design, implementation and evaluations. Moreover, Qiang implemented the SAFE framework, which is heavily used in Latte to evaluate logical policies.

I want to thank all the people who participated in my research. Great appreciation to Professor Thomas Ristenpart for his contributions in CQSTR and help to determine the initial direction of my research. Tom is a very smart person. He always comes with fantastic ideas. He is also a very

cool man that everybody would enjoys hanging out with for drinks, nachos and fun chats (He paid for them, thanks!). I want to thank Lichao Yin for his work in CQSTR. He helped me quite a bit in running and managing Openstack. Liang Wang and Adam Everspauph offered valuable feedbacks on designs and paper writing of all my projects. Venkatanathan Varadarajan, YanFang Le, Mark Mansi all helped proof reading my papers. Xiao Zhang hosted me during my internship at Google, where I accumulated many experiences in complex systems. I also had a good time collaborating with Professor Jason Mars and Lingjia Tang during my visit at Google on a short paper HaPPy.

My gratitude to all other members of my committe, Professor Aditya Akella, Prof Loris D'Antoni, Prof Kassem Fawaz for attending my Ph.D defense exam. Aditya was also on my committe at the time of my prelim exam. Thanks him for several excellent feedbacks.

Lastly, I want to thank my family for their support, my parents, my wife and two little members Gaga and Hulu. Particularly, I want to thank my wife Wenxuan for all she has done for me. She always holds my hands whenever I feel desperate and frustrated, in order to cheer me up. Her warm hugs, her pretty smile, and her earnest love all granted and will always grant me the strength and determiniation to go on and fight all obstacles on my way. Words are far from enough to express my heart. But never mind, she is my heart.

Again, it's my honor to ever know all of you great minds. Thank you all!

TABLE OF CONTENTS

Page

ABSTRACT i

LIST OF TABLES x

LIST OF FIGURES xi

1 Introduction 1

1.1 Motivation: Secure Data Sharing 2
1.2 Solution: Software Trust 3
1.3 Challenges 4
1.4 Latte Framework and Contributions 7
1.4.1 CQSTR: Secure IaaS Sharing Platform 8
1.4.2 TapCon: A Container Attestation Platform 9
1.4.3 Latte: A Code Based Authorization Framework 10
1.5 Dissertation Organization 12

2 Background 13

2.1 Code Trust and Authorization 13
2.2 New Implications of Code Trust 15
2.2.1 Public Cloud Providers as Partner 15
2.2.2 Containerization: Isolate Apples and Oranges 16
2.2.3 DevOps: Developer Writes Code, Code does Everything Else 18
2.2.4 Trusted Network Addresses as Secure Identifiers 21
2.3 Related Works 21
2.3.1 Industrial Efforts 22
2.3.2 Security of DevOps 23
2.3.3 Trusted Computing and Remote Attestation 24
2.3.4 Authentication and Authorization 26
2.3.5 Trusted Software 28
2.3.6 Cryptographically Secured Computation 30

Page

2.3.7 Logical Trust 31

3 CQSTR 32

3.1 Introduction 32
3.2 Background and Challenges 32
3.3 CQSTR Abstractions 35
3.3.1 Cloud Containers 36
3.3.2 State Assertions 38
3.3.3 API Restrictions 39
3.3.4 Security Analysis 40
3.4 Implementing Cloud Containers on AWS 41
3.4.1 Implementing Containers over AWS 41
3.4.2 Checking Audit Logs 43
3.5 Implementing CQSTR on OpenStack 45
3.5.1 Containers and Their Management 46
3.5.2 Attestation-based Access Control on Sharing 47
3.5.3 API Restrictions 48
3.6 Application Case Studies 49
3.6.1 Basic Pattern 49
3.6.2 SpamAssassin 50
3.6.3 PacketPig 52
3.6.4 PredictionIO 53
3.6.5 Experience 54
3.7 Performance Evaluation 55
3.7.1 Configuration 55
3.7.2 Application Performance 55
3.7.3 Microbenchmarks 56
3.8 Summary 57

4 TapCon 58

4.1 Introduction 58
4.2 Motivation and Overview 59
4.2.1 Trust Model 60
4.2.2 TapCon Service 60
4.2.3 Logical trust 62
4.3 The TapCon Layer 63
4.4 Logical Trust for TapCon 64
4.5 Prototype Evaluation 68

Page

4.6 Conclusion and Discussion 69

5 Latte 71

5.1 Introduction 71
5.2 Overview 71
5.2.1 Authorization Using Latte 71
5.2.2 Trust Model 73
5.2.3 Logical Trust in Latte 74
5.3 Attestation in Latte 74
5.3.1 Attestations and Endorsements 75
5.3.2 Validation Logic 77
5.3.3 Retrieving Metadata 78
5.4 Authorization Using Logics in a Cloud 78
5.4.1 Logical Guards 79
5.4.2 Layered Platforms and Attester Property 80
5.4.3 Network Authentication 82
5.4.4 Source-based attestation and Builder Property 83
5.4.5 Grouping for Distributed Systems 85
5.5 Latte Kubernetes 86
5.5.1 The Architecture 86
5.5.2 New Features and Guarantees 87
5.5.3 Verification 89
5.5.4 Implementation Details 90
5.5.5 Trusted Build Service 93
5.5.6 Summary of Latte Kubernetes 94
5.6 Applications of Latte 94
5.6.1 Joint Analytics on Spark 94
5.6.2 Trusted Spam Filtering Service 98
5.6.3 Other Applications 100
5.7 Implementation of Latte 100
5.7.1 Metadata Service 100
5.7.2 Latte Library 102
5.7.3 Port range management 102
5.7.4 Total Effort 102
5.8 Evaluation 103
5.8.1 Evaluation Setup 103
5.8.2 Attestation Cost 103
5.8.3 Authorization Cost 105

Page

5.8.4 Applications 106

5.9 Summary 107

6 Discussion 108

7 Lessons and Future Works 111

7.1 Lessons 111

7.2 Future Works 113

7.2.1 Secure Hardware and Multi-Clouds 113

7.2.2 Colocation and Side Channel Attacks 114

7.2.3 Network Authentication and Layered Networks 114

7.2.4 High Order Logic Trust 115

8 Conclusion 116

LIST OF REFERENCES 117

LIST OF TABLES

Table Page

3.1 Modified OpenStack components. 45

3.2 Functions for sealing in CQSTR on Openstack. 48

3.3 Application workloads and execution time. 56

4.1 Functions for sealing in TapCon. 61

4.2 Meaning of the recursive logic rules in Listing 4.1. 67

4.3 Container boot time for unmodified Docker and TapCon. 68

5.1 Simple attestation statements in Latte. Each statement asserts a fact—a logical predicate with constant parameters—attributed to an authenticated issuer. 75

5.2 Some predefined guard predicates in the Latte guard library. 77

6.1 Survey about how end users/operators set up the cloud services. There are 37 samples coming from different organizations. 109

LIST OF FIGURES

Figure Page

2.1 The software stack model of VM only and after containers are introduced. Containerization uses namespaces and overlay file systems to create isolated runtime for each individual application. Unlike the case with a shared application runtime, in which applications may be affected by many potential problems, such as shared library configuration and preloading, file system permission, and so on. the containerization makes a clean interface to reason about the application's behavior, based on the kernel interface only. 17

2.2 An abstract view of current DevOps process. The code repository comes with configuration files to direct how software should be built, tested and deployed. Build pipelines and configuration management software use such files to automatically build and orchestrate software on target platforms. The whole process can proceed without any human intervention at all. 19

3.1 The logical view of CQSTR. (1) Alice places data in bucket, and (2) publishes location of data and security properties required for access. (3) Bob retrieves information, and (4) launches service in cloud container with required restrictions. Service (5) fetches data, (6) computes over it, and (7) publishes results to allowed output server. 35

3.2 Sample manifest allowing *container2* to receive TCP packets on port 8080 with a download limit and bandwidth cap, use two images, access public volumes read-only, and write only to specified private volumes, with up to 100KB writes to a specified public object store bucket. 39

3.3 Basic Container Pattern 50

3.4 Configuration for SpamAssassin. Image name with prefix "std-" means unmodified applications and OS from official sources. Other images contain code that implements management functions. 51

3.5 Configuration for PacketPig. 52

3.6 Configuration for PredictionIO. 54

Figure Page

4.1 Layered attestation example: a trusted IaaS service attests a sealed TapCon VM, which launches and attests application containers. The shaded areas represent attested programs and their attestations stored in an IaaS metadata service. TapCon attests to the container's image, application code, and the launch script and parameters. 60

4.2 An application principal that performs attestation-based access control checks is an *authorizer*. It runs an off-the-shelf logic engine to evaluate logical guard conditions according to its local policy rules. The logical rules evaluate code attestations published at each layer, and endorsements of the attested code. . 65

4.3 Latency of queries for three types of compliance checks, measured under concurrency level C=10. Latency includes Styla prover cost, related scripting costs, and network latency. 69

5.1 Principal roles in three phases of Latte operation for a typical program running as an attested instance in the cloud. 72

5.2 Metadata assertions in Latte. Logical statements include attestations of newly launched guest instances, source certifications for images, and endorsements of properties of code objects. The metadata service indexes each statement according to its subject (e.g., an instance or code object). Each statement is attributed to a speaker—a principal that issued the statement—which may also be an instance that is the subject of other published metadata. Statements are linked to both speaker and subject. For example, "vm1" is attached to "VPC1" and uses "Image-Docker". 75

5.3 software stack view of a Spark application. An authorizer can root the trust of Spark executor process to the root of trust IaaS only when the intermediate layers are all attesters. 81

5.4 Chained attestations in an exemplary service–a Spark analytics platform with layering and composition. Each box has the attestations about an instance, labeled with their subject. A Spark worker instance at the top of the software stack is attested by a chain of attestations from lower platform layers, and statements from an attested cluster master admitting it to a cluster service group as a worker. 85

5.5 Simplified Kubernetes(K8s) architecture overview. There is at least a master VM, and multiple node VMs. On master VM the API server direclty stores resource metadata in a cluster storage(usually an etcd cluster), other daemons such like the scheduler, a node control daemon Kubelet, a node network plugin, and so on will pull resource specification from API server, and sync with actual resource object. 87

5.7 HiBench Spark Suite 106

Chapter 1

Introduction

This dissertation introduces a cloud authorization framework Latte that enables code identities as a necessary building block for authorizing tenant services. The code identity of a running program is composed by (i) a hash over its executable image or the source repository that produces such image, and (ii) the configurations of this program. Security properties of a program, such as firewall setup, isolation features and so on, ultimately depend on its code identity. Latte provides code identities in a compatible way. With Latte, an authorizing party can flexibly define its own trust anchors on a wide array of elements, such as hardened source code, secure configurations, and useful security assertions from trusted auditors or security tools. Based on these anchors, the authorizing party can continuously monitor expected properties about code identities that are associated with a third party service, so that it can have more insights about potential risks of using this service, e.g. it can check whether a remote service is patched against the latest vulnerabilities yet.

In this dissertation, I will discuss the importance of adapting code identities into authorization, and how code identities can be used to construct the basis for trustworthy collaboration between different cloud tenants. Then I will introduce how Latte and its components use code identities to improve the trustworthiness of applications of different cloud layers, including Infrastructure-as-a-Service (IaaS), Platform-as-a-Service(PaaS), and Software-as-a-Service(SaaS) clouds.

1.1 Motivation: Secure Data Sharing

Private data should always be kept safe: this is a long term goal for organizations that possess sensitive data. The safest way is to never give out the data to anybody. But in public clouds this is not the case: a cloud tenant DataProvider (DPR) often needs to share its data with a different tenant ServiceProvider (SPR) due to a variety of reasons, such as cheaper storage, faster development, or better analytics.

Consider a **motivating example** of email filtering. A tenant wants to check for spam emails. It decides to subscribe a filtering service of another tenant. The email owner is the DPR in this case, and the filtering service owner is the SPR. The same pattern is commonly seen in public clouds, e.g. a third party database addon on a PaaS cloud like Heroku [82], Machine-Learning-as-a-Service (MLaaS) [40, 56, 57, 167] which learns over DPRs' data sets, and so on. Sharing data to such external services come with risks, because the dataset (e.g., emails) can contain sensitive information. The SPR could leak the private data to other unapproved parties. For example, a service provider can misconfigure its firewall, use a weak password, or transfer clear text over internet. An attacker may successfully extract private data from compromised service provider's storage servers or networks. Similarly, operators within the service provider's company may break the code of conduct, and intentionally peek secrets of their customers. Given these risks, before a DPR can actually transfer data to the SPR, there should be an **authorizer**, i.e., a DPR's administrator or its email gateway, to evaluate how risky it is to use a remote service. But in current authorization systems, which largely rely on credentials such as digital certificates, provide too limited information to give authorizer the necessary assurance. From a typical credential such as an AWS token, an authorizer can usually only see the user identity of a requester, which is far from enough to help DPRs determine the risk conditions.

Although there are many efforts to address this visibility problem, such as requring addition compliance certificates on the SPR or applying data anonymization on the private data, they fall short in different ways, such as too complex to fully implement, or restricting the service functionality. I argue that trusting software is a better solution for such problem.

1.2 Solution: Software Trust

Software trust (or interchangeably, *code trust*) is to use security properties of the third party services as accurate terms to check why sharing data to SPR is secure, and to allow a DPR to independently respond to risk conditions. In the email filtering scenario, a concerned DPR may want to further require that (i) the filtering program is the latest version of open source program such as SpamAssassin [67], (ii) it is configured to not store any email and only store other statistic data locally without transferring them out, and (iii) only the network port for this service is opened, and no ssh is running. These security properties can better capture what are ”risky” to the DPR compared to credential only scenarios.

By **incorporating code identity of requesting services, i.e., which programs are requesting access with what configuration, as first-class entities into authorization**, an authorizer can naturally link these properties to a requesting service.

To see how it works out, I make a few practical ***assumptions*** based on several user surveys [160, 59, 52, 159, 123, 124]: first, the cloud tenants are willing to hand out their source code or binary applications to third party auditing services. Second, open source projects are playing an important role in a cloud service. A SPR may directly set up its service by customizing a long term supported official release such as a tuned MySQL database [115]. It may also use open sourced platforms such as Kubernetes [77] to orchestrate and isolate their services. Thirdly, I assume that SPRs will use public accessible tools, such as continuous integration (CI) pipeline, immutable infrastructure and security scanners to provide higher assurance, which I am going to discuss more in chapter 2. Lastly, the DPR, SPR, and the auditors and tools use external mechanisms to negotiate about how a security property should be encoded and interpreted. Besides a few Latte reserved properties (section 5.4), how to define a property is out of scope for this work.

These assumptions clarify a wide array of available origins for an authorizer to obtain security properties. For example, a SPR may already purchase an auditing service from VeraCode [170] to check its application for security risks. VeraCode can then endorse a property to certify that the hash of the inspected application runs in a locked down environment with only necessary ports

opened, that it does not store customer data locally, and that it does not initiate suspicious connections to unknown network endpoints. Similar origins include public knowledge on open source projects, e.g. the container orchestration platform Kubernetes [77] have no malicious backdoor, or assertions made by public available tools, e.g. the container scanner Clair [134] claims that a container image is not affected by latest Common Vulnerabilities and Exposures (CVEs).

An authorizer fully controls which endorsers should be trusted for their security endorsements, thus if it knows what is the code identity for a requester, it can leverage these security properties. A DPR can define in its policy that VeraCode (identified by its public certificate) is trusted to endorse such property, so that if a request is authenticated to come from a program identified by the endorsed hash, the DPR can logically infer that the running program has this endorsed property, which in turn provides assurance about that the requester will protect outsourced data securely. The mechanism allows an authorizer to timely and flexibly check a remote service for necessary security guarantees, thus can provide incompetence advantage that any previous authorization systems can not provide.

This dissertation presents an authorization framework Latte to realize the vision of software trust in public clouds. Latte incorporates code identities as the fundamental building block, and uses it to provide a more secure approach for data sharing toward cloud tenant services. Following sections will discuss why it is challenging to integrate code identities for software trust. Then I will describe how Latte approaches this problem, and each Latte project.

1.3 Challenges

The typical mechanism of integrating code identities is that a trusted program or hardware, i.e. *root of trust*, proves to a challenger that a code is running on the remote side. This is called *remote attestation*.

Acquiring code identities through attestation protocols are extensively studied in past work, ranging from initial work on trusted computing and remote attestation, to state of art hardware enclaves such as Intel SGX (§2.3.3). However, despite these efforts on remote attestation, there

are still four major challenges that prevents us from integrating code identity into authorization systems:

1. Code identity changed by insiders.

2. Code identity management for third party authorization.

3. Program layering in public clouds.

4. Compatibility with existing software.

The first challenge is called the time-of-check-time-of-use problem (***ToCToU problem***). This problem arises if the verified remote program or system gets changed in an undetected manner, i.e., the obtained code identity become invalid. In this dissertation, ToCToU problem are mainly about insiders. For example, if an administrator updates an attested program after its identity is acquired by a client, then the offered identity becomes obsolete. On an open system with dynamic input, it is very difficult to track the current state of a long running program.

The second and third challenges come from practically using code identities in authorization process.

The second challenge is the difficulty to leverage code identities in the third party authorization (**third party problem**). The term *third party authorization* means to authorize services from a different organization. A related concept is the *first party authorization*, which happens for authorizing services from the same party. The key difficulty for the third party authorization is that it is hard for authorizers of a third party to derive security properties from binary code identities, i.e., small digests over program texts and configuration files, such as a signed certificate or even a binary hash.

For example, consider that an authorizer needs to use a customized Spark service [28]. In the first party scenario, the service is built from a branch of Spark's source repository and deployed by the same organization. In this case, a shared data store for intra-organization usage can record the mapping between a program hash and its source repository as well as the building environment. Using a binary identity, an authorizer can still look it up in the program records to learn that the

interacting service is Spark. Then it can either directly check up the Spark source for security properties, or even rely on the public knowledge about the source, since Spark is actively maintained and used by a diverse group of people.

On the contrary, a third party usually does not have access to the same piece of information. Using just the binary identities, an authorizer can neither analyze the Spark service, nor can it conclude anything about its source repository. A partial solution is to provide a signature signed by the builder alongside the binary identities. If the builder is trusted, then what the builder certifies, such as the source repository or security properties of the service are logically trustworthy, too. However, for collaborating scenarios, it is meaningless if the SPR builds its own service. As a result, it is difficult for a third party authorizer to derive security properties of a remote service from its binary identity. What is worse, existing attestation mechanisms usually only provide binary code identities.

A slightly improved mechanism called *property based attestation* relies on a trusted party to derive security properties from binary attestation result, e.g. an auditor can sign a certificate for an inspected service image to endorse that it has been locked down. However, this suffers from the same problems of compliance checking: how can an authorizer define policies customized for their own need, and how can an endorsement from the translator reach the authorizer in time?

The third challenge is the ***layering problem***: an application is often heavily affected by its runtime environment. Suppose a SPR runs a Spark job to pull data from DPR's. The requesting process can run in a Spark container launched by Docker, which runs in a VM hosted by Amazon Web Service. These underlying layers usually have direct control over the hosted program. In the Spark job example, the Spark container is set up by the Docker daemon, which means one usually has to trust the Docker daemon in order to trust the Spark runtime, such as its security parameters. Moreover, the job is also distributed to multiple VMs. To trust the code identity of the Spark job, one has to authenticate identities of the environment and distributed components.

The question is: how can the DPR verify the validity of the code identity of the Spark job, with layered dependencies on other software whose code identity also needs validation?

A simple answer to this question is to eliminate the dependencies on software stack with secure enclaves. Secure enclaves, e.g. Intel SGX, are attested and protected by secure hardware directly, so that even an administrator of the whole software stack can not tamper with a running enclave. Such capability erases the need to depend on a whole software stack for enclave applications.

However, enclaves are not the silver bullet. For example, researchers have shown that enclave applications can still suffer from control channel attacks like Iago attack [49], in which a malicious host tricks the enclave application to misuse system calls and leak private data. Side channel attacks such as foreshadow [45] also allows a co-located program to steal information from SGX enclaves with shared CPU cache. Moreover, how to uniformly authorize both enclave application and regular applications is also unanswered. For authorizers, it is more reasonable to also obtain valid code identity of the hosting environment, e.g. what operating system and security protections are. Knowing this allows they to confirm the remote kernel is a standard Linux kernel without malicious system calls, and the system has disabled remote logins, e.g. ssh or vnc, to prevent administrators to login and reboot to another kernel. To the best of my knowledge, there is not yet an authorization system that deals with this layering problem in an elastic and multi-tenant environment like public clouds.

The last challenge is the complexity and compatibility with existing software. To make the work practical, the software I have studied in this thesis are all industrial quality software such as Openstack [125], and Kubernetes, and etc, each of which comprises a large code bases with several million lines of code. Introducing code based authorization to their credential only systems incur big engineering difficulties. At the same time, interoperability of software remains a big challenge for any general designs, i.e. different software tend to have application specific authorization protocols and dialects, making it hard to design a uniform code based authorization framework for general software without breaking their existing protocols and semantics.

1.4 Latte Framework and Contributions

This dissertation presents a new cloud authorization framework *Latte* that fully addresses above challenges of incorporating code identities in authorization and realizes the vision of software trust

in cloud ecosystems. In this section I will first introduce two cornerstone systems for building Latte. Then I will describe the Latte framework and how it works. For each of these pieces, i.e., the cornerstone systems and the framework, I will illustrate its outcome with the motivating example.

1.4.1 CQSTR: Secure IaaS Sharing Platform

The first system I built is CQSTR[1]. CQSTR is an IaaS-extension for managing *contained* execution, in which a group of tenant instances (VMs) have their external connectivity restricted according to a declared policy as a defense against information leakage. CQSTR introduces a new *cloud container* abstraction. Cloud container extends the notion of a container on a local operating system [4] to cloud-scale applications consisting of many VM instances and other resources. A cloud container policy specifies immutable confinement properties that limit network and storage access for computations in the container. In addition, the policy can define a set of images that are allowable to boot VM instances into the cloud container. The IaaS provider issues attestations of these properties to clients of the service. For example, CQSTR can attest that a service runs from known disk images containing a locked-down operating system and a trusted application framework, and that network access is limited. Cloud containers provide a data owner with a verifiable set of controls over outsourced private data. A client can verify that a service meets its required security policies to protect data from leakage and misuse. Security policy is designed and specified separately from the application and is enforced by the IaaS platform, which allows CQSTR to run existing operating systems and applications without modification. Further, CQSTR provides a foundation for more flexible control based on security mechanisms in the attested VM images, such as privilege separation, least privilege or defense-in-depth policies.

I implemented CQSTR on both Openstack and AWS. The main contribution of CQSTR is to provide trusted executions for collaborations between cloud tenants. In the motivating example, the SPR can run its filtering service fully contained in a cloud container, which is guaranteed by IaaS provider to only communicate with restricted network endpoints, i.e., the DPR's servers,

[1] Read as *sequester*: verb, meaning “to isolate or hide away”.

and to remain immutable during its execution. A trusted cloud provider proves to authorizers about this containment condition so that they can verify the private data won't be leaked. The incentives to trust in cloud providers are later detailed in §2.2.1. Besides, CQSTR is the basis to address layering challenges above by delivering VM code identities to authorizers. Latte uses the VM identities to construct logical attestation chain (described below) for authorizers to examine. CQSTR also addresses the ToCToU problem at IaaS level and maintains backward compatibility with tenant applications.

1.4.2 TapCon: A Container Attestation Platform

I then propose a logical attestation approach for application platforms anchored in a trusted IaaS cloud, and show how to use it for end-to-end scenarios that incorporate attested code identity into an access control scheme. To illustrate, I describe TapCon[2], an attesting container manager built on Docker [60]. TapCon publishes *source-based attestations* about the code identity of container instances that it launches. TapCon allocates IaaS IP address to each container to take advantage of the IaaS provider's managed network to block spoofed packets, so that network addresses of containers can be used directly as authenticated instance identifiers that bind a container securely to its attested code.

Main contributions of TapCon are that it largely improves resource usage of CQSTR and that it eases trusted application development. These are naturally achieved with containerizations, which is an important technical trend for enabling software trust (§2.2.2).

For technical challenges, TapCon addresses the ToCToU problem at container level, so that a TapCon container is guaranteed to be *sealed* so that it has the same identity during its lifetime. In addition, TapCon's usage of source based attestation is critical to the third party problem. In the motivating example, assuming the filtering service is open sourced on a popular website such as Github [73], then a DPR can download the source and verify desired security properties. Even if the code is proprietary, TapCon's logic structure can easily allow a mutually agreed party auditor to check the code separately. Unlike the auditors who check compliance rules regarding both code

[2]TapCon is Trusted Attestation Platform for Containers, or a fastener that anchors a structure to a solid foundation.

and people, these auditors only focus on properties of source repositories. DPRs can flexibly plug in endorsements from any of these trusted auditors to verify the trustworthiness of a TapCon container. Lastly, TapCon shows how to use logics to address the layering problem with two layers of identities, i.e., container identities and VM identities.

1.4.3 Latte: A Code Based Authorization Framework

Generalizing from CQSTR and TapCon, I present an authorization framework called *Latte* that realizes this vision of software trust in public clouds. Latte extends the metadata service in TapCon to store statements and uses the trusted network environment of CQSTR for remote authentication.

Central to Latte are the program instances running on a cloud platform, which are abbreviated as *instances*. For example, an instance can be a hosted VM or container launched from an VM/container image. Instances interact through a network controlled by the cloud provider, and may act as clients or servers for other instances or external entities. Latte identifies the code and configuration of a running instance, and provides authorization mechanisms to reason about its trustworthiness. For example, a DPR can specify an access control policy that allows access from instances running qualified software stacks with proper configuration (e.g., locked down).

Latte defines a basic vocabulary and set of tools for services to issue authenticated assertions (*attestation statements*) about various objects—e.g., hosted instances, program objects (identified by hash), and configuration templates—together with logical policy rules to derive checkable security predicates from chains of related assertions. Cloud systems can use these tools to expose metadata that is useful for trust and visibility. These platforms can attest to properties of instances (i.e., about running code), while software build and verification tools can endorse program security properties (i.e., about source code repositories or binary images).

With Latte, I show how cloud applications and services can define and invoke *logical guards* that validate assertions about instances requesting access, and reason about them to infer high-level security predicates needed for compliance with a logical policy. Similarly, a client can examine a service's attestation statements and endorsements and decide whether to trust the service by validating that it was launched by a trusted platform and its code was endorsed by a trusted entity.

These features enable *attestation-based authorization*, in which a service's access policy can consider attestations of code identity—together with endorsements of that code—for the instances requesting access.

Latte's use of logic allows us to address all the above challenges to practically use code identities in cloud systems. First, Latte's logical structure naturally supports *chain attestations* to validate a full stack of software. Latte generalizes the notion of network authentication developed in CQSTR and TapCon, and allows authorizers to trust not only IaaS network, but also application managed network, such as cluster network of Kubernetes [95]. Second, Latte's logic enables a *grouping* mechanism that allows authorizers to verify that all members of a distributed service meet trust requirements. These two mechanisms address the layering challenge. Third, usage of network authentication again maintains backward compatibility of existing systems. Lastly, Latte enables transitive trust from a source repository to produced image by a *trusted build service*. This brings generalized source-based attestation mechanism into cloud environment to any program instead of just containers, which largely overcome the third party challenge.

The main contributions of Latte are to implement a generalized code based authorization framework, and to provide a trusted platform to use the code trust and builds trustworthy applications easily. With Latte, authorizers can define accurate and timely authorization policies with customized security requirements. Such capability is crucial for constructing a collaborative ecosystem with high level security and automation, because Latte does not rely on an intermediate party. As for the motivating example above, a DPR can quickly reject any requests coming from a SPR service, if the service is not applied with the latest security patch. The decision making no longer waits for the much slower output of an auditor. Similarly, an authorizer can tightly control what operations can be done on a private data set with Latte. It does so by giving access only to programs with specific properties, e.g., a program that does not talk to public network. I implement a trusted platform based on Kubernetes. The Kubernetes cluster delivers end to end application examples with production quality software such as Spark to illustrate how to build secure analytics and extend them for other scenarios. Experiments show that Latte adds only minimal overhead.

To the best of my knowledge, Latte is the first practical framework that uniformly enables source code identities at different layers of public clouds, including IaaS, PaaS, SaaS.

1.5 Dissertation Organization

Chapter 2 describes the research background. There were discussion about incorporating code identity into authorization [154, 156] before, and there is also a rich set of research work that aims to build trusted systems based on remote attestation. This dissertation will discuss the new implications introduced by modern software engineering.

Chapter 3, 4, 5 describe the design, implementation and evaluation of CQSTR, TapCon, and Latte, respectively.

Chapter 7 forms a discussion about lessons learned from the research. Correspondingly, I present future directions in chapter 7.2.

Chapter 8 is the conclusion of the whole framework and code trust.

Chapter 2

Background

2.1 Code Trust and Authorization

As a harbor for various applications and services, today's cloud ecosystem has reached an enormous level of complexity that is embodied not only in its scale, but also in the diversity of system virtualization, service composition and deployment in the cloud. For example, an analytics service might use a synthesis of a layered software stack, extending from VMs provided by Amazon Web Service, to containers from a Docker service, with computing workers launched from a framework such as Spark. Service administrators face increasing challenges in handling authorization in such a sophisticated environment. Answers to authorizations are often deeply buried in the code running at the requester's software stack and in the way that the requester service is composed. For example, an administrator may need to know whether a requester is sandboxed, using Google's native client [183], Linux Containers, or Linux Security Modules; whether the requester is behind a firewall; or whether the requester is part of a distributed service and other members of the service are running similarly trusted code and configured to be secure. These use cases from sandboxed processes, containers, firewalls, and distributed services are examples that start characterizing the real-world need of authorization in the cloud.

Current authorization mechanisms unfortunately fall short, as authorization is often based on knowledge of a cryptographic key. A crypto key identifies its holder, which in turn allows evaluation against policies that describe what actions are allowed from the identity. However, due to complex layering and composition that become dominating the ecosystem, a meaningful authorization context is no longer comprised of a single identifier, e.g., a crypto key. Instead, applications

and services are isolated in different ways by nature, depending on the levels of virtualization (e.g., VM and container) they use and the means (e.g., master-worker, pub-sub, and replication) they were composed. When a new request arrives at an authorizer, the authorizer must be able to identify a set of items, containing relevant information about properties of a service software stack and all of the service's components.

These items are then closely associated with the code identity of a program. If we trust a program P to have certain properties, then we can infer that a running instance of P also has those properties. But how to be sure a cloud instance is running a trusted program P? In general, it is possible only if one controls the infrastructure P runs on, the build chain that produces an executable image for P, and any other software that is present at runtime to interpret or assist it—e.g., an operating system and runtime libraries.

In a cloud setting, clients of a service know little about any of the links in this trust chain, because the service is hosted by another party (e.g. an IaaS provider) and often managed by yet another party, e.g., a SaaS service provider, who may be a tenant of the IaaS service. Today it is common to rely instead on *social trust* in the service provider to maintain the advertised properties.

Software identity can be a stronger basis for trust when clients have knowledge of the software. For example, clients may have trust in installations of untampered open-source IaaS implementations such as OpenStack, container management services such as Docker, and data analytics platforms such as Spark. These systems are in wide use and their source code is open to public inspection. As noted by the Nexus OS authors, trust in code may also come from analysis, synthesis, or fiat [156]. Program analysis can verify security properties of source code [93, 39, 80, 79], and trusted code generators may assure various properties of generated code [83]. Software trust could also derive from "fiat" endorsement by a trusted authority, such as a curator of open-source code, or a private auditor for proprietary code.

When software is trusted, it can provide a stronger basis for trust in services running in the cloud through *attestations of code identity*—authenticated statements by trusted parties about the code that runs in a service instance. These attestations may be combined with endorsements of

the code or its security properties to infer trust in an instance. In particular, I focus on third-party authorization, in which a third party—other than the owner of a tenant service or of the host machine—consumes the attestations and endorsements for authorization. For example, a customer can trust a cloud-hosted service if it trusts the program that the service runs and the platform that hosts it.

2.2 New Implications of Code Trust

As I discussed in section 1.3, there were quite a few challenges to enable code trust in current systems. However, as software development becomes more mature in recent years, there are several technical trends that change the ground, and make it a right time to rethink about using code trust in software.

2.2.1 Public Cloud Providers as Partner

No system has ever achieved the same level of sharing and collaboration as public clouds. These large distributed infrastructures offer a common ground for application developers to run services with shared resources. Consequently, each cloud provider now represents a unique role on its own infrastructure, where it tightly controls the resources offered to tenants, e.g., VMs, storage, network. The cloud provider also knows necessary metadata related to security postures of tenant services. Without special notation I refer cloud providers specifically as IaaS providers. Such unique presence offers an appealing foothold for integrating software trust on public clouds. This is because cloud providers can act as the root of trusts and use such metadata and control mechanism to conclude whether a tenant service satisfies a given authorization policy. For example, a trusted cloud provider knows what VM image is requested to launch a tenant VM, then it will not boot the VM with any other image. If the image is further known to be a closed box that seals a spam filtering service, then it is much less risky for clients to use this service, compared to a pure black box with only a provable user/organizational identity.

That being said, there are many concerns about cloud providers being the powerful adversaries which control top privileges on their computing platforms. For example, if a cloud operator goes

rogue, then he can impose threats on tenants' digital assets, such as directly peeking of secrets from memory of a tenant virtual machine through hypervisor calls. However, I believe the risk from the cloud provider is much lower than regular external attackers or malicious tenants. In fact, trusts in cloud providers are usually well grounded in their good reputations and the fact that they are all heavily audited and monitored for multiple compliance standards, so that any violations are reported and will be punished [12]. In addition, to make the shared computing platform more competitive, a cloud provider is usually highly motivated in helping tenants to improve their security. For example, on the AWS cloud, tenants can launch services in a strongly isolated network (virtual private network), protect them via a dedicated API gateway, and authorize accesses to resources such as the storage objects with IAM policies [16]. Using these security features it is easier than that in pre-cloud era to set up secure services.

My premise in this paper is that *a cloud platform can act as a root of trust*, a partner to authenticate instances and expose metadata about them for access control and policy compliance, including attesting to software identity and security configuration, which authorizers can leverage to assess the potential risk of data leakage, and make more reasonable decisions. Meanwhile, the mechanism in Latte does not conflict with integration of secure hardware such as SGX to further reduce the risks. I will discuss the future work about SGX and multi-clouds in chapter 7.2.

2.2.2 Containerization: Isolate Apples and Oranges

An important technical trend in recent years is the fast adaption of OS containers. A container is a resource unit built on top of kernel abstractions to isolate applications. For example, a common Linux container is usually implemented with namespaces and control groups [140]. Namespaces can enforce processes in different containers to see different resources, e.g., network devices, filesystems, process IDs, user IDs, and so on. Control groups isolate the performance of containers, such as the CPU share, the memory budget, the IO bandwidth and etc. A container can also be a kernel builtin object, such as a Solaris Zone [166]. Popular container implementations such as Docker [60], Rkt [139], and so on have gained a lot of attention, and are proved quite successful so far.

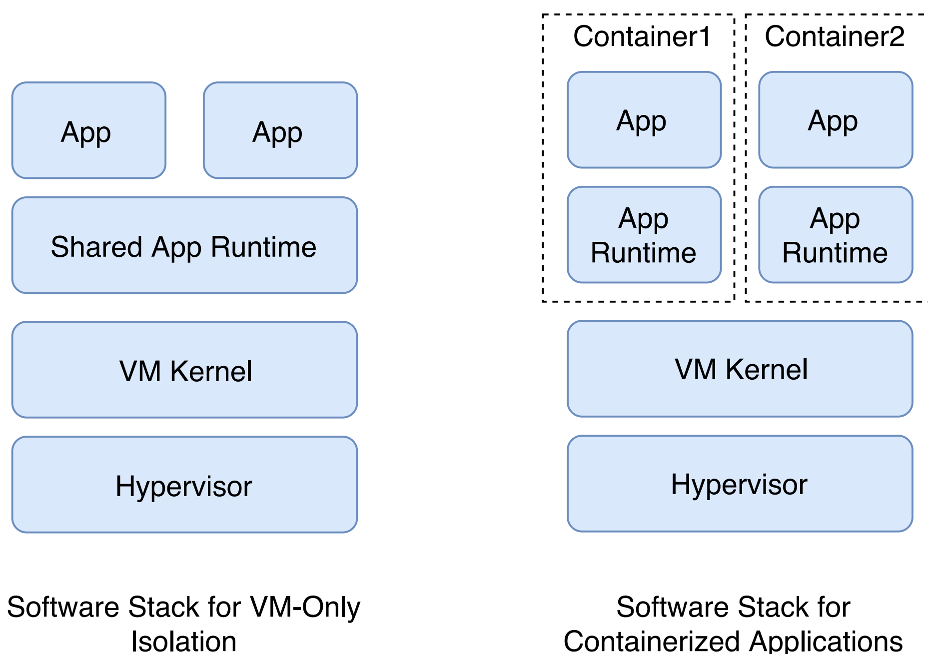


Figure 2.1: The software stack model of VM only and after containers are introduced. Containerization uses namespaces and overlay file systems to create isolated runtime for each individual application. Unlike the case with a shared application runtime, in which applications may be affected by many potential problems, such as shared library configuration and preloading, file system permission, and so on. the containerization makes a clean interface to reason about the application's behavior, based on the kernel interface only.

The emergence of containers forms one of the indispensable bolts that help fastening the software trust. The biggest impact from containerization is that it provides a clean isolation on both functionality and control for applications. As shown in figure 2.1, the sharing model of containers typically only shares the OS kernel and isolates the file system view and many other resources for each application container. This makes it much easier to reason about the functionlity of an application, as other applications can only interact with the target application through explicit kernel interfaces. It is much harder to do so without containers. For example, system wide shared libraries can affect multiple independent applications. Upgrading a library can sometimes break multiple other services. Similarly, improperly shared file systems may put secrets of several other applications at risk if one application with the same access right is compromised. These factors make it very hard to decouple the application to check from other applications. What is more, the concept of container can be easily generalized at different levels, e.g. a Spark job is often a set of

distributed Java processes with exclusive resources, which can be also treated as containers. It is more realistic to deal with code identities of these containerized applications. Lastly, containers play the key role in current DevOps model, which is the next technical trend I am about to discuss.

Benefiting from the fast movement on containerization, this dissertation will discuss the constructs on the state of art container platforms, Docker [60] and Kubernetes [77], to integrate code trust, and show how trust on containerized applications can be significantly improved.

2.2.3 DevOps: Developer Writes Code, Code does Everything Else

The philosophy of software engineering has significantly changed from what it was in the past few years. Current practices in software development has also benefited from containerization a lot. The easy-packing and reproducibility nature of containers have largely automated software development. Source code is built, tested, and deployed in fully managed pipeline according to specifications, which are also a part of the source code. Such automation in software is called DevOps practices. Latte uses DevOps practices extensively in its design, so that it can tightly control and verify the trustworthiness of components in the software development process. My design not only faithfully binds running program to their source repository, but also allows incorporation of expertise and public knowledge for a more accurate and agile authorization process. This distinguishes Latte from most of other related systems. In this subsection I will briefly cover the DevOps model and how it is related to the code trust vision that Latte beholds.

A simplified DevOps model is shown in Figure 2.2. In such model, developers also act as operators by instructing a dedicated build pipeline and configuration management software how to deliver and deploy a service instance. For example, on AWS, a developer can include a configuration file in her git repository deployed on Github [73] or AWS CodeCommit [10]. Every time she commits a change to the repository, AWS CodePipeline [11] will be notified by Git hooks, and automatically start actions described by that configuration file. The first step is usually building the source code, where the configuration file specifies build environment for constructing a well structured deployable image for consumption of configuration management tool such as Chef [50], Ansible [27], or Puppet [131].

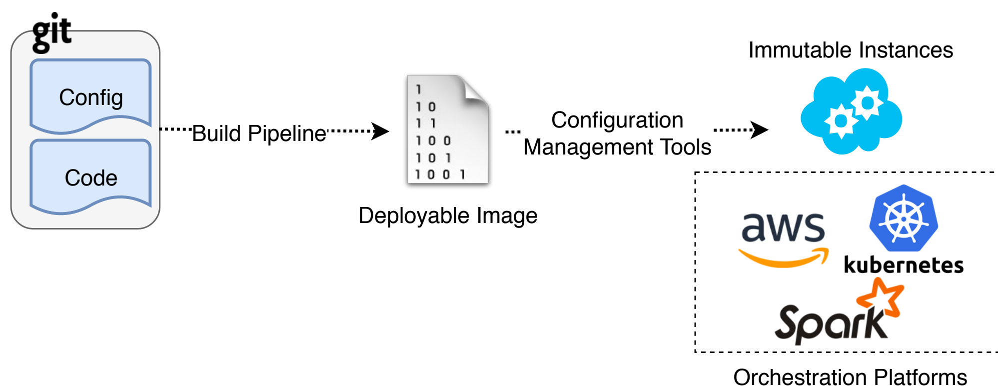


Figure 2.2: An abstract view of current DevOps process. The code repository comes with configuration files to direct how software should be built, tested and deployed. Build pipelines and configuration management software use such files to automatically build and orchestrate software on target platforms. The whole process can proceed without any human intervention at all.

The full automation of DevOps implies the possibility of integrating all these information into the paradigm of software trust. Should an authorizer be able to identify and trust the building facilities for a certain application executable, then the authorizer can also entrust the build time metadata from these facilities, such as what source repositories are used to build this executable. In fact, there are many trustworthy building tools and environments, such as Gcc for native binary [70], Apache Maven for Java [107], Docker for container image, and so on. The question then is how to reveal the code identity about the building facility of a requesting software instance. Latte formally addresses this in §5.4.4 and implements an exemplary build service in §5.5.5. The build service attests software images to its source repository. Whenever an instance is launched with one of such images, an authorizer can also reason about its builder, i.e., this build service. By trusting this build service, the source identity of an instance is available to authorizers. In other words, the automation of DevOps establishes a software only picture, and Latte uses it as a basis to deliver the source code identity of a running program.

Unlike binary identities, source code identities usually preserve original intentions of the programs that anyone with access to the source code can check for desired security properties, making them a key element for addressing the third party challenges (section 1.3). In section 5.5 I will show how to build and deploy a Kubernetes [77] service from its source repository. Anyone can

learn that a Kubernetes VM only runs a Kubernetes service, compiled from official Kubernetes source code, standard compiling tools. Moreover, from the build script in the source repository, it is easy to show that VM has been locked down to prevent further changes such as ssh from public network. The trust anchors in this example are all reasonable and easily justified: that official Kubernetes service won't try to compromise its users' secrets, that IaaS will launch VM faithfully, and that the build service such as AWS CodePipeline will correctly follow the build script in source repository and build the Kubernetes image. In contrast, drawing similar conclusions with only binary identities would be way harder, because it loses the information about how each file is produced. For example, proving that a random Kubernetes daemon does not contain any backdoors would be a tough challenge for binary analysis.

Another practices accompany full automation is the usage of immutable infrastructures or immutable instances [13]. The central idea of immutable infrastructure practices is pretty simple: configurations and executable instances of applications should be immutable during their life time, and new changes are committed to source repositories to trigger the DevOps pipeline for new deployments. Old deployments remain running until new ones become healthy. For example, the AWS CodeDeploy service supports a "blue-green" model for deploying new services. When the service code or configurations are changed, commit hooks in the repository will invoke the CodeDeploy to deliver green (new) VMs while keeping blue (old) VMs running. At a certain stage the traffics to blue VMs will be redirected to green VMs once they become fully booted.

The benefits of immutable instances are obvious for code trust, since their configurations are frozen at launch and all changes are applied by restarting the instances with new configurations. This provides a solid starting point to further adapt *sealed instances*. The difference between sealed instance and immutable instance is that the mandatory immutability of the former. In the trusted systems that I ported to Latte, such as Openstack, Docker, Kubernetes, and etc, they include mandatory access control policies to enforce that management operations can not tamper with attested programs. Making programs as sealed instances prevents the ToCToU problem, as the program identity would remain unchanged until its death.

2.2.4 Trusted Network Addresses as Secure Identifiers

Any code attestation system must provide some way to authenticate communications with the attested instance, so that insecure software cannot masquerade as a secure instance. Hardware-level attestation—and many software attestation systems—bind a software identity to a keypair whose private key is held by the attested service. This approach introduces various overheads and practical challenges to manage keypairs safely, and may require substantial changes to software to sign and/or encrypt communications.

As an alternative, authenticating network addresses do not have such troubles. For example, it is not a problem to leak a network address, which is originally public accessible. Trusting code enables us to leverage the managed network of the trusted code to block any spoofed packets. For example, the premise of a trusted cloud provider enables us to use the network addresses as strong identities within the cloud network, because the cloud provider uniquely assigns IP addresses to VMs and will drop packets with spoofed addresses [14, 188]. Similarly, if all nodes of a VM cluster are verified to be running a trusted Kubernetes service and proper network configuration, then it is less risky for a container hosted by Kubernetes to trust its cluster network, because it can be configured to run crypto protected tunnel transparently. On a secure network, any communications from an assigned network address are known to have originated from the instance that owns the source address. This property enables us to use network addresses as secure identifiers for instance endpoints [26, 96].

2.3 Related Works

This section mainly discusses the related works about software trust in the past. The purpose of this section is mainly to understand how past efforts are dedicated to achieve the goal of trusting software.

2.3.1 Industrial Efforts

While the holy grail for data protection is to be absolutely secure so that data won't leak under any circumstance, what people do practically today are termed as *data anonymization*, *social trust*, and *compliance based trust*. However, these approaches all have their problems.

Data anonymization. Data anonymization means a DPR tries to remove sensitive information from a dataset by erasing their values or replacing them with random ones before sharing the data with a SPR. But this can hit hard on the SPR service sometimes. For example, if the sender addresses are replaced with random names before pushing emails to a spam filter service, then the service immediately loses an important heuristic for scoring spams. Meanwhile, it is not always possible to anonymize a data set securely. For example, network trace anonymization is a challenging problem, because it is hard to preserve the necessary network properties such as the topology, while not exposing privacy of the enterprise network [137]. In contrast, social trust and compliance based trust are more general to use.

Social Trust. Social trust is one of the most widely used approaches so far. An authorizer approves actions purely based on the reputation of the requester (or a remote service to use). For example, Gmail [74] users can trust Gmail for properly handling their emails without leaking information out, since the company is known to have strong security protections, strict code of conduct, etc. But social trust is a subjective form of trust, because there is no definitive rules to sort out which organization is more trustworthy. The result is that such trust can be blind sometimes. For instance, the incident of Ashley Madison data breach was a result of such blind belief that the website would properly protect their personal information, whereas the website didn't [35]. As I discussed before, the authorizer has no control nor visibility over what can happen on an outsourced dataset at all. The requirement of good reputation can also become a problem for a healthy cloud ecosystems, where thousands of companies who haven't established a fame that can earns them trusts. Code trust on the other hand is more objective, as it accurately defines what code and what configuration should be trusted, regardless of the identity of the service owner.

Compliance based trust. In addition to social trust, compliance standards are published by standardized organization or government authorities to rule the actions for computation providers. Qualified auditors will go to the provider's place to observe, to interview and to test the execution of the standards. For example, the PCI-DSS standard [129] is published by payment card authorities such as Visa [171], Mastercard [106] and so on. It aims to ensure users' credit card information is handled securely at the target environment. There will be many human involved factors, as well as black or white box testing to examine if there is any violation. Other common compliance standards include HIPAA [65] for medical information protection, GDPR [172] for general data privacy for individual. In general compliance processes are to provide a trustworthy environment for execution, but they do not concern much about what application is actually up to. Besides, compliance standards can be vague, and rely on interpretation from the auditors, so that it may not accurately reflect what is really needed for authorization purpose. Lastly, implementing compliance standards can impose heavy burden on service provider. Like social reputation, a small but innovative company with only a few employees may have neither budget nor incentive to implement all compliance rules. I believe code trust is a good alternative for these companies, who may use Latte on a complaint cloud to build up services that offer stronger and more accurate guarantee than compliance standards.

2.3.2 Security of DevOps

There are many tools and works that try to secure the DevOps process. The ecosystem built around containerization is pretty active. Docker provides a notary [61] service to validate container image layers come from a trusted developer. There are also a wide range of security tools scans container image for potential vulnerabilities [134, 135, 86]. PCHECK [179] analyzes source code to detect latent errors early before they actually happen. Rehearsal [152] validates the correctness of Puppet configuration scripts for correct orchestration. SELinux [158] and Apparmor [31] are common mandatory access control on application privileges. Seccomp [148] secures system calls on Linux. Cilium [53], calico [88] and weave-net [177] implement label based network policies

on Kubernetes. HashiCorp Vault [78] provides credential management and tracking for containers. Sysdig Falco [87] audits behaviors of containers and networking in Kubernetes.

These works represent different software elements in DevOps process. Latte's usage of logics can easily integrate them as building blocks to enhance security. For example, an authorizer can use Clair [134] to check for critical CVEs for a remote container for assurance of code quality.

2.3.3 Trusted Computing and Remote Attestation

There are quite a few of remote attestation mechanisms (e.g., TPM, TXT) that enable trusted execution based on a hardware root of trust [108, 143, 69, 157, 142, 153]. Latte is similar to these in that it attests code identity and allows clients to verify that a service is running the proper version of code in a proper configuration. On the other hand, these works usually focus on single host attestation, instead of considering applications in public cloud scenario.

Past works mainly focus on binary code identities, which are usually hashes over executable binaries and configuration values [143, 69]. Terra [69] supports layered attestation rooted in hardware where each software layer (hypervisor, operating system, etc.) has a unique encryption key at each level for remote communication. In contrast, Latte applies layered attestation in a trusted cloud setting, using network addresses as secure identifiers and avoid the need for cryptographic protocols. Azure provides shielded VMs for stronger protection against cloud operators [111] and relies on TPMs and attestation to confirm the integrity of VMs, but only as a binary image. BIND [153] performs fine-grained attestation on binary code regions. Both these approaches, by only attesting to binaries, will have problems about managing code identities for multiple users (Section 1.3). Latte can attest to the source repository of a running service, which greatly simplifies this problem.

Some systems advocate property-based attestation [51, 142], where a trusted third party attest to properties of the entire running platform, instead of a single program hash. Similarly, Santos et al build a trusted-computing-based runtime that seals data from malicious cloud operators based on instance properties [144]. These systems do not address the code layering (Section 1.3), which means they can only consider properties of the whole platform, instead of reasoning about more

finer grained and complicated relationships within the platform, such as co-located containers. Latte allows authorizers to reason about the software stack as well as distributed components. It further extends software identity to source code, and supports third-party authorizers, making it suitable to protect inter-tenant interactions in the cloud.

Attestations provide a basis for access control to services and data. Singularity [154] and Nexus [156] explore attestation-based access control in an operating system on a single host. Latte's use of logical trust is similar to logical attestation in Nexus [156], but extends logical attestation to a cloud setting with distributed applications, and is based on standard Datalog logic. Like Latte, OpenStack Congress [133] allows verifying policy compliance using Datalog rules. However, Congress targets compliance checks for system policies using system-generated metadata. Latte extends this idea to higher level and incorporates software identity.

To further protect trusted software, Intel's SGX extension creates hardware-isolated environments (enclaves) for trusted code and private data [25]. SGX has inspired many new designs [37, 84, 147, 33, 36]. VC3 [147] and Haven [37] use SGX to ensure private data is not released to untrusted components (i.e., the programming framework, hypervisor or OS). SCONE [33] runs Docker container in SGX. Ryoan [84] incorporates SGX enclave based DAG computation with information flow control to protect data privacy. Asylo provides a framework to integrate enclave attestations into application policies, e.g. a storage ACL [36] The general assumption is that user data are threaten by a powerful adversary who controls the entire software stack, i.e. the cloud provider [155, 145]. Latte takes a different assumption that IaaS cloud providers could actually be a root of trust when reasoning about trustworthy of a tenant service [9]. I believe that this trust assumption is reasonable and matches the practical realities of cloud computing today; the goal of my work is to leverage this trust to bootstrap richer security controls for safe cross-tenant data sharing in large-scale cloud services [71], and not just in individual computers. In principle, future advances in trusted execution technology could extend the trust chain down to the hardware by attesting the trusted cloud stack itself.

2.3.4 Authentication and Authorization

There are rich literatures describing authentication and authorization for computer systems.

Authentication is the process for a requester to prove the ownership of a principal. A principal is a name and associated metadata that a human or a service registers at an identity provider service. The central problem for authentication is that how a requester R proves that it controls a principal to another another entity S, such as a remote service of another organization, or the identity provider of R.

Proving the ownership of R to its identity provider is straightforward, since the provider controls necessary information when R is registered. This process is usually *credential based*: the owner of R shares a secret such as a password to bind with R during registration, so that the identity provider assumes whoever knows this secret is the owner of R. For example, to log into a Linux system as a user, a login process acts as the identity provider, who looks for user identity in file */etc/shadow*. A human being can provide the correct username and password combination to prove she controls the user account. The login process then acknowledges and creates a new shell for her.

On the other hand, it is more complicated when S is not the identity provider, because S does not know the secret of R as well. In this case, the first step of authentication is for a requester to authenticate to the identity provider in charge of R, which will issue a temporary credential to endorse about the ownership of R. This credential expires in a short time, e.g. an hour, and it does not need to reveal anything about the permanent credentials, i.e., the registration time secret, of R. Kerberos [118] is one of the early examples which performs a two step authentication. The authentication server signs an authenticated requester a temporary grant ticket, which can later be exchanged for a service ticket to prove the identity at another service in the same network domain for access. To authenticate to Ss across organizations, protocols such as OpenID Connect (OIDC) [122] and Security Assertion Markup Language (SAML) [120] are used to exchange credentials.

The key difference for Latte's authentication is that Latte entrusts the cloud network to provide unique address binding for the program instances as discussed in §2.2.4.

Authorization is the process of controlling an authenticated identity for certain actions. It often appears in forms of access controls. Authorizers define policies about *under what conditions should who do what*. When an authenticated request arrives, authorization systems retrieves related policies and evaluate against the request. Access is granted if all related policies are satisfied.

Classified by the policy type, the most commonly used authorization methods are access control lists (ACLs), role based access control (RBAC), or attribute based access control (ABAC). For example, Linux filesystem usually manipulates a 10-bit ACL to define access rights for the owner, the owner group and other users; AWS offer IAM mechanism as a mix of RBAC and ABAC to control access based on users, groups, roles and some attributes of the requester such as source IP address [15]; Kubernetes also supports ABAC mechanism based on the "label" attribute of each container to enforce network policies network access policy cilium [53]. Authorization in Latte is a special type of ABAC that leverages code identities of cloud programs and related security properties as authorization attributes. To further manipulate hardware issued code identities, Asylo [36] provides a similar framework to specifically attest secure hardware enclaves.

Classified by the principal to control, authorization can either appear as discretionary access control (DAC), e.g. Linux file system permissions, which authorizes the current user, or mandatory access control, e.g. SELinux [158] and AppArmor [31], which authorizes the real initiator of a request. Latte authorization can be either DAC or MAC, as the attestation chains in Latte present both information of the requester and the real initiator.

Most authorizations are centralized, i.e., using a central source of information to evaluate policies. Appel et al proposed a decentralized framework called proof-carrying authorization[1] (PCA) [32]. The PCA framework uses high order logic to check proofs along with web user scenarios. Latte implements code based authorization with first order logic, and focuses on trusting software. Macaroons [41] develops a decentralized authorization mechanism in public clouds. The caveats embedded in a macaroon token can be used to confine when, where and who can use the credential for access. Such restriction is similar to the defense in depth associated with Latte, such that credentials can be fixed for a specific code setting. Meanwhile, Latte uses centralized interface

[1] Although the publication title is about authentication, the authors later clarified it should be authorization.

to store and fetch code based information. I will discuss more about decentralized storage for Latte metadata in chapter 7.2.

2.3.5 Trusted Software

There are a great many of works that actual build up trusted systems that can be integrated into Latte. What Latte does is to provide a secure way of using code identities of these systems as well as application specific metadata that reflects actual security settings. Latte defines a set of standards for porting multi-tenant platforms (§5.4.2); and it requires no changes to adapt end applications into the framework. A ported application should have its code identity available inside Latte's metadata storage, where authorizers can define policies to fetch and check its trustworthiness through its identity and any associated metadata. These trusted systems can be classified as below:

Closed box A large number of works seek to improve control over sensitive data. One solution for the data owners to run analysis code only in instances (VMs) under their direct control. Mittal et al. [113] describe such an approach for packet trace analysis. It precludes instances that run third-party proprietary code or are managed by any party other than the data owner; Latte allows both.

Limited Interface Another approach is to build limited interfaces to code that is trusted not to leak data. Bunker [112] creates a closed-box for the specific case of network trace analysis on a single node. Airavat [141] ensures privacy-preserving outputs for MapReduce workloads by confining untrusted mappers and running only trusted reducers. PINQ [109] implements differential privacy mechanisms based on C# LINQ [101], and provides a restricted programming language to compute on sensitive information. These works are complementary to Latte: one could for example set up the trusted code (e.g., Bunker, Airavat, or PINQ) as an attestable program in Latte, and obtain the benefits of both systems. Here the role of Latte is to attest to clients of the service that it runs the correct (trusted) code and is properly locked down. Brown and Chase [44] proposed a similar idea for a PaaS service that attests to the code running in an instance.

Information flow control CQSTR project (§3) provides an abstraction that can be viewed as static form of mandatory access control using Information Flow Control, including Decentralized IFC (DIFC [114]). IFC systems track labels on data (a program's inputs "taint" outputs they may have affected) and confine untrusted code by blocking unsafe data flows to channels of lower clearances. DIFC systems also provide a natural means to grant specific authority to trusted code, e.g., to declassify its output data by changing the label. Examples include Airavat and OS-level systems including Histar [186], Flume [94], and Asbestos [63]. DIFC has also been implemented for a distributed environment (e.g., [187]). While DIFC is powerful in concept, it faces many challenges to integrate with existing software. For example, the label propagation and management usually require dedicated network protocols, which is not easy with existing code. OConnor et al implemented PivotWall using on SDN network [121]. But how it should work with overlapped network in public clouds is not clear. Latte's usage of code identities allow an integrated DIFC system to be verified to work correctly by any authorizers. By trusting a remote system to be a DIFC system, an authorizer may not need to rely on its hosted program instance for being trustworthy. For the motivating example of email filtering, the filtering service owner can launch the service inside a cloud container that is only allowed to communicate with the email owner. In this way, there is no need to know about what code filtering service is running, since it won't be able to leak data through the network anyway.

Application Sandbox Application level sandboxes can effectively isolate untrusted software so that they can not cause damage [183, 66, 98]. They are another basis for software trust. Both Nexus [156] and Ryoan [84] used application sandboxes. Again, these applications can be integrated into Latte, so the authorizers can learn about how a remote application is restricted, e.g. on the Native Client [183] it can confine what system call can be called by the application.

Verification and Model Checking Formally verified software, such as IronClad [80], IronFleet [79], Verdi [178], Sel4 [93], VCC [54], ExpressOS [104], and HyperKernel [117] provide another basis for trust in systems. Alternatively, sandboxes like native clients [183], Vx32 [66],

Minibox [98] restrict the behavior of a black box binary to preempt undesired behavior. Amazon is known to apply formal verification in work [119]. TLS library s2n has been verified for its HMAC implementation, and aims to provide ground for security reasoning [149]. There are also many other works that can help to secure the system like model checking [92, 99, 180], specification reasoning [119, 185]. As noted above, these approaches are complementary to Latte, as they provide additional information that can be used in attestation statements for reasoning about what code to trust. For example, if an application can be verified to have no network access in its code, then it is a strong guarantee that this application won't leak private data through network channels.

Secure systems for operators One of the realistic concerns toward current systems are malicious insiders. According to IBM's annual security report, insider attacks can be a great threat for industry like financial services, where 58% of the attacks in year 2016 came from malicious or unaware insiders [136]. WatchIT [151] designs a new system to restrict privileged administrators to help customers with balance between security and usability. Similarly, both Azure and Google Cloud Platform provide shielded VMs for stronger protection against rogue operators [111, 85].

2.3.6 Cryptographically Secured Computation

A different way to securely collaborate can be achieved with cryptography constructs. Secure multiparty computation (MPC) enables multiple mutual distrustful parties to collaborate without disclosing each other's secret [100]. However, it does not apply to scenarios when user data has to be outsourced like in my motivating example. In addition, the performance can be very slow. A recent work reports the speed of 9 million AND gates per second on AWS computing instances [176], which is several orders slower than normal computation. Similarly, verifiable computation proves the correctness of remote computations [173, 150, 43, 174], but it requires the verifier to know about the computation, which sometimes can be proprietary.

Fully homomorphic encryption (FHE) operates encrypted data without decrypting them [72, 168]. However, the performance is still the concerning factor. For example, Dai et al reports 7.3 second processing per AES block [55], Gai et al needs about 10 minutes to encrypt 10MB

data [68]. Recently the first commercial FHE application is launched, and allows 4KB biometric vector of patients to be secured by one-way FHE in reasonable amount of time [130]. There are still more challenges to make the scheme work with scenarios similar to my motivating example, where Tera-byte or even Peta-byte level data can be expected.

2.3.7 Logical Trust

Logic serves as the foundation of Latte's attestation architecture. Logic rules capture precisely how to validate chained attestations and combine them with endorsements and other assertions for rigorous and verifiable security checks. The vocabulary is easily extensible for a wide range of attestations, endorsements, and user-defined compound policies, decoupled from the Latte implementation. Extensions to the vocabulary require no change to the Latte framework itself: only the logic templates and matching policy rules need must change.

Latte uses ordinary safe Datalog [47] for authorization policies. To my knowledge Latte is the first use of Datalog for attestation and the first use of logic for cloud attestation. Previous uses of Datalog as a trust language include Binder [58], SD3 [91], RT [97] and SeNDLOG [8]. Nexus [156, 146] introduces logical attestation based on a more expressive logic; I contend that Datalog offers sufficient power and is fast enough for practical use.

Chapter 3

CQSTR

3.1 Introduction

An increasing fraction of computing services run on managed infrastructure-as-a-service (IaaS) systems such as Amazon AWS, Google GCE, or Microsoft Azure [81, 175]. They form an ecosystem of inter-connected tenant services run on a common virtualized platform, sometimes sharing data [71]. Despite a common underlying platform, *safe cross-tenant sharing* generally requires blind trust: clients with sensitive data must trust their partners to protect it, just like that in the motivating email filtering example.

This chapter introduces the CQSTR as an IaaS extension. It first introduces the additional background about using IaaS for secure data sharing (§3.2). It then describes the design of cloud container (§3.3), and two implementations on Openstack (§3.5) and AWS (§3.4). Following the implementations are three comphrensive example applications that can be used with CQSTR (§3.6). After that I will evaluate the CQSTR with micro benchmarks and application workload (§3.7), and briefly summarize this chapter (§3.8).

3.2 Background and Challenges

My work focuses on infrastructure-as-a-service (IaaS) clouds that provide customers with virtual machine instances and other resources. Customers specify a disk image stored in the cloud, and the IaaS cloud provider selects a physical machine and boots a virtual machine from the chosen image. IaaS systems also provide customers with virtual disks, blob storage and other services whose use are mediated by the cloud provider.

Cloud applications are increasingly constructed as a mix of IaaS-provided services and third-party services provided by other tenants running on the same IaaS platform. The goal of my work is to improve trust between components of such systems. To set context, I outline an exemplary scenario for which current solutions prove unsatisfactory.

An example problem: secure data analytics. Suppose a cloud tenant Alice has shopping-cart data, and would like to use an analytics service run by another cloud tenant Bob to make shopping recommendations from the data. However, the data has competitive value, so she would like to make sure that Bob cannot use the data for anything but generating recommendations.

The ideal workflow for this scenario would be:

1. Alice pushes her data to Bob's service every day.
2. Bob's service trains a prediction model based on the data each day, which might require tens or even hundreds of VM instances.
3. The prediction service deploys the model to a web service, to which Alice submits requests for shopping predictions. The web service may itself consist of many instances and tiers.

Alice's security requirement is that her history data and queries for prediction are kept secret from Bob and any third parties. Bob may have proprietary algorithms running in his service, and will not expose his code to Alice, so Alice cannot run the prediction service herself.

Note that the scenario above arises in practice already: companies like Google, Amazon, and BigML offer such machine-learning services and in many cases keep their algorithms secret [76, 19]. This example represents a *client/server* setting, for which containment is sufficient: Alice, the client, provides data to Bob's contained computation service, and the output is delivered only to Alice. In a *delegated* setting, the output of the service may instead go to Bob or an unrelated third party; in essence Alice delegates control over the data to the service, so some trust in the service code is required. For example, a hospital may allow researchers to use the results of a data analysis. Beyond machine learning, many other applications fall into this setting of cross-tenant

sharing between a data provider (Alice) and a compute provider (Bob). More examples are in Section 3.6.

These scenarios all prompt the key question faced in my work: ***How can a trusted IaaS provider safely assure Alice of Bob's secure handling of her data?***

On using existing IaaS features. Leading IaaS clouds such as Amazon Web Services (AWS) are feature-rich, and one might expect that such cross-tenant sharing scenarios can be handled with existing functionality. In particular, AWS offers virtual private clouds (VPCs) [20], which support a logically isolated network for a set of instances, and AWS Identity and Access Management (IAM) provides role-based access control over data and services. Bob could launch his service within a VPC, and Alice could grant Bob via IAM the permission to fetch her customer data (e.g., from an S3 bucket) after checking the service properties for compliance with her policy.

However, there is no agreed-upon mechanism in current IaaS clouds for services to expose security information to clients. And, there are numerous individual services that must be separately managed to ensure security. Section 3.4 discusses the challenges of implementing cloud containers on AWS, and Section 2.3 has discussed other approaches, such as cryptographic multiparty computation, information flow control, or trusted hardware.

Threat model. As I discussed in new implications in section 2.2.1, I view the IaaS provider as trusted. This is a reasonable presumption for users of cloud systems today. I therefore consider threats such as insider attacks by IaaS employees, cross-VM isolation boundary violations [138, 189, 169, 182, 102], or compromise of the IaaS control plane (e.g., exploits against hypervisors) to be out of scope. These threats are important, but are not addressed by CQSTR.

Instead, I focus primarily on threats arising in applications running on top of the IaaS platform. I consider for simplicity settings with two tenants, e.g., Alice and Bob in my example above. I consider situations in which either Alice or Bob behaves maliciously, because of (accidental) misconfigurations or insider attacks, e.g., by application developers. I also consider remote attackers: one goal is to attest to Alice that Bob is using best practices for defense.

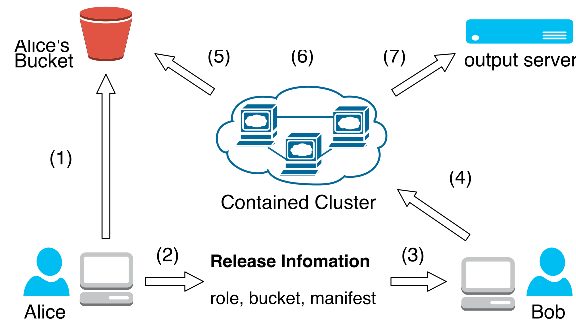


Figure 3.1: The logical view of CQSTR. (1) Alice places data in bucket, and (2) publishes location of data and security properties required for access. (3) Bob retrieves information, and (4) launches service in cloud container with required restrictions. Service (5) fetches data, (6) computes over it, and (7) publishes results to allowed output server.

3.3 CQSTR Abstractions

My central hypothesis is that carefully chosen, but relatively modest, extensions to IaaS platforms can provide a foundation for constructing trustworthy cloud-hosted services. To this end, I describe the design of CQSTR, which extends IaaS cloud management platforms to enable tenants to share data safely and assert more control over use of their data.

Overview. CQSTR comprises three central elements:

1. *Cloud containers* are groupings of VM instances and are the units of security policy management.

2. *State assertions* are authenticated statements from the cloud provider about the security properties of a service's container.

3. *API restrictions* ensure that IaaS management APIs cannot be abused to violate containment policies.

Cloud containers group the VM instances and related resources comprising a service and allow attesting to their security properties collectively. State assertions provide clients a mechanism to verify the configuration of a service, such as the isolation policies of its container or the set of

VM images used to boot instances in the service (i.e., a form of remote attestation). Finally, API restrictions limit the use of provider APIs that might compromise a container's asserted security properties. CQSTR leverages the IaaS provider's trusted network for secure cross-tenant communication; IaaS providers already control their networks to prevent spoofing and snooping by their tenants [15].

In client/server settings, the service is launched into a cloud container. A client consumes state assertions to verify that the container's security properties meet its policy requirements for protection against data leakage. In delegated settings, the client can pass the service provider a description of the required security properties, and then verify that the service meets those properties before handing it data. A data owner can also place security conditions on stored data (e.g., storage buckets or objects) to limit access to only those containers whose security properties comply with the conditions. Figure 3.1 illustrates the use of the system.

3.3.1 Cloud Containers

The key new abstraction provided by CQSTR is the cloud container, which is a set of VM instances sharing access to a private network and cloud resources, with a common set of networking, storage, and boot image restrictions. Thus, a cloud container logically extends operating system containers (e.g., Linux containers [5]) to a cluster. Abstractly, a container is described with a *manifest* that specifies a set of *security properties* detailing explicitly what instances in the container can do; anything not listed is prevented. Launching an instance into a cloud container restricts what code it can run (described below), to whom it can communicate, and where it can store and retrieve data.

Cloud containers build on existing virtual private network functionality already available on AWS/EC2 and OpenStack to limit the set of reachable hosts and to enforce firewall rules. A cloud container policy may also enforce traffic controls, such as limiting the ingress or egress bandwidth of a container, and the total amount of data that can be transferred.

A service may consist of one or more cloud containers. When a service has multiple instance types internally, such as master and worker in Hadoop, cloud containers can further secure a system

by placing each class of VM in a separate cloud container that has its own set of boot images and accessible network/storage. I term this a *cloud container group*.

Storage containment. Storage protection is enforced by extending the mechanisms to control the resources accessible to a cloud container with existing IaaS services. In addition to access control lists on storage objects, CQSTR adds sets of reachable objects to a cloud container description, providing complete control over the storage and other objects accessible to a service. CQSTR allows controlling both access to existing storage objects as well as what new storage objects, such as disk images or blob storage buckets, can be created from a container.

Trusted images. In delegated scenarios, a client may require trust in the code executing in the service before releasing information to it. For example, a hospital may require that a data-mining service uses a known and endorsed software image that enforces an information disclosure policy. A container manifest can therefore specify a set of *trusted images* eligible to run in the container. In this scenario, the client must trust the container's images to behave correctly, either through direct knowledge or endorsement by a trusted third party. All other images are prevented from being used to boot VMs within the container.

Cloud container API. A cloud container is created with a *manifest* (see Figure 3.2 for an example) that specifies the security properties of the container, or by creating a default container and explicitly setting the properties. CQSTR provides APIs to create, read, update, and delete cloud containers. Once created, the owner of the container can launch VM instances into the container from trusted images. However, once an instance has been launched into a container, the container properties are *frozen*, meaning they can no longer be modified and clients are assured they will remain in force.

3.3.2 State Assertions

A client of a service using CQSTR can make access control or usage decisions based on the security properties of the service's cloud container. There are two mechanisms for verifying assertions over a cloud container's state. First, a client can directly query the provider for assertions about a service, such as how the network is configured or what storage objects are accessible. Second, a client can place access control lists on storage objects that specify the desired properties of services accessing those objects. I describe in Sections 3.4 and 3.5 how to do this with small extensions to existing IaaS capabilities.

Properties covered by state assertions are subject to two conditions. First, a property must be *knowable* to the IaaS provider: it is easily observed by the cloud management plane. For example, CQSTR cannot assert properties that would require introspection within VM images or inspection of network traffic data contents. Second, the property must be *stable*, meaning that once it is true it remains true. As an example, "has an instance stored more than 1 MB of data?" is a stable property, whereas the amount of data stored is not stable, as it changes whenever an instance writes more data. The reason for targeting stable properties is to avoid TOCTOU bugs [42] and for efficiency: clients can cache the results of most assertions.

Immutable configurations. Many useful properties already satisfy both conditions. One is "what image did an instance boot from?" The provider knows this easily, since the image was specified in the request to the provider, and it is stable. But other useful security properties are not stable. In AWS and OpenStack, configuration properties of an instance, such as the network security group, can be modified by the owner at any time. If a data owner granted access based on one configuration (e.g., when the security settings were compliant) but the configuration changed later (on purpose or accidentally), then data could be released.

CQSTR addresses this concern by restricting the set of management operations allowable on cloud containers once they are frozen. Specifically, operations cannot change the outcome of any state assertion. For example, an owner cannot reconfigure the network to allow more access, or add more trusted images. I call the properties of a cloud container an *immutable configuration*.

```
network:
  container: container2,
  direction: ingress,
  protocol: tcp,
  local_port: 8080
traffic:
  container: container2,
  out_quota: 100MB,
  bandwidth: 100KB/sec
volume:
  ro: public,
  rw: container
objectstore:
  bucket_name: dataprovider.hadoop
     visibility: public,
     write_quota: 100KB
images:
  hadoop_master, hadoop_slave
```

Figure 3.2: Sample manifest allowing *container2* to receive TCP packets on port 8080 with a download limit and bandwidth cap, use two images, access public volumes read-only, and write only to specified private volumes, with up to 100KB writes to a specified public object store bucket.

Manifest templates. It may be expensive to check that the manifest for a container meets all required restrictions. CQSTR therefore supports *manifest templates*, which are parameterized manifests. A tenant specifies the template and its parameters when launching an instance. For example, a common policy is to allow network traffic from a single client to a single port; the manifest template specifies complete isolation of network and storage except for this one port, with the client's IP address as the port parameter.

To check that a cloud container complies with a given parameterized template, a client first checks that the template matches the expected template, and that the parameter matches the expected parameter; this is much simpler than checking that an arbitrary manifest enforces a superset of required restrictions.

3.3.3 API Restrictions

The final element of CQSTR is a set of restrictions on the allowed management operations. As noted above, some operations are prohibited to ensure that configurations are immutable. In addition, IaaS APIs may leak information from a running instance, or to modify its behavior. API restrictions on a cloud container prohibit use of APIs that violate containment, such as creating a VPN tunnel or, more insidiously, writing to a log monitored by the IaaS provider, such as Amazon's CloudWatch service [18]: log entries could expose private data outside the cloud container.

API restrictions limit what the owner of a container can do. Any action that copies or transfers storage must apply the same controls on the data after copy or transfer. Thus, snapshots are given

the same ACL as the source virtual disk. This also applies to booting a VM instance: an image generated within a cloud container cannot be used to boot instances outside the container.

These restrictions cannot fully prevent covert channels. For example, a service could intentionally vary its resource usage to encode information through fine-grained billing statements, or through monitoring APIs like AWS CloudTrail that record every provider API call [17]. In my implementation I strive to limit high-bandwidth covert channels.

3.3.4 Security Analysis

I briefly discuss some of the security benefits of CQSTR. I assume that an attacker cannot subvert the cloud provider's code, and hence cannot subvert network and storage restrictions. In addition, an attacker cannot spoof packets within the cloud network. I address three threat cases: malicious computation owners, malicious code, and remote attackers.

Malicious owners. A service owner may attempt to force trusted code to leak data. This can be done by SSH'ing into the VM, inspecting system logs, or using cloud APIs. Here, firewall rules can restrict the ability of the owner to access instances within the cloud container, and API restrictions prevent bulk channels such as logs or volume snapshots. Finally, a client may choose to trust only images with locked-down configurations that prohibit remote console access, or images endorsed by trusted third parties.

Malicious code. The worst case of attack is when VM instances run malicious code. Here, cloud containers can prevent leaking through explicit network and storage channels, but cannot prevent all covert channels through the IaaS API, as in the billing and logging examples.

In a client/server setting, output is accessible only to the client. In a delegated setting, though, output may go elsewhere. For service-generated data, such as the program output, logging, or performance metrics, traffic control on cloud containers mitigates the risk by limiting the output size and bandwidth. I show in Section 3.6 how an application-specific proxy can be used to further check or sanitize output.

Remote attackers. For third-party attackers, the primary defense is the network isolation and firewall rules. These network defenses can largely prohibit public access to instances within a cloud container.

3.4 Implementing Cloud Containers on AWS

To explore the potential for secure cross-tenant sharing with current IaaS cloud APIs, I implemented elements of the cloud container abstraction as a library above Amazon AWS. AWS provides key building blocks for the cloud container abstraction. Tenants may group instances in a virtual private cloud (VPC), and use VPC primitives to restrict network connectivity and access to some types of storage objects (e.g., S3 buckets). Role-based access control (through AWS IAM) can also limit access to storage objects. However, AWS only provides partial support to control access to all resources from a VPC, and it cannot freeze a VPC to prevent configuration changes. For these reasons, the AWS cloud container library uses *auditing* of VPC configurations and data access.

The audit-based implementation for AWS monitors compliance with a declarative policy description for a cloud container. It serves as a case study of how to use AWS security mechanisms for cross-tenant cooperation. However, the reliance on auditing security logs necessarily implies that the library cannot *prevent* non-compliant operations, but can only *detect* abuses afterwards. In addition, the library has shortcomings stemming from the limited control over logging granularity in today's AWS API. Most significantly, the logs reveal unrelated information about the service owner's account to the client, and collusion with a third account can open hidden channels for data leakage, as described below.

3.4.1 Implementing Containers over AWS

VPCs form the basis of a cloud container implementation on AWS: CQSTR launches the instances comprising a container within a VPC, which controls network reachability and firewall rules. AWS does not yet support traffic shaping, so there are no bandwidth limits or traffic caps.

To restrict access to storage, the VPC specifies in advance any S3 buckets that the VPC writes to, so a client can check compliance with its policy. The service owner configures a *VPC endpoint* [24] that specifies a set of pre-existing S3 buckets accessible from the VPC; the endpoint denies write access to all other S3 buckets. Contained services cannot create new buckets.

When a client wants to grant a service access to its stored data, additional VPC setup steps are necessary. The service owner creates a VPC, and passes its ID to the client. The client creates a new role with access to its data, restricts the role for use only by the VPC, grants the service owner access to the role, and passes the role's name (ARN) to the service owner. The service owner can then configure instances to assume the role when accessing the data.

For other storage resources, such as EBS volumes or message queues, AWS does not provide mechanisms to restrict a VPC to access only a predefined set of objects: instances within the VPC can reach a resource if any party grants the VPC access to the resource. Restricting destination IP addresses is not enough, as AWS services may share a front-end server. As I discuss below, CQSTR uses auditing to verify that all resource accesses from the VPC are compliant with the policy. Similarly, AWS does not have a mechanism to limit the set of images bootable within a VPC, so CQSTR checks audit logs to verify that all instances in a VPC were booted with compliant images.

State Assertions. A client-side library checks compliance with security properties using two mechanisms. First, the client can contact the *config service* [22] to determine the properties of a VPC and its service; the service owner must grant clients access for this. In advance of contacting a service, clients query the configuration service for relevant security properties and verify that they meet requirements (e.g., restricted networking).

Clients must also verify that instances within the VPC do not transmit data to unapproved objects, such as EBS volumes or SQS message queues. However, the config service does not provide configurations of all objects, such as volume snapshots, S3 buckets, and SQS queues. Therefore, clients must retrieve audit logs for these resources to verify that past operations did not violate container security policies.

API restrictions. Cloud containers require immutable security properties that prevent TOCTOU races. However, AWS does not provide mechanisms to prevent management operations from changing the configuration of a VPC. Thus, it is not possible to freeze a configuration to make its properties immutable. As with storage access and attestations above, CQSTR on AWS relies on auditing as a substitute for the freeze operation: clients of a service must use the library to scan audit logs both before and after using the service to verify that the configuration never violated the client's desired policy. For example, the client must verify that VPC configurations do not change.

3.4.2 Checking Audit Logs

AWS provides powerful features for auditing VPCs and their configurations. The CloudTrail security event service [17] can log operations within a tenant account on an ongoing basis. I observed that the latency between an event and a record showing up in CloudTrail was about 5 minutes. While I were writing this paper, Amazon updated CloudWatch [18] with sufficient functionality to implement auditing at lower latency. As CloudWatch audits are functionally equivalent to CloudTrail, I describe my implementation in terms of CloudTrail.

I found that even when complete audit records are available for AWS objects, such as VPCs, the security logs do not always have complete information. For example, some objects have default settings that are not provided in CloudTrail logs, such as routing tables for VPCs. In these cases, I augmented log processing with calls to the AWS configuration service and use configuration histories to retrieve the complete settings.

My audit log processing tool is about 700 lines of Python code and monitors 24 types of AWS objects.

Auditing handshake. Normally, audit logs are available to the account running a service. To allow clients of a service to check compliance with CQSTR security properties, the clients must be able to access the logs. Setting up a monitored cross-tenant VPC takes a number of steps, as, the service owner must make log events accessible to clients. The client creates a log bucket, creates a role, grants the role access to the log bucket, and grants the service owner's account access to the

role. To make the log tamper-evident, the client creates an SSE-KMS key and grants the service owner's account permission to encrypt CloudTrail logs with the key. It passes the names (ARNs) of the bucket, role, and key to the service owner. The service owner enables CloudTrail logging on its account using the received ARNs.

Limitations of AWS auditing. I found that the AWS support for security logging is not well-matched to the needs of secure cross-tenant sharing. Generally, CloudTrail logging is designed to enable an account owner to monitor operations taken by identities associated with the account, rather than all operations associated with a given group of instances (e.g., a VPC). While a VPC is bound to account, its instances can issue operations under roles from a different account. Operations issued under a role are logged to the role owner and not the VPC owner. If the service uses a role from a third account other than the client or service owner, the audit logs go to that third account and are not visible to the client.

CloudTrail logs also compromise privacy when used for cross-tenant monitoring. The security log includes *all events* in the service owner's account, and not just the events pertaining to a particular VPC of interest. If the service owner regards its logs to be confidential, then this confidentiality is violated. CloudWatch events can, in theory, filter out unrelated events involving non-target VPCs. However, in my experiments I found that filters must be updated when objects are created or deleted. In an asynchronous setting such as AWS, this can lead to lost audit records. Moreover, with multiple clients, events from one client are exposed to all others. One alternative is to use a third-party trusted auditor to check a service's compliance with the policies of its clients; ideally the IaaS provider would play this role. Furthermore, audit logs are a high-bandwidth covert channel and can allow a service to export sensitive data out of the VPC.

These limitations could be resolved by providing auditing on the granularity of an instance or VPC, rather than per account. In any case, the integrity of the auditing approach depends on comprehensive logging and monitoring of *all* data channels out of a VPC. As the default it so allow access to unaudited resources, security depends on extending the audit tool for each new resource introduced by AWS.

Components	Functionality	Modification
Nova	Compute Service	Container manager, metadata service and API Restrictions
KeyStone	Authentication Service	Attestation-based access control
Neutron	Network Service	Network reachability and traffic control
Swift	Object Storage	Storage containment
Cinder	Volume Storage	Storage containment

Table 3.1: Modified OpenStack components.

More generally, I conclude that adequate control of cross-tenant sharing requires specific support within the cloud provider API. Furthermore, it requires continued logging *after* using a service to ensure there are not data subsequent breaches. While security logging and auditing can be a basis for secure cross-tenant sharing, my premise is that an authorization model is a stronger basis for secure sharing. I therefore turn to an implementation of CQSTR on OpenStack.

3.5 Implementing CQSTR on OpenStack

I implemented CQSTR as an extension to OpenStack. Unlike the auditing based approach that is possible with current IaaS APIs, my OpenStack CQSTR implementation provides immutable cloud containers that allow only those resource accesses that are enumerated explicitly as compliant with a policy. This provides stronger guarantees.

I implemented CQSTR as an extension to OpenStack Kilo (2015.1) [125]. In total, I added 15 new files with 3,096 lines of code and modified 2,386 lines of code across 52 files. For comparison, the original components I modify comprise 455,247 lines of Python code. The added API restrictions comprise less than 500 lines.

OpenStack overview. OpenStack provides a complete set of software to implement a public or private cloud. It is widely used both within enterprises and as the basis for several public clouds including Rackspace and DreamHost. Table 3.1 describes the key components I modify.

Recent versions of OpenStack provide a *trusted computing pool* built on TPMs and an attestation server [6]. The server verifies measurements made by the TPMs, and allows clients to query

the attestations for specific machines. This provides assurance about the code booted on a single machine, but not about the network configuration or use of management APIs. Furthermore, the goal of trusted computing pools is to verify trust in the hypervisor, which is complementary to my focus.

My implementation comprises several components:

- *Container Manager*: stores container information and implements container APIs.
- *Network Agent*: enforces network control in *Neutron*.
- *Storage Agent*: isolates object namespace on *Swift*.
- *Metadata Service*: provide state assertions on *Nova*.

The crux of my implementation is to show how small modifications and extensions to existing services can provide a much richer security platform.

3.5.1 Containers and Their Management

CQSTR implements the container manager as an extension to the Nova compute service. All container and container group management APIs (Create, Read, Update, Delete) are provided as HTTP-based Nova APIs. Currently I require that a container be a member of exactly one container group. Containers are mutable until the first instance is launched into it; at that point they are frozen and become immutable. CQSTR supports dynamically adding and removing resources to/from a container, which can be important for allowing the container to scale with the computation dynamically.

For debugging purposes, CQSTR also provides a *debug* mode that allows configuration changes after instance launch. This allows an owner to extract debug logs or adjust security properties while instances are running. I found this to be critical when setting up containers. Debug mode can only be enabled before a container is frozen, though, and prevents the use of state assertions. When a container is frozen, running instances are terminated, as their configurations may have changed. Instances can then be (re-)launched.

Network isolation is built on OpenStack's tenant network. Each container group is put in a shared layer-2 network, and cloud containers have their own subnets. All these networks are fully managed by IaaS infrastructure, and cannot be modified administratively. I built traffic control in *Neutron*'s gateway to limit the aggregate bytes transferred using IPTable's *quota* module. I have not yet implemented bandwidth limits.

CQSTR provides containers with private volume storage using Cinder. Volumes that are used inside a cloud container are marked as container-local unless they are explicitly allowed to escape as listed in the manifest. Local volumes and their copies/snapshots cannot be used outside the container.

The Swift object store lacks a few features of AWS S3, such as VPC endpoints to restrict the set of accessible buckets. To control storage access, I built a storage agent inside the Swift server that implements a private namespace for cloud containers. Object accesses are directed to this namespace unless the bucket name is explicitly listed on the container manifest. The namespace is derived from the tenant account ID and the container ID. This design allows containers in a container group to share objects through Swift without exposing them outside the container. I use this feature in Prediction IO (see Section 3.6).

3.5.2 Attestation-based Access Control on Sharing

I extend the OpenStack *Metadata Service* to provide state assertions about cloud containers. A tenant using a cloud container can explicitly grant access to designated services for specific metadata properties. A client can contact the metadata service with a container ID or its IP address for state assertions about a service before using it.

CQSTR also introduces attestation-based access control, which allows embedding state assertions on resources such as Swift buckets: only services meeting required assertions can access the resource. Here, the resource owner (e.g., client of a service) creates a role, grants it the desired access, and specifies the required state assertions, which can be either a container ID or a manifest template and parameter. Upon receiving a request, the resource server (e.g., Swift) consults the metadata service for the information about both the requesting instance and its cloud container.

API and Options	Description
`Rebuild`	Disable rebuilding a VM with new image and configuration
`Console`	Disable consoles, e.g. VNC, RDP, serial and etc
`AddVif`	Disable plug in additional network interface
`InjectFile`	Disable VM launch option to inject file directly
`TenantNetwork`	Restrict what private network to use
`SecurityGroup`	Restrict what security group to use
`Snapshot`	Restrict where instance snapshot can be launched

Table 3.2: Functions for sealing in CQSTR on Openstack.

Only if the requester meets the security requirements is access granted. I modified OpenStack to support dynamic creation and sharing of such roles (similar to what AWS's IAM mechanism already provides).

These requests to the metadata service for state assertions provide an IP address for the service. As IaaS networks prevent spoofing, the IP address identifies the client of a resource server. To ensure uniqueness, requests to a resource server must use a globally unique IP address. These addresses can float between instances, so CQSTR imposes a configurable waiting period on reusing these addresses so that assertions can be cached.

A tenant can implement their own storage service that uses CQSTR services. I implemented a library for external services to request and verify container manifests, so that any tenant-implemented service can enforce the same types of policies as platform services.

3.5.3 API Restrictions

I implement API restrictions by adding hooks in existing OpenStack services to change behavior when cloud containers are in use (Table 3.2). I modified storage APIs to implement mandatory access controls (i.e., private volumes or storage objects cannot be made public) by setting initial ACLs on objects or volumes created by a container to prevent access from outside the container. OpenStack allows configuring a VPN with an "allowed address pair" that bypasses network security, so I disable the APIs for that feature on instances in cloud containers.

I also patched many management APIs to prevent side channels. As an example, I prevent snapshots on instances in containers, prevent moving IP addresses between instances, and prohibit most calls to management APIs from within a container. I strove to limit covert channels that would enable malicious instances within a container to violate bandwidth limitations. A few such covert channels still exist, such as the previously mentioned billing example.

3.6 Application Case Studies

I have applied CQSTR to three real applications: SpamAssassin [67], a spam filter service; PacketPig [128], a network trace analyzer; and PredictionIO [30], a machine learning service. For each application, I describe how I configure the container to maximally restrict the service from releasing data. In this section, an operator is the one who sets up the target containers, and a client is the one who receives the output data from the service. Note that a client may or may not be the data owner, and a client can also be an operator in some application scenarios.

3.6.1 Basic Pattern

The basic pattern for deploying a service with CQSTR is to identify the inputs, outputs, and storage requirements, and instantiate a cloud container that restricts networking and storage to just what is required. Two of the applications I investigated follow a simple pattern I call "single output, local storage" meaning that they produce output to a single endpoint, and require local storage for intermediate results. The manifest template for this pattern is shown Figure 3.3. Note in this context, the word "public" means external to the container, rather than public to the world.

This template enables the service to receive connections by network hosts parameter1 on TCP port parameter2, which can be another container or a specific address, and to read and write private local storage and read public read-only data from outside the container. This template does not restrict which images can be used to boot instances: the output and input are the same endpoint, so there is no need to trust the service to make delegated access control decisions. These restrictions are a limited form of mandatory access control over the data passed to the target cloud

container. Once a contained service is launched, the service owner cannot change these controls, so data inside will always be contained as the manifest describes.

```
network:
  public: <parameter1>,
  direction: ingress,
  protocol: tcp,
  local_port: <parameter2>

volume:
  ro: public,
  rw: container

objectstore:
  default: local

image:
  any
```

Figure 3.3: Basic Container Pattern

I will extend this basic pattern with additional containers in order to implement richer multi-container functionalities. With CQSTR alone at least one separate container is needed for each customer, which can be inefficient for lightweight services. But, CQSTR can act as a platform for building higher layers of trust, so finer-grained containment is possible. For example, a trusted OS running within CQSTR could isolate instances of a service with containers.

3.6.2 SpamAssassin

SpamAssassin has a simple work flow: a client sends the service a stream of emails, and the service scans each email for spam. It adds an email header indicating the likelihood that it is spam and then returns the message back to the client. My security requirement is that SpamAssassin should not disclose emails to other parties, including the service's operator.

This workflow fits the pattern shown in Figure 3.3, with the client's address as parameter1. The client of SpamAssassin queries the metadata server with the service's address, and verifies that the returned manifest uses the template with the correct parameters.

I extend this basic design to allow the operator to pass labeled training data to the service enabling it to learn additional spam patterns. Here, I ensure that the act of providing training sets does not leak any information: even TCP ack packets from the service could be used to leak client email content.

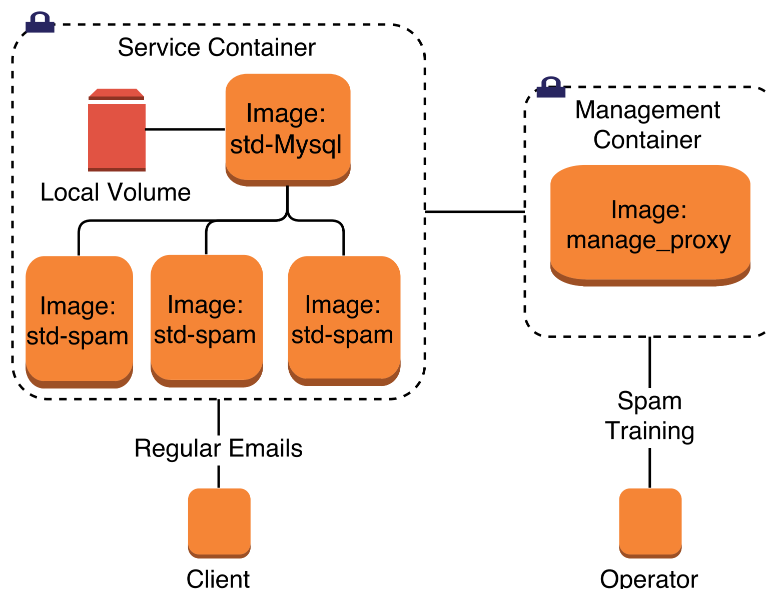


Figure 3.4: Configuration for SpamAssassin. Image name with prefix "std-" means unmodified applications and OS from official sources. Other images contain code that implements management functions.

I solve this by extending SpamAssassin with a second container including a single *management proxy*. A management proxy is a small VM instance that contains only trusted code and provides a limited set of management operations to an isolated service, much narrower than the service interface. This proxy is comparable to AWS API Gateways [21]. Each proxy in my deployments use a slightly modified Ubuntu 14.10 installation that includes a script to export desired management functionality through an HTTP interface. The functionality of a proxy should be simple and open source, so that it can be trusted.

The SpamAssassin proxy allows a single management operation, which is to accept training data from the operator. The service passes the data to SpamAssassin, and returns a fixed acknowledgment to the operator. It runs in a separate container from the service, so the proxy cannot access or return any of the client's email to the operator. The proxy's container allows access to the operator and to the service's container, and I allow the service's container to communicate with the proxy, as shown in Figure 3.4. Because this proxy could be a channel to leak information, I require a locked-down trusted image for the proxy.

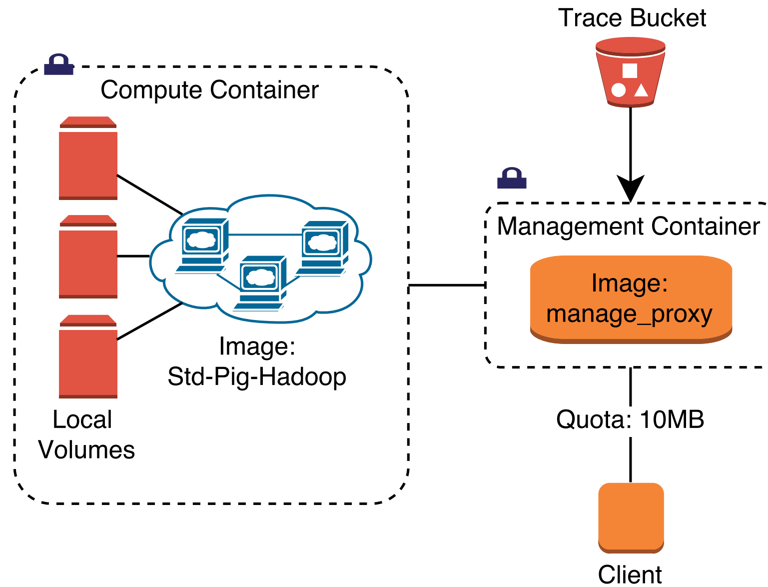


Figure 3.5: Configuration for PacketPig.

3.6.3 PacketPig

PacketPig is a distributed network trace analysis tool [128] built on Apache Pig [29] and Hadoop. I use CQSTR to allow untrusted users to submit scripts that analyze sensitive network traces stored in a Swift bucket by ensuring that the service cannot reveal much about the trace. Unlike SpamAssassin, the data is not owned by the client; instead this is a delegated setting and CQSTR is used to limit the information clients extract rather than prevent release completely.

My basic configuration follows the pattern described above: PacketPig runs in a container that only allows access to a single port to submit scripts and receive results. To limit the amount of data released, I enforce a total download limit on the container that is a small fraction of the total trace size so the complete trace cannot be exposed. The trace data is stored in a Swift bucket. It is not possible to directly reference a Swift bucket without modifying the application. Instead, I integrate the data importing function into a management proxy described below. The data owner sets an ACL on a trace bucket that only grants access to a role from the management container.

I implemented a management proxy that, in addition to providing network traces, allows dynamic addition and removal of nodes in the service, and modifies query scripts to add random noise. PacketPig can only communicate with the proxy, and clients submit scripts and receive result from

the proxy. This configuration is shown in Figure 3.5. The set of nodes is changed by providing the address of the new instance to add or the existing instance to shut down; the proxy modifies Hadoop's configuration and restarts the cluster with the new configuration. The script rewriter illustrates use of a proxy to sanitize inputs and outputs for better security and privacy. It adds uniformly distributed noise to the *SUM* function in scripts, with respect to the average and variance of the data. More complex mechanisms like differential privacy could be implemented [62], but are out of scope for this paper. This use of a proxy demonstrates the more general idea of an *output proxy* that checks or sanitizes results before returning them to the client.

This configuration can also be used more generally as a contained Hadoop/Pig cluster, where clients submit jobs without worrying that the service owner may leak scripts or data. The only modification to the above design is to change the management to allow all scripts.

PacketPig demonstrates how CQSTR can restrict and isolate data flows. However, service-generated data, such as program outputs and logs, might still carry sensitive information. Hence, in deployment one would need to carefully whitelist which scripts and supporting service code can be allowed to run within the cloud container.

3.6.4 PredictionIO

PredictionIO is an open-source prediction service, and demonstrates how CQSTR solves the example problem in Section 3.2. PredictionIO reads a client's data, with which it trains a model based on a code template from the client. With the model, PredictionIO deploys a webserver to make online predictions. My security goal is that the training service does not leak the client's data; the data owner allows the PredictionIO service to train on its data and make public predictions from the resulting model but the training data must stay private.

I created an image using the standard Prediction IO codebase with a trusted prediction template. I install this on a locked-down configuration of Linux with unnecessary services (including sshd) disabled. In addition to the PredictionIO code, the service also runs a standard Apache Spark cluster [28] to provide the compute power needed to generate the prediction model.

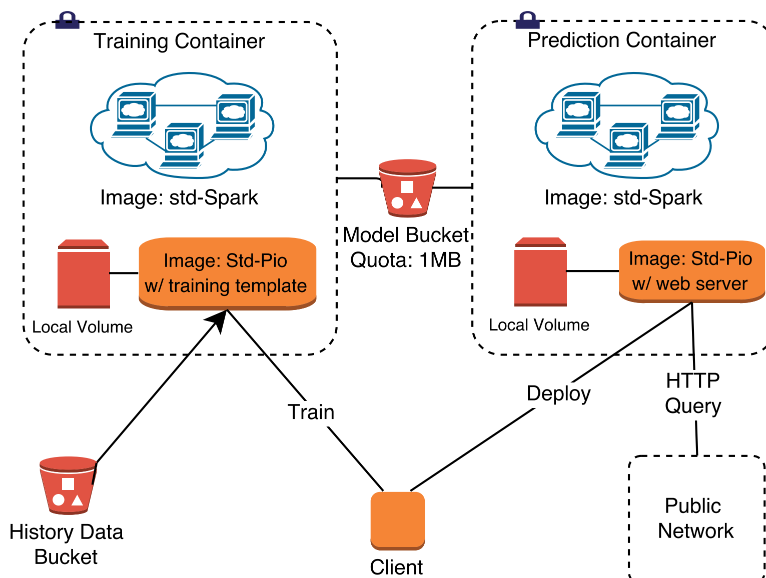


Figure 3.6: Configuration for PredictionIO.

As shown in Figure 3.6, I deploy the training service in a cloud container with access to the client's data source. The web server for prediction queries runs in a separate container that shares a Swift bucket with the training container, and allows public access. I limit the quota of data that can be written into the bucket to the model's estimated size (1MB).

This configuration ensures that the training data is limited to the training service, and the publicly accessible web server has no direct access to the data. Even if the web service container is compromised due to an application or OS vulnerability, the attacker cannot access the training data.

My deployment of Prediction IO with CQSTR demonstrates how an application can be decomposed into components, each in a separate container. Using a similar decomposition with rules allowing access to specific endpoints, an application can contact trusted third-party services. If they are deployed in cloud containers, CQSTR can also verify their trustworthiness.

3.6.5 Experience

Porting the three applications to operate in CQSTR was straightforward, requiring no source modification. The management proxies are small, simple services to code. I ran into a few issues

with services that assume open Internet access: SpamAssassin looks up the hostname in all messages using DNS, but access to DNS can leak information about what messages are being scanned. Some prediction templates for PredictionIO access external services, which may not be allowed in a cloud container. In both cases, I disabled code that accessed external services without hindering the desired functionality.

3.7 Performance Evaluation

I evaluate the performance cost of CQSTR's modifications to OpenStack on my three applications.

3.7.1 Configuration

I run CQSTR with nine physical hosts on the Wisconsin cluster of CloudLab [1]. One node is the cloud controller, one is the network gateway, two serve as both volume and object storage, and five are compute servers.

All nodes have two 8-core 2.40 GHz Intel E5-2630 CPUs with hyperthreading enabled and 128GB memory. The code is stored on a local SAS disk, while storage services (Swift and Cinder) use an SSD. User VMs are connected through a GRE tunnel over 10Gb Ethernet, with a separate 10Gb Ethernet for storage requests. I multiplex the management network with the data network, with address spaces isolated.

I used Ubuntu 14.04 LTS image for guests and services. Both guests and services run in m1.large instances, which have 4 virtual CPUs without usage caps, 8GB memory and 80GB file-backed storage on a SAS disk. Management proxies are lightweight and run in m1.small instances with 1 VCPU and 2GB memory.

3.7.2 Application Performance

Table 3.3 shows the workloads and average completion time over 10 repetitions for PredictionIO, PacketPig, and SpamAssassin. The Email dataset is the EDRM Email DataSet V2 [2], the network trace comes from ISTS12 [3], and both training and predictions use data from the Million

Application	Workload	Native	CQSTR	% diff.
SpamAssassin	Filter 100,000 emails	598 sec	604 sec	1.0%
PacketPig	Fingerprint 146GB trace	856 sec	869 sec	1.5%
PredictionIO Training	Train on 10,000 songs (1.8GB)	26 sec	27 sec	3.1%
PredictionIO Prediction	Perform 1,000,000 predictions	914 sec	925 sec	1.2%

Table 3.3: Application workloads and execution time.

Song Dataset [38]. All results have standard deviations below 3%; the prediction workload has a 1% deviation.

The performance of PredictionIO predicting and PacketPig on CQSTR are 1.5% slower than the native versions. The latency of predictions is unchanged. For SpamAssassin CQSTR is 1% slower.

3.7.3 Microbenchmarks

I created a set of microbenchmarks to better understand why application performances was unchanged.

Container management. I measured the cost of management operations to create and launch a service in a container. Creating a container takes 1.5 seconds and configuring its security properties takes 3 seconds, which is a fraction of the time to instantiate and boot an instance (11 seconds).

Existing management APIs. CQSTR imposes additional checks on existing IaaS management APIs, e.g., attaching a volume or IP address to an instance. In measurements, I found these checks add less than 2% overhead.

Network performance. CQSTR adds extra firewall and traffic rules compared to normal instances. Using iperf to test bandwidth and latency, I found no measurable difference.

Storage performance. For volume storage, additional checks are only imposed when a volume is attached, so there is no performance impact on data access. For object storage, Swift must call the metadata service with the requester's IP address to verify state assertions, which takes 30ms on average. The result is cached for the minimum IP address reuse time (10 minutes). In

my experiments, the average latency of object storage requests is unchanged at 12ms, but the 99th percentile latency increases from 250ms to 750ms, reflecting the time spent verifying the requester's capabilities. Bandwidth is unchanged.

External storage performance. External services must fetch state assertions before granting access to data, which takes approximately 700ms. However, the result of checking assertions can be cached as with Swift, and hence has little impact on overall performance.

These results explain why application performance is largely unchanged: containment operations rely on the existing behavior of IaaS services, or are cacheable, so there is effectively no performance overhead at runtime.

3.8 Summary

Controlling use of data is central to secure computing. CQSTR is the first system to leverage existing IaaS infrastructure to provide control over data usage to clients of a service, and to do so without TPMs or cryptographic protocols. CQSTR is directly applicable to a wide variety of data-analysis applications, and can also be used as a general application-structuring technique. It provides a basis for OS and application-level mechanisms to further confine code. However, CQSTR currently works for a single provider, as it relies on IP addresses for authentication. How to extend across multiple cloud platforms remains an open question.

Chapter 4

TapCon

4.1 Introduction

Services are often composed of multiple pieces of software, managed by a variety of frameworks. In the cloud, infrastructure-as-a-service (IaaS) providers launch virtual machine instances (VMs) with specified program images. Within those VMs, a platform service based on a container manager such as Docker may further launch microservices from yet another set of images.

CQSTR provides assurance for IaaS level tenant service, however, it does not yet answer the question of how to reason about security properties of applications beyond the IaaS level. For example, a service could run as a Docker container inside VM. Clients of this service may want assurances about the service's container image before trusting it with sensitive data. I want to enable reasoning about the whole stack of software of such Platform as a service (PaaS) layer by considering the properties of the VM, the orchestration platform, the container, and how they are combined.

For this purpose, I describe TapCon, an attesting container manager built on Docker. TapCon publishes attestations about containers, which clients of TapCon can use to verify security properties of these containers, so that they can decide whether a target container can be trusted. I first provide an overview (§4.2), followed by a description of the TapCon service (§4.3), the role of logical attestation (§4.4). I conclude with a brief evaluation of my current prototype to show that attestations are sufficiently cheap for practical use.

4.2 Motivation and Overview

As another motivating application scenario, consider the problem of *joint data mining*: suppose that two groups each have a private dataset and want to cooperate by running analytics program P over both datasets (A and B) together. They trust that P produces output that does not expose confidential details of A or B to one another. They choose to leverage cloud infrastructure and common platform frameworks—an open analytics stack in Docker containers—to deploy P and grant it access to A and B. How can they ensure that their datasets are accessible by a VM instance only if it runs the correct P?

Attestation-based access control. I propose that each group installs an access policy that permits data access from an instance running P—an example of *attestation-based access control*. The cloud storage service that stores A and B examines attestations of the layered cloud stack to verify that each data requester complies with a policy that the data owner provides or approves: e.g., it grants access if the request originates from a suitably trusted program P running in a secured environment (§4.2.3, §4.4).

Sealed software appliances. For cloud attestation to be secure, a trusted program P must be *sealed* so that the instance owner is blocked from subverting it through the management API, e.g., by replacing P after the instance launches, or taking a snapshot that exposes its data (e.g., datasets A and B). Instead, each trusted program P incorporates its own management API that is part of P's definition and respects its advertised security properties. I refer to programs with this property as sealed software appliances.

Layered Attestation. I focus on validating the end-to-end trust chain for application instances running at the top of a layered cloud stack. Each layer of the stack launches a virtualized environment or interpreter for programs it launches above it, and publishes attestations for those programs. Figure 4.1 depicts layered attestation of an application container by a server running TapCon, which runs as a VM that is itself sealed and attested (§4.2.2, §4.3).

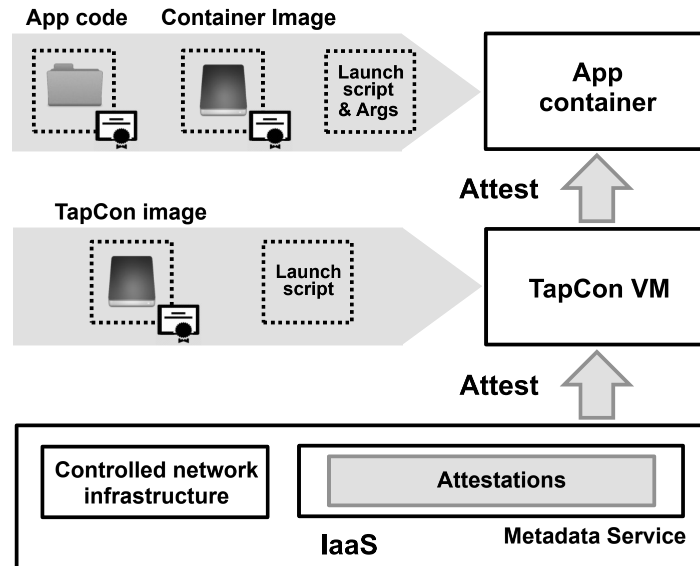


Figure 4.1: Layered attestation example: a trusted IaaS service attests a sealed TapCon VM, which launches and attests application containers. The shaded areas represent attested programs and their attestations stored in an IaaS metadata service. TapCon attests to the container's image, application code, and the launch script and parameters.

4.2.1 Trust Model

TapCon is built on top of CQSTR. I assume IaaS provider is trusted, and TapCon VM image is a well known image that can be trusted to contain the TapCon service. CQSTR issues trust statements attesting to this image for each TapCon VM, and includes a trusted *metadata service* to store these statements (Section 3.3).

4.2.2 TapCon Service

To illustrate how to build an environment for layered cloud attestation, I developed a simple tenant-managed platform-as-a-service (PaaS) layer. TapCon is a virtual appliance image that implements a sealed Docker container service running as a tenant VM over the IaaS layer. TapCon supports a novel combination of features for practical cloud attestation.

Seal the management APIs. Container states can be directly altered using management operations provided by container daemons. For example, one can directly copy files from/to a Docker

API and Options	Description
`MountNS`	Disable user mount namespace
`PrivilegedFlag`	Disable privileged container
`Exec`	Restrict executing only predefined command in a container
`Copy`	Disable copying in/out files to/from container
`BindMount`	Disable mounting host path as volume to container
`Volume`	Restrict volume to a few specific locations
`Capability`	Disable capability add
`SecurityOpts`	Disable modifying security options like seccomp, AppArmor profile
`Device`	Disable host devices access
`NetMode`	Only support ”bridge” mode so far
`Inspect`	Hide environmental variables in output
`ImageCommit`	Disable snapshotting image from running containers

Table 4.1: Functions for sealing in TapCon.

container through the daemon. Table 4.1 describes the APIs to fully or partially disable in order to make sure attestations are valid during the life of containers.

Source-based attestation and certified builds. Attesting to a binary hash is sufficient for first-party attestation: it proves to the instance owner that the instance was launched securely from an image built by the owner. But for third-party attestation I also need assurance of the provenance of the binary. TapCon attests that a container is launched from a trusted code repository—*source-based attestation*. It generates the corresponding binary through a known and trusted build chain that is part of its trusted computing base (certified builds).

Authentication. Any code attestation system must provide some way to authenticate communications with the attested instance, so that insecure software cannot masquerade as a secure instance. Hardware-level attestation—and many software attestation systems—bind a software identity to a keypair whose private key is held by the attested service. This approach introduces various overheads and practical challenges to manage keypairs safely, and may require substantial changes to software to sign and/or encrypt communications.

As an alternative, my system leverages the managed network of the IaaS provider to block any spoofed packets. On a secure network, any communications from an assigned network address are known to have originated from the instance that owns the source address (§4.3). This property enables us to use network addresses as secure identifiers for instance endpoints [26, 96].

4.2.3 Logical trust

Trust in an attested service is based on a chain of statements extending across layers in the system. The chains are rooted in one or more trust anchors: for example, a relying party may trust the IaaS provider and authorities who endorse source code or binary images. In the joint data mining scenario, the data server validates the attestations for the instance running P, and checks each data request for compliance with access control policies of the owners for datasets A and B. The chain is rooted in the IaaS layer and extends through the TapCon PaaS layer (Figure 4.1).

My approach to attestation-based access control uses standard Datalog logic to represent attestation statements, access policies, and other statements of trust (§4.4). For example, logical trust is also useful to represent endorsements of trusted programs. An access policy may specify trusted programs directly—e.g., by hashes over source or binary artifacts—but a more general approach is to grant access to programs with *security properties* of interest. Then, for example, if a new version of a program P is endorsed as having the relevant properties, there is no need to change the policies to reference the new version. For example, an endorser of the program P in the joint data mining scenario might assert that P does not leak its A and B inputs. The owners of A and B must trust the endorsers a priori (e.g., by social trust) or based on statements about the endorser by another trusted party (e.g., the owner's employer, or a consortium).

Logic gives us a powerful language to represent the statements of the endorsers and attesters, and also any delegations that provide the basis for trusting them. The logical trust system can expose all such trust assumptions and reason from them rigorously to infer security properties for attested services.

4.3 The TapCon Layer

TapCon implements a simple tenant-managed PaaS service with features for layered attestation. Any IaaS tenant can instantiate a TapCon cluster using a *TapCon base image*. My modified IaaS service launches a TapCon VM instance, and attests it as running the TapCon image. It stores the attestation in its CQSTR metadata service.

The TapCon image contains a standard Linux stack with a Docker daemon to launch containers. I modified the daemon to issue attestation statements for its containers, identifying the launched image (source code and data), along with parameters and launch scripts.

Authentication by network address. An attested instance is a security principal whose identity in TapCon is authenticated by a network address. In doing this, I presume that the cloud provider configures ingress filtering and host-side virtual networks to block any packets with a spoofed source IP address from entering the network. Public clouds (e.g., Amazon/AWS [15]) support this property.

In my TapCon prototype, the OpenStack IaaS platform initially assigns an unspoofable IP CIDR with 256 addresses to each TapCon VM; all containers launched within the VM receive an unique IP address on IaaS network. At container launch, TapCon issues an attestation statement that binds the code identity to the container's IP address. The metadata service records the TapCon instance—also authenticated by its network address—as the "speaker" of the attestation.

Layering. This architecture extends naturally to higher layers. For example, if software in a container is itself capable of launching secure instances (e.g., processes running within a trusted interpreter/sandbox) and sub-delegating port ranges to them, it may publish an attestation for an instance to the metadata service. These statements are authenticated as spoken by the container because it sends them from a port that it controls. Another party—e.g., a client or a storage service—may accept the attestation if it trusts the container to issue such statements. For example, it may trust that the container is running a trusted interpreter program, based on a TapCon attestation about the container itself. In this way the sequence of statements about the layers of the software

stack form a logical *attestation chain* in which trust in each layer derives from statements by the layer below. These chains are validated by recursive logic rules that can accept chains of arbitrary depth (§4.4).

Certified builds. TapCon binds each attested container instance back to specific source code: its trusted build service publishes attestations about the source code and other elements of the build. I modify Docker's build system to provide tighter control over the sources of build information. First, the build service accepts only a Git URL to specify the source to build, which embeds a self-certifying Merkle hash over a source code version. Second, all files downloaded during the build are hashed and attested along with the Git URL. Thus another party can verify that code was built with trusted tools, libraries, and packages, and anyone may audit the build environment by reproducing the certified build locally and comparing the results. To provide a practical starting point, I enable the use of a few official base container images, e.g., Ubuntu and Debian. Other external base images are blocked.

4.4 Logical Trust for TapCon

Logical trust offers a natural formalism to represent the statement types summarized in §4.2—layered attestations, program endorsements, network address delegations—and also validation rules, access policies, and local trust anchors. My approach follows my work on SAFE [46]: it uses pure Datalog as the trust language, uses scripted linking to connect related statements in DAGs that match the delegation structure, and publishes linked trust statements in an indexed put/get store—the IaaS cloud metadata service. To use Datalog as a trust logic, I extend its syntax to add a "says" operator (":") that attributes each statement to a speaker—the authenticated principal that asserts it, e.g., an instance bound to a network address.

Compliance checking. Any running program may use an off-the-shelf logic engine (Styla) to query and validate an attestation chain according to its locally accepted trust anchors and policy rules. The authorizer's logic engine and policy rules consider only statements whose authenticated

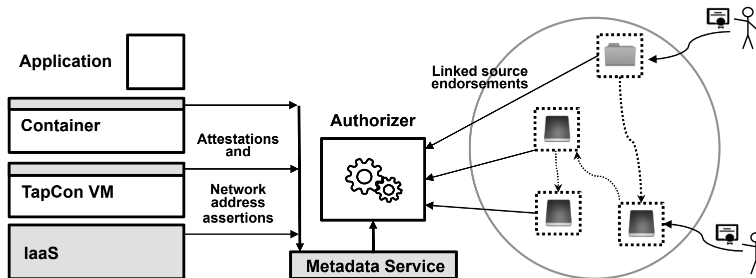


Figure 4.2: An application principal that performs attestation-based access control checks is an *authorizer*. It runs an off-the-shelf logic engine to evaluate logical guard conditions according to its local policy rules. The logical rules evaluate code attestations published at each layer, and endorsements of the attested code.

speakers are locally trusted to make those statements. The logic engine infers this trust by recursive application of the policy rules.

Trust anchors. All valid inferences are grounded in locally accepted trust anchors, which are the bases for the recursion. For example, an authorizer must trust the IaaS provider (authenticated by keypair) and any TapCon instance as a source of attestation statements; Listing 4.1 uses an `attester` property (rules **R1** and **R4**) to represent this trust. **R4** presumes that a trusted *endorser* endorses the TapCon binary as a secure container service that issues trustworthy attestations. Given that the TapCon source code and build procedure are open, anyone can verify that it implements a secure container manager and build chain. For simplicity I presume that the endorser is trusted a priori to say so, i.e., it is a trust anchor.

Layered attestation chains. An attestation chain is a linked DAG of authenticated logic statements that establish the code identity, execution integrity, and network address(es) of the program instance at the top of a cloud stack (Figure 4.2). Layered cloud services, including IaaS providers and the TapCon platform (PaaS) service, issue statements about their child instances at launch time. A TapCon VM that attests a launched container—binding it to an image and a network address range—also issues statements to bind the image with a source code repository. Other parties (*endorsers*) make statements about the security properties of programs (images and/or source repositories). The issuers of all of these statements link them in overlapping DAGs, following [46].

An authorizer may consider all of the statements in an attestation chain together to make access control decisions. It checks compliance with its local policy by evaluating a specified guard condition for access control against a linked set of logic assertions and policy rules governing the authority of speakers to make those assertions. If a request is valid, the engine generates a logical proof of compliance.

Listing 4.1: Policy rules to validate a layered attestation chain.

```
(R1) attester(H) :- trustedCloudProvider(H).

(R2) runs(Instance, Image) :-
       runsInstance(H, Instance, Image),
       attester(H).

(R3) runsInstance(H, Instance, Image) :-
       AuthNID: attest(Instance, Image),
       bindToID(H, AuthNID).

(R4) attester(Instance) :-
       runs(Instance, Image),
       E: endorseAttester(Image),
       endorser(E).
```

Validation rules. Listing 4.1 shows simplified rules R1-R4 to validate an attestation chain. Each rule has a *head* on the left, which represents a belief that is implied by (“`:-`”) a list of subgoals in a *body* on the right: the head is true if all subgoals in the body are true, under some assignment of string values to variables (capitalized terms).

R2 allows an authorizer to infer that an instance runs a specific program, if some valid host attests it and is trusted to issue such attestations. The value of P is a hash over the code for P and its configuration, as shown in Figure 4.1. **R3** authenticates an attestation statement to a host,

R1. Accept attestations from IaaS cloud providers I trust: grant them the `attester` property.
R2. Believe that a named `Instance` runs a named `Image` if some host H attests that it does, and I trust H to issue attestations (H has the `attester` property).
R3. Believe that a host H issued an attestation statement if it was spoken by an authenticated identity `AuthNID` (a public key or secure network address) bound to host H.
R4. Trust an instance as an `attester` if it runs software that is endorsed by some trusted `endorser` as implementing a secure platform whose attestations are trustworthy.

Table 4.2: Meaning of the recursive logic rules in Listing 4.1.

e.g., if it was spoken from a network address that is bound securely to that host by other rules for $bindToID$ (not shown). It could use other authentication methods (e.g., keypairs) to establish the binding; this is a form of reconfigurable authentication [105]. The recursion in **R4** enables these rules to check layered attestations of any depth. **R1** is a basis for the recursion: it accepts attestations from a trusted IaaS provider.

Validating an attester. **R2** requires that the attesting host H at each step is accepted as a valid $attester$: this property captures the belief that H is faithful in launching guest instances, binding them to secure network addresses, and attesting their programs. This condition is satisfied if, for example, H is a trusted IaaS cloud provider (**R1**), or if H is an instance that is itself attested by its own host as running a secure program, and some endorser E endorses that program for the `attester` security property (**R4**). Trust in the endorser E—and in the IaaS provider—is also derived from authenticated (e.g., signed) logic statements and/or local policies (not shown). An authorizer could, for example, accept endorsers that are approved by its enterprise or by an open-source consortium, or it could choose to trust only itself. Endorsers may endorse other security properties as well; I can adapt the rules to limit the properties that each endorser is trusted to assert.

Authorizing data access. In my joint data mining scenario, a policy decision to grant access to dataset A or B can be based on the identity of the code running in the requesting instance, and the data owner's beliefs about the security properties of that code. As shown in Figure 4.2, the data

Docker	TapCon	Posting Statements
488ms (±40ms)	497ms (±49ms)	31ms (±8ms)

Table 4.3: Container boot time for unmodified Docker and TapCon.

owner attaches an access policy to an object. The data owner trusts the authorizer—in this case, a data storage service—to apply its policy faithfully.

Listing 4.2 shows an exemplary rule that verifies (i) that the $Owner$ says that $Endorser$ is trusted to endorse $Property$ about the program, (ii) that $Endorser$ says that $Program$ has $Property$, and (iii) $Owner$ says the object can be accessed by a program with $Property$.

Listing 4.2: Policy rule that grants access based on accepted properties of an attested program that the requester is running.

```
hasAccessPrivilege(Program, ObjID, Owner) :-
  Owner: trustEndorserOn(Endorser, Property),
  Endorser: hasProperty(Program, Property),
  Owner: accessPrivilegeByProgramProperty(
                              Property, ObjID).
```

4.5 Prototype Evaluation

I measure the overhead of my TapCon prototype to attest launched containers and for attestation-based access control checks. I use a 3-node cluster on CloudLab. Each node has 16 Intel E5-2630 cores, 128GB memory, and two 10GbE interfaces. The cluster runs Openstack and Docker with extensions for TapCon. I use the Styla logic engine to perform compliance checks.

Boot overhead. The table 4.3 compares the latency to launch a container in Docker and in TapCon. The third column shows the time for TapCon to attest a container to the metadata service.

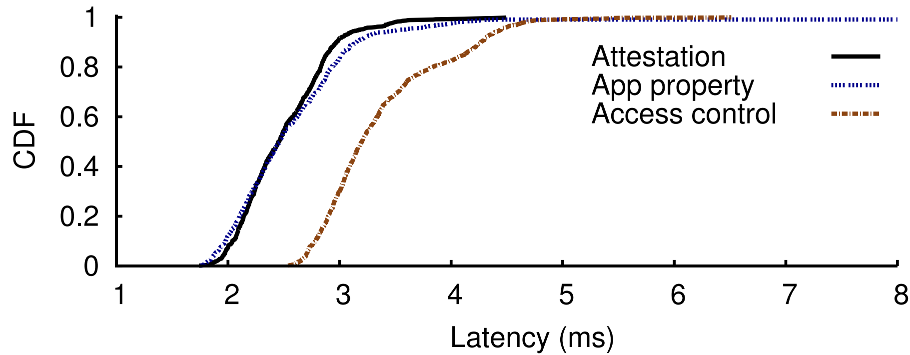


Figure 4.3: Latency of queries for three types of compliance checks, measured under concurrency level C=10. Latency includes Styla prover cost, related scripting costs, and network latency.

Overall, TapCon is only 2% slower than Docker, because posting statements can be overlapped with container launch. The overhead to attest VMs is negligible, as each VM takes tens of seconds to launch.

Performance of access control. I measure the latency for three types of attestation-based compliance checks: validation of layered attestation chains, validation of endorsed application properties, and attestation-based access control (Listing 4.2). I issue to the metadata service statements for 10,000 containers, 100 endorsed properties for each application, and 100 ACL entries for each data object. The test harness performs 100,000 random checks of each type on a hot cache. Figure 4.3 shows that most of the queries complete within 5 ms. Access control queries rely on ACL entries, the cost of which is linear to the ACL length.

4.6 Conclusion and Discussion

The chapter proposes to incorporate attestation above the standard IaaS cloud abstraction, and shows how to apply it to layered platform services. The paper describes a small set of cloud provider extensions directed at enabling a rich foundation for trustworthy cloud computing. In particular, TapCon serves as an example of how layered attestation makes it possible to deploy *tenant-managed* security services, promoting extensibility of secure cloud platforms. I recognize

that the topic of logical trust—a key element of my approach—is unfamiliar to many in the systems community, despite well-understood rigorous foundations and many case studies showing potential for practical value. I hope that the paper can spark wider exposure and discussion of the power of logical trust and its potential uses for data-centric trust in a cloud setting.

TapCon does not address all the problems. The service only works on containers, and it is a single host service, which has no mechanism for supporting distributed applications as well as their complicated configurations. Further, the assumption about flat network model can be over-simplified for modern platforms. For example, Kubernetes platform typically operates two networks, one isolated network segment for containers, and one for service access from public network. How to properly fulfill network based authentication would become a new challenge under such scenario. Lastly, using TapCon requires trust on the TapCon VM image, which is a binary artifact itself. How to enable source based attestation so that authorizers can analyze only source repository is necessary for practical usage.

Chapter 5

Latte

5.1 Introduction

Latte fully realizes the vision of pervasive attestation and flexible authorization in a cloud environment (section 2.2). This chapter first provides an overview to the framework (§5.2); then I will define the basic mechanism and logics for for code based authorization in public clouds (§5.3). Based on the basic mechanism, I will describe a list of necessary logical extensions that capture common software use patterns in public clouds (§5.4). Section 5.7 includes necessary details to implement such basic mechanism and logical extensions. To demonstrate how the concepts work together, I extend TapCon design to Kubernetes [77], a state of art container orchestration platform (§5.5). The Latte-aware Kubernetes provides a solid starting point for using code trust on IaaS clouds, and I developed rich application scenarios which overcome all the challenges at the beginning of this dissertation (§5.6). Evaluation results show that integrating Latte into public clouds only incurs negligible overhead.

5.2 Overview

5.2.1 Authorization Using Latte

Figure 5.1 depicts the phases and principal roles for a typical program running as an instance in a Latte-enabled cloud. In the build phase, a build service (*builder*) prepares a program image and certifies the build from an identified source repository. Other parties (*endorsers*) may issue endorsements of the source code (top) or built image (bottom). In the launch phase, a Latte-enabled hosting cloud platform (an *attester*) launches an instance from the image and certifies its image

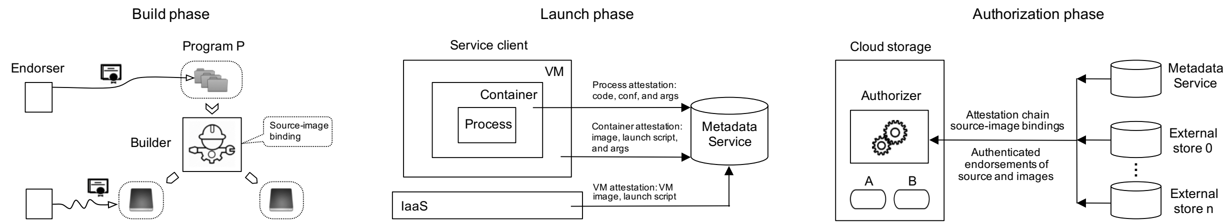


Figure 5.1: Principal roles in three phases of Latte operation for a typical program running as an attested instance in the cloud.

identity and configuration. In the authorization phase, an *authorizer* introspects on the instance metadata (certifications and endorsements) to check compliance with a policy. For example, if the instance requests access from a Latte-enabled service (it acts as a *requester*), the service may perform an attestation-based access control check (acts as an *authorizer*). The basic authorization policies and endorsements are defined in section 5.3.

These phases are similar but in fact different from TapCon in two ways: firstly, TapCon can only build local containers on each TapCon VM, while Latte allows usage of any building tools, which will publish binding of general program images to their source repositories . In Latte, authorizers can reason about the trustworthiness of the binding by checking against the entire building history of a program, such as the tools and environments (section 5.5.5). For example, a program owner could rely on popular continuous integration/continuous delivery (CI/CD) facilities, and authorizer can define authorization policy to restrict what CI/CD tools, e.g. compiler version, must be used to build the requesting program. Secondly, not only individual containers, but also arbitrary distributed software instances can integrate with Latte for code based authorization on platforms integrated into Latte (§5.4.5). For example, in §5.5, I will discuss how multiple Kubernetes VMs can be verified to comprise a trusted cluster.

Latte defines client libraries to be used in above phases: an *attestation library* that issues statements from predefined templates, and a *guard library* to check retrieved metadata for compliance with a policy, which is specified as a set of logical rules.

5.2.2 Trust Model

Root of Trust. Latte makes the same assumption as CQSTR and TapCon about IaaS platforms. It assumes security and isolation properties of IaaS as a given. Latte can be implemented in a more general way, but in this dissertation I mainly focus on a single IaaS environment. §7.2 provides a discussion about ground the trust chain with multiple federated clouds, or with hardware roots of trust, and/or instantiate security-critical code modules with a hardware-attested minimal trusted computing base (TCB), as in Flicker [108], Haven [37], and SCONE [33].

Trust in Code. For all instances running in Latte other than IaaS infrastructure, an authorizers can reason about their trustworthiness. This is done through defining logical guards(§ 5.4.1). Authorizers loosely define trust anchors, such as lists of trusted code, expected properties, and other logical facts that are related. Such anchors are created according to authorizers' own believes in code, or are endorsed from trusted endorsers of authorizers. Logical guards then leverage these anchors to check about whether a target instance is trustworthy or not.

Trust in Network. Unique network addresses are used by Latte to retrieve metadata about a certain instance. I choose to trust the IaaS network initially, following the assumption of IaaS being a root of trust. The IaaS platform controls the internal network that interconnects its tenants, so that network addresses cannot be spoofed or forged [14, 188]. Latte uses instance IaaS addresses as authenticated principal identifiers. This choice distinguishes Latte from many previous systems that use public key infrastructure to authenticate attested instances. I assume an unconstrained flat IP namespace on IaaS network, e.g., this can be achieved, since IPv6 networks have been more widely deployed in recent years [23]. Relative to these approaches, Latte is compatible with applications that do not use cryptography, reduces communication overhead, and does not require secure key management, which itself is a challenging problem [48].

Sealed Instances. Attested properties are only valid if instances are *sealed* to block any tampering that might undermine the properties: their configurations are fixed and they cannot be changed

by management APIs, which should be enforced even for owners of these programs. Sealing requires that the application code itself incorporates its own management interface, which is subject to inspection at endorsement time. For example, a privacy-preserving survey program might provide APIs to close a survey and output aggregated data, but not to read individual responses. These requirements are already met in both CQSTR and TapCon, as well as a modified Kubernetes, and also a Spark cluster described in section 5.6.

5.2.3 Logical Trust in Latte

Latte logic is extended from the TapCon mechanism. The extended logical approach allows Latte to address several challenges for cloud attestation. Importantly, logical rules can integrate statements published by multiple principals, enabling an authorizer to inspect the full stack of a distributed cloud application. I show how the logical structure supports querying instance properties and configurations (§5.4.1), layered cloud platforms (§5.4.2), authentication by network addresses (§5.4.3), source-based attestation (§5.4.4), and composition for horizontally scaled cluster services (§5.4.5). Generally, an authorizer can use Latte rules to specify who it is willing to listen to, what it trusts them to say, and what it needs to hear to approve compliance with its policy.

As I will show, logic enables a rich space of policies that combine metadata from multiple sources in a unified way. Evaluation cost scales with the complexity of the policy and the length of the compliance proof (§5.8). While complex logical policies take longer to evaluate than simple role-based ACLs, my results show that the relative cost of logical access checking can be negligible in practical scenarios. Moreover, caching intermediate results can reduce this cost [110].

5.3 Attestation in Latte

Latte defines an architecture and data model to expose attestation statements about instances and their configurations, maintain them in a cloud metadata store, query the store to retrieve groups of related statements, and process these statements to evaluate policy compliance. Latte reuses and generalizes rules back from listing 4.1, and makes them part of standard constructs in Latte library.

Statement	Description
`runs(PID, ImgHash)`	Instance `PID` is launched from the image with hash `ImgHash`.
`config(PID, Key, Value)`	Instance `PID` was launched with configuration property (Key, Value)
`bindToID(PID, NetAddr)`	Instance `PID` is bound to network address `NetAddr`.

Table 5.1: Simple attestation statements in Latte. Each statement asserts a fact—a logical predicate with constant parameters—attributed to an authenticated issuer.

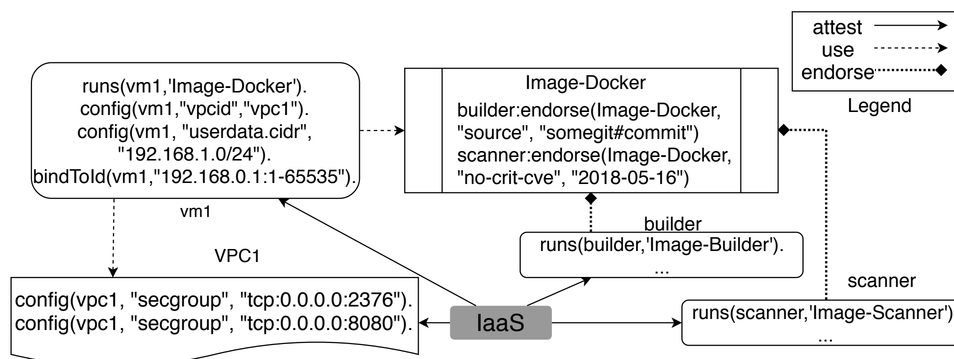


Figure 5.2: Metadata assertions in Latte. Logical statements include attestations of newly launched guest instances, source certifications for images, and endorsements of properties of code objects. The metadata service indexes each statement according to its subject (e.g., an instance or code object). Each statement is attributed to a speaker—a principal that issued the statement—which may also be an instance that is the subject of other published metadata. Statements are linked to both speaker and subject. For example, “vm1” is attached to “VPC1” and uses “Image-Docker”.

5.3.1 Attestations and Endorsements

When a Latte-enabled cloud platform launches an attested instance it publishes one or more attestation statements (Table 5.1) about the instance’s code identity and configuration parameters (key-value pairs). Latte’s metadata service generates a unique principal identifier (PID) for each instance.

Instances or other principals may issue endorsements of code objects (source or binary) identified by a unique hash. Latte endorsers represent these properties as key-value pairs whose meanings are user-defined. Listing 5.1 shows example endorsements coded as logic assertions. The first is from Clair [134], a container image analyzer, that determines that a container has no known critical level vulnerabilities in the CVE database as of a given date. The second is made by an auditor

that the image is sealed against remote shell access (*ssh*) once launched. The third endorsement binds a VM image to its source code on a GitHub repository.

Latte authenticates the issuing principal (speaker) of each statement and attributes the statement to its speaker. Any statement spoken by an instance within the cloud is attributed to the speaker's instance PID. A statement issued by an external principal outside the secure cloud network is attributed to the hash of the external principal's public key.

Listing 5.1: Sample endorsements.

```
endorse(img, "no-crit-cve", 2018-5-17).
endorse(img, "no-ssh", true).
endorse(img, "source", https://github.com/
boot2docker/boot2docker.git#2bb74c92).
```

Listing 5.2: Logic rules to infer instance properties from attestations and endorsements.

```
(F0) trustedCloudProvider("[IaaS-ID]").

(F1) endorser("[endorser-keyhash]").

(R0) attester(H) :- trustedCloudProvider(H).

(R1) runs(Instance, Image) :-
        Host: runs(Instance, Image),
        attester(Host).
(R2) hasProperty(Image, Property, Value) :-
        E: endorse(Image, Property, Value),
        endorser(E).
(R3) hasProperty(Instance, Property, Value) :-
        runs(Instance, Image),
        hasProperty(Image, Property, Value).
```

Guard Predicate	**Description**
`hasConfig(I, Name, Value)`	check if instance I has a configuration
`attester(I)`	check if instance I has attester property (§5.4.2)
`builder(I)`	check if instance I has builder property (§5.4.4)
`hasProperty(I, P, V)`	check if instance I has a customized property P of value V

Table 5.2: Some predefined guard predicates in the Latte guard library.

5.3.2 Validation Logic

Listing 5.2 presents a sample set of logic rules that an authorizer uses to reason from attestations and endorsements to infer properties of a running instance. The inference rule and definition is same as that in TapCon (§4.4).

Facts F0-F1 and Rule R0 configure trust anchors for the policy. F0 states that the authorizer trusts the IaaS as a cloud provider; F0 implies (via R0) attestations issued by the IaaS layer are trusted. F1 asserts that statements signed with a specified key come from an endorser.

Rules R1-R3 capture what it means to validate a simple attestation. Rule R1 implies that if a trusted attester—such as the IaaS cloud provider—asserts that some instance was launched from a specific image, then the authorizer believes it. Rule R2 implies that it trusts endorsers, so if any endorser asserts that an image has some specific property, then the authorizer believes it. Rule R3 implies that an instance launched from an image with a property also has that property (i.e., if an program is secure, then a process running the program is secure).

Suppose an authorizer runs a guard policy that requires the requester to have some specific property. The authorizer loads the logic in Listing 5.2 together with pertinent attestation and endorsement statements, and tests a goal statement `hasProperty` for the desired property. To simplify the work for authorizers, I include a few commonly used guard predicates in the guard library (Table 5.2). If a matching attestation (`runs`) and a matching endorsement (`endorse`) are present, then the inference engine in the guard library concludes that a requesting instance has the required property.

Listing 5.3: Logic rules using `builder` predicate and constrained list of endorsements.

```
(R4) builder(Instance) :-
        hasProperty(Instance, "builder", true).
(R5) hasProperty(Image, "builder", true) :-
        E: endorse(Image, "builder", true),
        trustedEndorserOn(E, "builder").
```

5.3.3 Retrieving Metadata

Each basic statement applies to a single subject. The subject of an attestation is an instance. The subject of an endorsement is a code object (source repository or image). The Latte metadata service (MDS) indexes the statements it stores by their subjects (Figure 5.2). It also stores a mapping from cloud network addresses to instance identifiers (PIDs) (§5.4.3).

An authorizer queries the metadata service for published metadata about an instance identified by network address or PID. The metadata service returns attestations for the instance and any stored endorsements for its code. I show how this indexing can be extended to chained attestations and endorsements in §5.4. The Latte guard library also allows an authorizer to import additional logic content from external services such as a designated Web service or a certificate store (e.g., queried by image hash), as shown in authorization phase of Figure 5.1.

5.4 Authorization Using Logics in a Cloud

Latte supports flexible and general cloud attestation building on the logical foundation in §5.3, including extensions for common software deployment patterns. In this section I discuss the standard logical rules Latte uses to realize these extensions. Note that these rules are not the only ways to address corresponding problems.

Listing 5.4: Logic rules to infer instance configurations.

```
(R6) isolatedContainer(Instance) :-
        H: hasConfig(Instance, "volume", ""),
        attester(H).
(R7) sparkMasterCmd(Instance) :-
        H: hasConfig(Instance, "cmd0",
           "start-master.sh"),
        attester(H).
```

5.4.1 Logical Guards

Guard policies can incorporate rules that require arbitrary conjunctions or disjunctions of basic properties to be satisfied. In essence, a guard is an extended access control list that identifies sets of instance properties that are compliant with the policy, e.g., what access is granted to a requester.

Beyond attestations about the running code, guard rules may consider instance configuration properties. Latte-enabled cloud platforms expose configuration metadata as attested key-value pairs (Table 5.2, §5.7). An authorizer can check for the presence of a specific configuration value with a guard rule requiring its presence. For example, Listing 5.4 shows rule R6 verifying that a Docker container mounts no volumes, so it cannot store any data persistently after it terminates. Rule R7 verifies that a Spark instance starts with command line "cmd0" indicating it a cluster master. I describe how each platform attests configuration in §5.5,5.6.

Guard policies are structured as sets of facts (e.g., trusted principals or other statements the authorizer believes) with guard predicates that help derive high-level properties of an instance.. For user convenience, Latte defines a library of useful guard predicates listed in Table 5.2. Authorizers can use these guard predicates as building blocks for more advanced policies. For example, rule R4 in Listing 5.3 infers a guard predicate called `builder` from an instance property; §5.4.4 and Listing 5.6 show how a rule for source-based attestation uses `builder`.

R5 in Listing 5.3 also illustrates how guard policies use `trustedEndorserOn` to constrain their delegations of trust to endorsers. The policy in Listing 5.2 trusts a named endorser to assert

any property of the code object. R5 limits this trust: it requires that the authorizer trusts the endorser specifically to assert the `builder` property. This restriction on endorsement is applicable to inference of other principal properties, such as `attester` (§5.4.2).

For certain use cases, authorizers need to verify that a correct guard has been properly set. For example, in §5.5 authorizers need to verify a correct join guard is set on Kubernetes master. Such check is beyond my first order logic policy, and requires high order logic instead. To workaround this problem, current Latte design requires the authorizers to publish expected guard policies to a known location, e.g. an S3 bucket. A program that needs to use such policy first sets the correct configuration value, then it downloads the script and publish both the URL and the checksum of the content as a self-spoken statement.

When authorizers expect a guard policy to be used in a remote instance, they first verify if the code identity of the instance is trusted. This first step establishes trust on the instance, then the self-spoken statement about the guard policy can be examined. The authorizer also downloads the script (before authorization), and compares the checksum to instance claimed value. In this way, the problem is reduced to first order. The rule (R12) in listing 5.8 shows such an example.

5.4.2 Layered Platforms and Attester Property

I next show how to extend logical cloud attestation for layered environments, such as IaaS virtual machines running Docker containers running Spark programs. The premise of layered attestation is that any platform-as-a-service (PaaS) server may itself run as an attested instance, or even as a tenant of an attested instance. A PaaS server loads and runs code images in a PaaS-specific format (e.g., Docker container images, Spark .jar files), generating a new instance with its own network address. For attestation to be useful, Latte requires that:

- instances on PaaS are isolated from one another and from external tampering.
- instances on PaaS are sealed from being tampered with PaaS management operations.
- instances on PaaS receive unique network addresses (discussed below).
- PaaS attests hosted instances and calls out to the MDS to publish attestation statements.

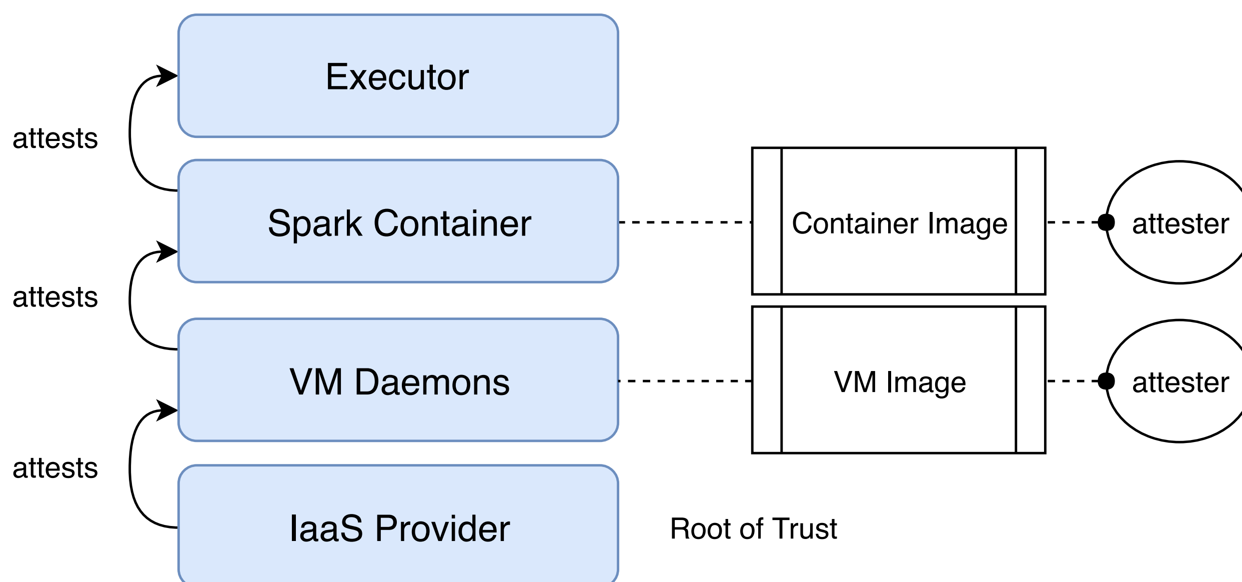


Figure 5.3: software stack view of a Spark application. An authorizer can root the trust of Spark executor process to the root of trust IaaS only when the intermediate layers are all attesters.

An example of such a layered PaaS server is TapCon.

These requirements are captured as the `attester` property, which is used by authorizers to validate the software stack of a requesting instance. To conclude this property on the requesting instance, an endorser asserts this property on an image that implements a layered execution service, after verifying that it meets above requirements. If an authorizer trusts the endorser, and a trusted cloud provider attests the PaaS instance running the image, then it can infer an `attester` property of the instance. Likewise, execution platforms launched within a PaaS platform can also be attesters. Rule R8 in in Listing 5.5 codifies this recursive definition of an attester.

Rule R1 in Listing 5.2 together with R8 can be applied recursively to validate the attestations issued by a layered attester. In this way, Latte's use of logic naturally validates *chains* of endorsements and attestations describing the entire software stack of an instance. For example, consider a setup of Spark applications in figure 5.3. A Spark job will send out data request from an executor process, which can run within a Spark container hosted by a Kubernetes VM. To reason about the executor's code identity and security properties, its hosting container must attest and isolate this executor properly, while the hosting VM must do the same to the container, and so does IaaS to

Listing 5.5: Policy rule for inference of an attester.

```
(R8) attester(Instance) :-
        hasProperty(Instance, "attester", true).
```

the VM. It is easy to see that in this example, rule R8 and R1 together validate the case for the container and the VM, while rule R0 and fact F0 terminates the recursive process as the root of trust. Any failure to see the attester properties would result in a deny the executor request, as its identity can not be faithfully concluded by the authorizer.

To support fast verification of attestation chains, the Latte MDS maintains the lineage between image endorsements and attestation of an instance launched from that image, and the lineage between an instance attestation and its hosting platform attestation. Using this linkage, an authorizer can retrieve a complete attestation chain without unnecessary statements.

For examples of attester programs, I implemented it on CQSTR, TapCon, Kubernetes (§5.5), and Spark (§5.6).

5.4.3 Network Authentication

Latte delegates network addresses hierarchically to ensure that each instance has exclusive control of a unique network address. If an instance is an attester, it controls a block of addresses and invokes the MDS to delegate addresses to its child instances at instance creation time. The MDS verifies that the attester controls the delegated addresses, and updates its map of addresses to instances. The attester must ensure that the instances it creates can transmit or receive only on their assigned addresses. This ensures that network addresses are accountable: each packet uniquely identifies the instance that sent it. The Latte prototype implements address delegation in CQSTR, TapCon, and Spark (§5.6). Particularly, to delegate address to processes, the Latte prototype includes an extension in Linux kernel for TCP/UDP port management, so an IP address can be further broken down and delegated to new instances (section 5.7.3).

Application Network an attester can manage its own application network. This means an instance hosted by this attester can have different network addresses depending on which network is being used. In Latte, MDS is not aware of the application network address. If an attester sets up multiple addresses for an instance, then it should also map these addresses to either IaaS network addresses or instance PID. Currently the Latte prototype still assigns IaaS IP for application network, i.e., the Kubernetets network in my implementation. I will discuss the issues and extensions of application network in §7.2.

5.4.4 Source-based attestation and Builder Property

Many previous systems support attestations for binary program objects. However, software trust is often based on inspection or analysis of source code. For example, FindSecBugs [34] is a source-level analyzer that checks Java source code for common vulnerabilities, e.g., to ensure that the code sanitizes user inputs properly. A key basis for trust in open-source software is open inspection of the source code by a community. Furthermore, attesting source is critical for sharing data across tenants: how can a data owner they trust a binary program without knowing how and from where it was built?

Attestation is more valuable if an authorizer can apply safety properties of source code to binary objects derived from it. In general, that is possible only if the build chain is also trusted [165].

I extend Latte's logic with rules to reason about builds and other program transformations. Latte defines two exemplary endorsement properties: `builder` and `source` for this purpose. The `builder` property represents a belief that an image implements a *trustworthy build service*. A trusted build service runs as an instance that is attested as launched from an endorsed `builder` image via the rules in Listing 5.3 combined with rules R1 and R3 in Listing 5.2.

A builder issues an endorsement after each successful build, using the `source` property to assert that the image derives from a source repository fingerprinted by a secure hash. Listing 5.6 gives a logic rule to apply properties of source to a derived binary. Rule R9 says that if a certified builder B endorses $Image$ as derived from a source repository version $Repo$, and trusted endorser

Listing 5.6: Policy rule to apply a source endorsement to an instance that is attested to run an image built from this source by a certified builder.

```
(R9) hasProperty(Image, P, V) :-
        B: endorse(Image, "source", Repo),
        builder(B),
        hasProperty(Repo, P, V).
```

E says $Repo$ has property P with value V, then the derived image also has property P with value V.

With this extension, Latte provides a powerful mechanism for an authorizer to acquire trust in a service: if it trusts how an image was built from a source repository, and endorsements of safety of the source code, then it has a strong basis to trust an instance executing the image. This construct also naturally aligns with current software development facilities. For example, AWS provides a hosted build service called CodePipeline [11]. Authorizers in Latte could reason about the entire software cycles from its code commit to the final deployment.

To demonstrate how this works, the Latte prototype implements a hosted build service (§5.5.5). This build service will build VM and container images on client requests, and issue attestations to bind the image hash with the repository used before returning the image back to its clients. When an authorizer fetches the instance attestations, these statements will also be retrieved by image hashes, since the instances are already bound with image hashes in the layering logic (§5.4.2). This means authorizers can further validate the speaker of these image attestations, i.e., the build service, the same way as they validate a normal instance. Binding the source repository of the build service is tricky, since the first build service can not build itself. I present a solution to bind source repository of the build service to its image using reproducible build technique, which can generate images with byte-to-byte equality, so that anyone who rebuilds the image with the same condition can have the same image hash, proving the links between the repository and the generated image.

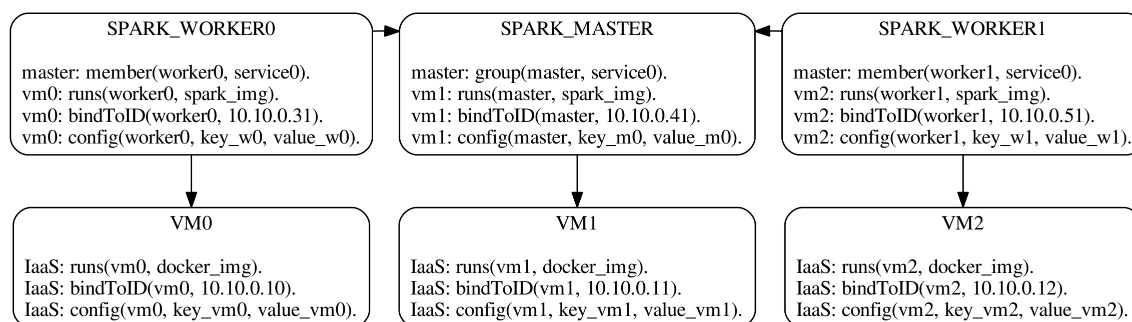


Figure 5.4: Chained attestations in an exemplary service–a Spark analytics platform with layering and composition. Each box has the attestations about an instance, labeled with their subject. A Spark worker instance at the top of the software stack is attested by a chain of attestations from lower platform layers, and statements from an attested cluster master admitting it to a cluster service group as a worker.

5.4.5 Grouping for Distributed Systems

Cloud-hosted services often comprise many server instances grouped in a cluster for horizontal scaling. These instances may share data or have other internal relationships and dependencies. It follows that trust in the integrity of a service as a whole requires some degree of trust in the integrity of all of its instances.

Latte implements a simple grouping mechanism to support clustered services. My approach presumes that a cluster service consists of a group of *worker* instances led by a *master* instance. Each worker contacts the master to join the service group as a worker. The master controls membership in the group. Specifically, Latte grouping requires that the master acts as an authorizer to verify the metadata of each worker and validate its code identity and configuration.

After validating each worker, the master issues a statement to the metadata service granting the worker membership in a named group. This statement is stored with the worker instance metadata (example of Spark cluster in Figure 5.4). An authorizer that queries for the worker receives the metadata of the master as well for validation. An authorizer checks a worker by verifying (i) it is a member of a duly constituted service group and (ii) the master complies with its policy. If

the master's membership requirements (i.e., code identity and configuration) are configurable, an authorizer can verify that the master's configuration meets its requirements.

The authorizer may optionally validate the worker's metadata. However, in my prototype the authorizer does not validate that the worker meets the master's requirements for membership in the group. Rather, it validates its trust in the master and then accepts that the trusted master has validated all of its workers. This optimization reduces validation costs substantially.

Currently grouping is used to verify both a Kubernetes cluster (§5.5) and a Spark cluster (§5.6).

5.5 Latte Kubernetes

To demonstrate how code identities can be used to authorize trusted applications, and to ease the development of trusted applications, I extended the implementation of TapCon to Kubernetes, to provide a starting point for developing meaningful use cases of Latte. The reason of going for Kubernetes is to better support distributed applications instead of standalone containers.

In this section I will first present an overview of this Kubernetes service and the features for running trusted applications. Then I will show how it can be verified by an authorizer for these outcomes. Lastly I will describe the implementation details of new features and a trusted build service.

5.5.1 The Architecture

Kubernetes is the current state of art container orchestration platform. A typical Kubernetes cluster runs as VMs in a flat network, as shown in Figure 5.5. The main functionality of Kubernetes is to provision and manage containers and related resources, such as storage volumes, network addresses, and etc. To achieve this goal, Kubernetes run two types of VMs, the master and the nodes. The master manages the metadata of provisioned resources. It receives client requests to create/update/delete these metadata and sync them to a distributed cluster storage, which is accessible only by a service called *API server* on the master. All other master services and nodes pull related events using the API server's interfaces periodically, and reflect the metadata changes on actual resources. For example, a client can request to create a *pods*, i.e., a list of co-located and

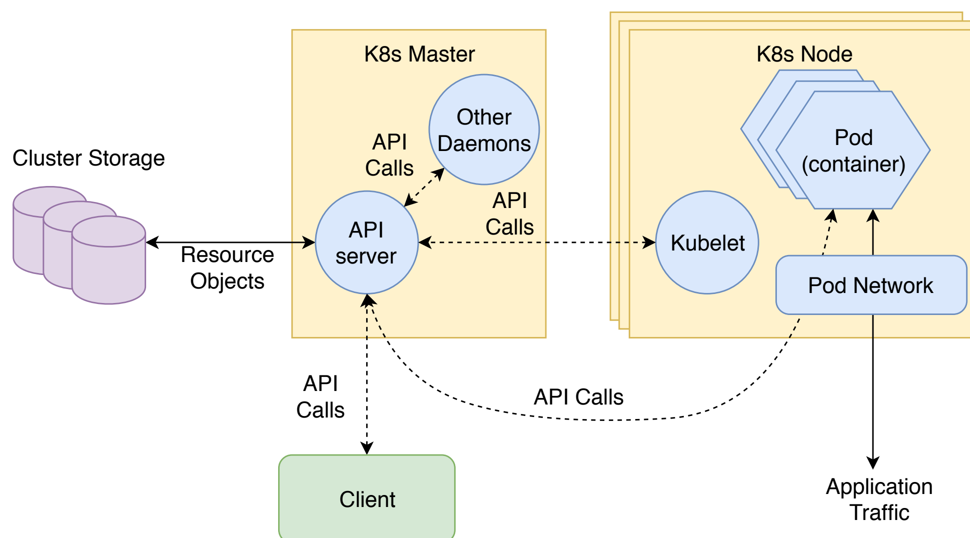


Figure 5.5: Simplified Kubernetes(K8s) architecture overview. There is at least a master VM, and multiple node VMs. On master VM the API server direclty stores resource metadata in a cluster storage(usually an etcd cluster), other daemons such like the scheduler, a node control daemon Kubelet, a node network plugin, and so on will pull resource specification from API server, and sync with actual resource object.

fate-sharing containers. Upon receiving a creation request, the master's API server will extract the specification for this pod, and put it into the cluster storage. Afterwards, the pod scheduler will detect this newly available pod, and schedule it to a node. Then on the target node, the *Kubelet*, i.e., the control daemon on each node, will notice this scheduling event, fetches the specification through the API server, and launches the pod using designated container image and parameters in the client request.

5.5.2 New Features and Guarantees

The goal of setting up this Kubernetes cluster is to use it for running trusted applications. For this purpose, I added a few features to the original version:

Network Authentication. Each pod in this cluster runs with an IaaS IP address. This allows an authorizer to authenticate a pod the same way as a VM. Note this does need support from underlying IaaS network to correctly route the IP to the hosting VM. For example, on Google Cloud Platform, IP aliases can be created for each VM, so that it can be assigned on a container [75].

Listing 5.7: Sample configuration statements for a Pod

```
config(Pod, ‘args0’, ‘start-slave.sh’).
config(Pod, ‘MASTER_URL’, ‘spark://master-url:7077’).
```

Attested Pods. Each pod is attested to their code identities. In addition, its command line arguments, environment variables, and the specification file that is used to create the pod are also published as configuration statements. For example, Listing 5.7 shows an example of two configuration items of a Spark worker pod. The pod runs a script of name ‘start-slave.sh’, and sets an environment variable ‘MASTER_URL’. It should be noted that the configuration space of a pod is quite large. So far I only explored a limited set of them, and avoided those sensitive but difficult options in building trusted applications. For example, I have not configure volumes for application pods yet, so that these pods do not have persistent storage. However, this is not a problem for an ephemeral computation. How to examine with all possible configurations is still an open question.

Sealed VMs and Pods. All VMs and Pods in this Kubernetes cluster are sealed against tampering with their attestations. That is to say, a Pod owner or the cluster administrator can not abuse or misuse management APIs, e.g., to directly change a file in a container file system, or launch an arbitrary command into the container. Similarly, the VM is locked down, so that only Kubernetes service is exposed as public service. Remote login like ssh and vnc are disabled. What’s more, it is not allowed to launch root privileged containers in the cluster. This means the cluster administrator can not use ssh to log in and change the configuration files.

Isolated Pods. Pods are already isolated from each other by default on an unmodified Kubernetes cluster. Latte Kubernetes further offers to create a namespace with declared network connectivity policy. The policy is immutable and can only be set at creation time. All the pods within this namespace subjects to the control of this policy. The motivation for the connectivity is the same as cloud container’s network control in CQSTR.

5.5.3 Verification

For authorizers to trust the Kubernetes service, it means to verify that above features are true. This has two requirements for an authorizer: first, each Kubernetes VM must be examined to launch from my Kubernetes repository [181]; second, the source repository is trusted to provide above guarantees.

The first claim can be justified directly in the authorization policy. What an authorizer specifies in the policy is '`hasProperty(NetAddr, source, Expected)`', here `NetAddr` is a Kubernetes VM's IP address, `source` is a property name (§5.4.4), and the `Expected` is the expected source repository URL. The logical inference engine uses the logic extensions in §5.4 to automatically infer the expected source binding for a given VM.

The second claim requires checking the properties of the Kubernetes source repository, which can not be automatically done. In fact, this is an extremely difficult job if the entire Kubernetes source repository has to be examined. However, a practical starting point is to inspect only for Latte-related modifications, which has only a few hundred lines. The changes can be obtained by comparing the delta of the used version and an official branch. The premise is that Kubernetes is heavily used and actively maintained by a diverse group of people, so that the quality of the code can be guaranteed. For instance, an authorizer can trust that an official Kubernetes does not contain intentional back doors or other malicious code. In this way, verifying the source code for providing above features are much easier, and can be done by any experienced auditors.

Besides the changed source files for implementing above features, the auditor then needs to check the build script. She confirms that running the build script of the repository on a trusted build service (§5.5.5) will generate a locked down VM image, which includes only files of the Kubernetes service and dependent system facilities. In addition, the Kubernetes service should use a known secure configuration, and is the only exposed public service after the VM boots. There should be no remote login such as ssh or vnc. The auditor also checks the build script uses the right version of source code to compile Kubernetes service into the output VM image.

The auditor endorses the `attester` property to the source repository if above aspects are verified. Other properties defined by this endorser can also be endorsed, but how authorizers can learn the semantics of these properties are up to the endorser.

5.5.4 Implementation Details

The changes are implemented on Kubernetes version 1.12.2. This section first introduces how the cluster is provisioned and configured. Then I will discuss implementations of specific features introduced in the Kubernetes, and a sample authorization policy that can be used to verify above notes.

5.5.4.1 Cluster Bootstrapping.

Any cloud tenant can launch the Latte Kubernetes service. The Kubernete cluster is launched into a CQSTR's cloud container. The usage of cloud container is to issue attestations for each VMs. Unlike the use cases in CQSTR, this cloud container no longer disables read only cloud API such as the console logs, because the Kubernetes service can be proved not to abuse such log and dump out client secrets.

The service initialization is handled by 'kubeadm', an official orchestration utility of Kubernetes. On the master, it runs and dumps a join token to console log, and the node VM can acquire this token through IaaS API, and join using this token. The configuration passed to the kubeadm is simple. Right now it only specifies a 'join guard' parameter (introduced below) and other options are default. For example, the privileged containers are by default disabled. The master further configures cluster network using Calico [88]. Its IP tunneling and SNAT function is disabled. Instead, the cluster CIDR allocated is an IaaS CIDR, so that each pod can receives an unique IaaS IP.

5.5.4.2 Implementing Latte Features

Sealed Pods. I reuse the sealing mechanism in TapCon as shown in Table 4.1, as Latte Kubernetes uses TapCon to manage actual containers. For example, the Kubernetes API to directly log

into containers, or execute arbitrary command, including even the administrator. This means my Kubernetes provides sealed pods, which remains immutable after they are launched.

Isolation. Kubernetes already isolates pods using Linux namespace, control groups, and other security features used by TapCon. The namespace network connectivity is created with a Kubernetes network policy object. This network policy object will have a semantic field 'namespace selecotr' set to the associated namespace ID, and it can no longer be changed in order to regulate the traffic in this namespace. When the namespace is deleted, the policy is also removed.

Delegating Node Verification to Master. Kubernetes cluster is dynamic such that node VMs can join and leave freely. It would be a burden for authorizers to keep track of this. Thus, I implemented the group extension mechanism (§5.4.5) to delegate the task to the master VM, with a 'join guard' parameter in Kubernetes API server. The rationale of specifying a join guard as URL and checksum is discussed in the §5.4.1. This guard is specified by the cluster administrator as an URL and a checksum in the master configuration, which points to the actual guard script hosted on an external storage location, such as a file hosting website. The master will download this script and call it whenever a new node VM asks to join the cluster. The node is admitted into the cluster only when the join guard evaluates to true on the master. This delegates the need for an authorizer to check all node VMs to the master VM, and the authorizer only needs to verify the master's code identity and security properties. After a node VM is admitted into the cluster, the master will issue a membership statement for each of them so that when they reach out to an authorizer, the authorizer can transitively fetch attestations of the master through the speaker name of the membership statement. This means verification of the cluster equals to verification of the master.

5.5.4.3 Authorization Policy

Listing 5.8 shows a simple rule for verifying a Kubernetes master (R11), and the join guard to check a Kubernetes node (R10). In R11, the trust of Kubernetes source code files and build

Listing 5.8: Kubernetes guards example. The guards verify that a Kubernetes VM runs with trusted source and configuration.

```
(R10) approveKubernetesNodeJoin(Instance) :-
  hasProperty(Instance, "trustedKubernetes", "latte-k8s"),

(R11) validKubernetesMaster(Instance) :-
  hasProperty(Instance, "trustedKubernetes", "latte-k8s"),
  trustedKubernetesMasterConf(Instance).

(R12) trustedKubernetesMasterConf(Instance) :-
  Instance: hasConfig(Instance, "join-guard",
    "http://<storage-server-url>/approveKubernetesNodeJoin#checksum"),
  hasConfig(Instance, "enable-privilege", "false").
```

script is captured as a special property *trustedKubernetes*. This is endorsed by a trusted auditor Alice (similar to the earlier fact F1). Alice follows above steps to check for the source code, and endorses this property to certify the trustworthiness. The binding of the source repository and the VM instance is also implied in this property, since the auditor will only endorse the correct source repository. The name of the property now is just a plain name, but the endorsement won't conflict with other endorsement with the same name, since Latte allows an authorizer to only use a trusted endorse's statements.

Similarly, R11 expects a predicate *trustedKubernetesMasterConf* defined in R12, which checks configuration of the master VM. The semantic of R12 is to check if the join guard parameter is self-claimed as the expected URL and checksum. The self-claim can be trusted because the *trustedKubernetes* predicate already proves the source code of this instance, which will issue a self-claimed statement about the URL it uses to download the join guard, and the computed checksum of the guard script.

5.5.5 Trusted Build Service

To implement source based attestation, I constructed a *trustworthy build service* to build virtual machine and container images from a git source repository, and endorse the binary with a `source` assertion. The build service runs a pre-defined environment with standard build tools and takes as input the location of a code repository, including a revision hash and the build script. The build script can download standard packages from a list of trusted software repositories or compile them from source. The build service issues a `source` endorsement to bind the hash of a generated image to the source repository and revision, and then outputs the stream of image binary back to the requester.

As the fundamental piece that enables source based attestation, I believe it would be worthwhile to also connect a build service instance to its source repository. In this way, anyone can power up a trusted build service to build their own images with source endorsement. However, this is quite tricky: normal instances are endorsed by the build service for the source-image binding, but what about the first build service instance itself?

There are several options. The straightforward way is to have a human developer to manually run its build script and to endorse the generated image as an external endorser. Trust about a build service then reduces to the trust of its source repository, and the human developer (and his building environment). Similarly, an instance, launched from an image endorsed with the *builder* property by a trusted endorser, can also build the service image and endorse for its source binding. This also reduces to trusting the build service source code and a trusted endorser.

I took a different approach by applying the reproducible build technique [7]. The reproducible build technique means that running the build script using the same toolchain will always generate the same output. It removes the time and randomness from the build time variants. The benefit of using reproducible build is obvious. It is self endorsed that anyone can prove that a program hash denotes an image built from this repository. This is because the verifier can run the build and compare the output hash with the one to be verified. It is unlikely one can construct another source repository and generate the same hash. As a result, trust in the build service can be concluded by inspecting its code base, which is small and easy to trust on.

5.5.6 Summary of Latte Kubernetes

This section presents a Kubernetes cluster that can be verified in Latte for being trusted. The cluster attests running pods, seals them from administrator of both the Kubernetes cluster and the pod owner, and lastly provides additional network isolations for them. The cluster is a good starting point to run trusted applications. Using Latte Kubernetes, an authorizer only needs to verify the application in pod, and does not need to worry about the owner of the cluster due to the sealing guarantee.

Limitations In Kubernetes, a pod can originally be created with multiple containers. But these containers break the isolation boundary to share many resources such as the file system. This creates challenges for verifying the trustworthiness of a multi-container pod. In Latte Kubernetes service, each pod can have only one container, in order to eliminate such sharing. Applications of Latte are not affected by this restrictions.

5.6 Applications of Latte

In this section I will introduce how to use the Latte Kubernetes cluster as a starting point to build up other trusted applications with their code identities and properties. I assume a verified Latte Kubernetes cluster has been set up as described in §5.5. This cluster is owned by an IaaS tenant that differs from any party in below examples, i.e. the data owners, the application owners, or the IaaS provider.

5.6.1 Joint Analytics on Spark

The first application use case is a joint analytics problem: an analyst from a touring company A wants to analyze the taxi route of tourists to determine what places are more attractive. For data analytics with only one data provider, it sometimes makes sense to have the data provider to run analytic services in her own isolated environment. However, for scenarios with more parties, it can be more convenient to use code trust to approve qualified programs only.

The Scenario. An analyst from A wants to run an analytic program SpatialSpark [184], which is a Spark based big data processing framework. SpatialSpark can operate jointly on two geography databases for actions such as intersection test. The analyst would like to check the hot spots on map data, owned by a company M, with respect to the taxi route traces, owned by a company T. T and M store their dataset on different HDFS, each guarded by a HTTP authorization proxy. Both T and M may want to make sure the data are used only by good programs, since the routes may be used as auxiliary data to expose passengers privacy (consider the leakage on Netflix prize contest [116]), and fine granularity map data can contain sensitive locations such as military zones. Thus they need to specify authorization policy in each authorization proxy to check if a request from A is sent out by a trustworthy program.

SafeSpark Service To ensure the safety of computation, A uses an enhanced Spark version, called SafeSpark [132]. SafeSpark is a modified Spark platform for computation with protected sensitive data. SafeSpark implements similar functions as the Kubernetes VM in attestation, sealing and isolation for Spark jobs. SafeSpark attests each Spark job to its submitted jar files and job options. The client side job submission is modified to admit its executors into a Latte group. This group then exposes the Spark application driver's code identity to an authorizer. By design, clients of this service can only run Spark jobs, and get the results. They can not change the job configuration after it is launched, and they can not fetch/write intermediate result of a job. Cluster administrators can add or remove nodes, regulate the resource usage, and monitor job status. But they can not change a running job's configuration or read/write intermediate result. Further, there is no API for editting master/worker configuration in place. The SafeSpark service is run as multiple pods on the Latte Kubernetes cluster.

SpatialSpark. SpatialSpark encapsulates a few geographic algorithms and different parameters. The SpatialSpark application does not need any modification. It can be launched with a few options specified as command line arguments: the two datasets to be joined, the output file, the columns to

use as join condition, the operation to perform, and some algorithm specific parameters. There is no interaction between the SpatialSpark job and its submitter.

Verification of Trustworthiness. Verifying SpatialSpark requires verifying both the application and the hosting SafeSpark service. For SafeSpark, the trustworthiness is defined similar to the case of the modified Kubernetes cluster. An authorizer should verify its source code repository is SafeSpark's source, and then delegates an auditor to check the implementation about the attester properties. For SpatialSpark, authorizers need to verify the source code repository, and then check if the application uses reasonable algorithm parameters.

Guarantee about Sealed Instances. Above verification standards also imply that the owner of an instance, e.g. a SafeSpark pod, a Kubernetes VM, or a SpatialSpark executor process. To see how this happens, as a recap, firstly the Kubernetes cluster is sealed to block APIs that can change the attested code identity of a running pod and the VMs. Whether the owner of Kubernetes is trusted by data providers (M and T) and the analyst from A, he won't be able to change any attested Spark pods. In addition, the SafeSpark service also does not provide APIs to modify a running Spark job. This guarantees that VMs, pods, and Spark jobs are all properly sealed once the verfication of the Latte Kubernetes, the SafeSpark are done. It forms as the basis to authorize the SpatialSpark based on its source code and configurations.

More Assurance from Security Tools. The application scenario can be further extended with the usage of security analysis tools. For a comprehensive example, I set up a container image scanning service Clair [134] as an instance endorser. Clair runs in a TapCon instance and checks whether a client-provided image has any known vulnerabilities (CVEs) above a given severity level. If so, it issues an endorsement of the image `"no-crit-cve", 2018-5-17` to indicate no critical level CVEs were found in the CVE database as of the specified date. In the application use case, Clair scanning result is used as a proof that code of a container is not affected by critical flaws that have been revealed. Tools such as Qualys for scanning VM images [103] could be similarly adapted. The example indicates that security tools can be more involved into trust process

with Latte. Authorizers define a collection of trusted source code or image as endorsers, and can consume their endorsements database as the foothold to reason about security posture of a directly interacting instance.

Authorization Policies. I describe a set of guard policies for my Spark cluster when accessing data in HDFS. An application is deployed as attested instance, either in a container hosted by TapCon, or on Spark's worker node.

The sample guard targets a data owner who wants to limit access to an instance (i) running a known Spark distribution (ii) without known vulnerabilities (iii) configured in a closed network (iv) running an analytics program with known source and configuration to the authorizer (v) container is launched by a trusted Kubernetes cluster. I assume that the data owner *Alice* trusts that the code in the Clair source repository correctly checks for vulnerabilities in container images. Alice also expects the program to run expected ”within” range query, instead of anything else.

Listing 5.9 lists a complete guard policy Alice can put on her data to safely grant access to Spark executors. R13 extends the predicate “trustedEndoserOn” to trust an instance running the Clair source on the `"no-crit-cve"` property.

Rule R13-R18 captures the safety requirements needed to grant an executor access to data. The `safeNode` predicate requires a worker node to be admitted by a Spark master running Latte Spark, and runs locked down configuration. The Spark master is trusted to check the worker runs correct code and configuration. The `safeJob` predicate checks a driver has admitted the executor into its group . It further examines this driver runs on a worker node subject to safeNode rule, and the driver is attested by the worker to a qualified image (a .jar file endorsed by Bob). Finally, `grantAccess` combines the two rules to ensure that the requesting executor runs on a qualified worker node, and driven by a qualified application driver SpatialSpark. Check of predicate verifyKubernetesExt in R17 is optional with the assumption of a pre-verified Kubernetes cluster. This step checks if the host of the worker, i.e. the Kubernetes node VM, is admitted by a master into Kubernetes group. Predicate validKubernetesMaster has same semantic as previouse section.

Using the guard above, the owner of dataset can check the operation predicate, the broadcast algorithm, and require the output file to be put into a specified bucket for further inspection before returning to the runner of SpatialSpark.

5.6.2 Trusted Spam Filtering Service

When source code is not available for the end application (I assume sources for its running platform are still available), authorizers can still use Latte to verify the trustworthiness of an application. In this example, I consider the motivating case of spam filtering service (section 1) on my Kubernetes cluster. The filtering service runs as a container. It uses a binary image or a build repository that uses a binary executable file for the container image, so that its clients can not reason about the action of this proprietary executable. However, with provided isolation and network policy in Kubernetes, I am able to verify: a given spam filtering container is only allowed to talk to a specific IP, or even another instance with certified code properties, such like the email gateway of its client.

Listing 5.10: Policies to verify the network level isolation of a given container.

```
(F3) approvedEndpoint("192.168.1.1").

(R19) approveEmailAccess(Container) :-
        verifyKubernetesExt(Container),
        verifyNetworkIsolation(Container).

(R20) verifyNetworkIsolation(Container) :-
        \+ hasInvalidEgressEndpoint(Container),

(R21) hasInvalidEgressEndpoint(Container) :-
        hasConfig(Container, "egress", Addr),
        \+ approvedEndpoint(Addr).
```

Listing 5.10 shows the simple policies to evaluate network isolation of a container. F3 defines an approved network endpoint, which is a white list of IP address or Kubernetes labels. These endpoints descriptor defines authorizer's expectation about which endpoints this container should

Listing 5.9: Sample access policies to protect grant data access only to certain analytic jobs.

```
(R13) trustedEndorserOn(Instance, "no-crit-cve") :-
        hasProperty(Instance, "source", ClairSource).

(R14) safeNode(Node) :-
        Master: member(Node, MasterGroup),
        hasProperty(Master, "no-crit-cve", 2018-5-16),
        hasProperty(Master, "source", LatteSpark),
        hasConfig(Master, "volume", "").

(R15) safeJob(Executor) :-
        Driver: member(Executor, DriverGroup),
        approveDataAccess(Driver),

(R16) grantAccess(Executor) :-
        Worker: runs(Executor, SomeImage),
        safeNode(Worker),
        safeJob(Executor).

(R17) approveDataAccess(Instance) :-
        Worker: runs(Instance, AppJar),
        safeNode(Worker),
        hasProperty(AppJar, source, git://spatial-spark#ae2f6241),
        verifyKubernetesExt(Worker),
        hasConfig(Instance, predicate, within),
        hasConfig(Instance, broadcast, false),
        hasConfig(Instance, output, s3://monitored-buckets).

(R18) verifyKubernetesExt(Container) :-
        hasConfig(Container, host, KubeVM),
        KubeMaster: member(KubeVM, KubeMasterGroup),
        validKubernetesMaster(KubeMaster).
```

talk to. Rule R19-R21 then examines if the container runs on a trustworthy Kubernetes, and Kubernetes does not issue any network policy that allows egress traffic to endpoints outside of the approved ones. Since network policies are immutable after they create, authorizers could trust Latte Kubernetes to isolate the container from leaking the private emails it have seen.

5.6.3 Other Applications

Attestation protected credentials. I integrate attestation into OpenStack's authentication service Keystone, so that one can create a role with a guard policy specifying who can use the role. Keystone will check the guard at authentication time, and only issue an access token if the client instance passes the guard. This allows credentials to be bound to a specific software stack or even source repositories, so that only instances using images built from the required repository can authenticate with the role. For example, to evaluate the scenario, I defined policies that only grant access to a utility container. Only the utility container can then authenticate itself to use the distributed credentials . With such capability, even if *the credential is lost*, anyone trying to use the credential must run in the exact same software stack.

Database protection. I implement connection-based protection in mysql-router proxy for MySQL [126]. The proxy loads a guard and verifies that all incoming connections are from approved instances. This can be used to harden the database port so it can only be accessed from certain instances.

5.7 Implementation of Latte

The mandatory components of Latte framework consist of the metadata service and the client library.

5.7.1 Metadata Service

The Latte metadata service (MDS) stores statements as objects in a key-value store indexed by the subject of the statement: instances (attestations) or code objects (endorsements). To accelerate

fetching all the statements needed for authorization, the MDS automatically links instances to related statements, such as their launching instance and image. The Latte library defines an interface for authorizers to fetch the transitive closure of all statements pertaining to an instance.

The MDS is structured as a *front-end* that implements the client API and metadata management, and a scalable *back-end* storage service. The front-end is largely stateless and can be replicated for scalability. For the back end I use Riak [164], a distributed key-value store that is fault tolerant and scalable.

Network authentication and control. The MDS internally uses PIDs to refer to instances, and stores a map from network address to PID. All access to the MDS is authenticated by address using this map. To allow address delegation, an instance may be created with a range of addresses, which are passed to the MDS in CIDR format [64]. When an instance delegates addresses using the `bindToID` statement, the MDS verifies that the issuer controls the addresses (i.e., it is bound to a range including the addresses). The MDS uses a hash map and interval tree to cache the mapping of IP and port ranges to instance PIDs.

Caching and consistency. The front end of the MDS caches recently accessed statements. Most statements are immutable, and only need to be evicted when an instance is deleted. Similarly, the back end can use eventual consistency for most data: if a statement is missing it may temporarily cause a guard to fail, but retrying authorization will eventually fetch any missing statements. The one piece of data that requires strong consistency is the map of network addresses to PIDs: if an address is reused and an MDS is unaware, it may fetch statements for the previous instance using the address. I require strong consistency for `bindToID` statements. In addition, the mds assigns a time-to-live, so that cached values will be re-fetched periodically.

Garbage collection. Statements are garbage collected automatically when their subject is defunct. An instance is defunct when it has terminated *and* the subjects of any statements it has issued are also defunct. An image is defunct when it has been deleted from the MDS *and* any instances launched from that image are defunct. My prototype does not permit statements about a live subject to be withdrawn.

5.7.2 Latte Library

The Latte Library comprises two parts: an API for issuing attestations and endorsement statements and an API for authorization. Statements are marshaled as JSON requests and sent to the MDS using HTTP. The authorization functionality runs in a separate container, which which the library communicates via local RPC.

I implement authorization using SAFE [46], which uses the Styla Datalog engine [161] to check guard policy. The library takes as input a requester's network address, and contacts the MDS to fetch statements about the requester. Authorizers can also pass additional endorsements to consider, but the authorizer must verify the endorsements' authenticity. The implementation caches statements using the same rules described above.

5.7.3 Port range management

I implemented a port management extension in Linux Kernel. It ties a list of ports to each process. A process can only use associated ports (explicitly or implicitly) for any socket communication. A process is also allowed to delegate its own port range to its children processes, after which the parent can no longer use these ports until the children terminate.

5.7.4 Total Effort

Latte: Attestation and guard libraries comprise 4322 lines of C++ and the metadata service took 1761 lines of Go and 959 lines of python. I link against SAFE.

Attesters: I reused code from CQSTR and TapCon comprising 6000 lines of Python and Go, and 852 lines in the Linux kernel. Kubernetes is modified with 973 lines of Go and a few Python scripts. Spark changes required 268 lines for adding attestations and authorizations based on SafeSpark version, and a 133 lines wrapper in C to enforce port usage using my added system calls.

Authorizers: HDFS changes are 480 lines, the MySQL router took 278 lines, and I added 20 lines in CQSTR's version of Keystone. There are in total 710 lines for all guards.

Endorsers: The build service is about 500 lines, and the Clair scanner is 172, both written in Go.

5.8 Evaluation

The costs of Latte come from (i) issuing attestations during instances startup and (ii) evaluating guards for authorization. I evaluate these costs separately, as well as the overall performance overhead on applications

5.8.1 Evaluation Setup

I evaluate Latte on a 6-node cluster on CloudLab [1]. Each node has 20 Intel E5-2660 cores, 160GB memory, and two 10GbE NICs. The cluster runs OpenStack and Docker with modifications for Latte. I use four compute nodes for OpenStack, one for a network gateway, and one for the cloud controller. The metadata service's frontend runs on one compute node, and backend storage runs on the other three compute nodes. The 4 compute nodes are also used for applications such as Spark. I configure VMs with 4 VCPUs and 16GB memory. The Kubernetes cluster has five VMs, one master and four node VMs.

5.8.2 Attestation Cost

The metadata service introduces the dominant overhead of attestation, as it must be contacted to issue and delete statements when instances start and stop.

Methodology I simulate parallel instance startup by running driver programs on multiple machines that issue the attestations needed to start VM, container, and process instances. I evaluate the cost for deployments using 1 (VM only) , 2 (VM + container), and 3 (VM + container + process) layers. The table below shows the number of instances at each level for each configuration.

Layers	VMs	Containers	Processes	Total
1-layer	204,800	0	0	204,800
2-layer	4096	50	0	204,800
3-layer	1024	50	4	204,800

The VM posts 10 configuration statements for each container. I run this experiment using 32, 128, and 256 threads.

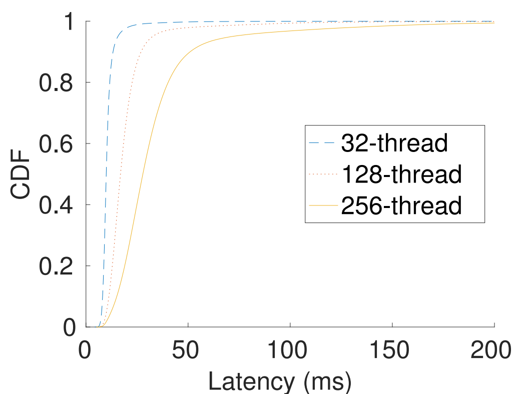
I measure the total time to post all statements and report on the throughput and latency from Figure 5.6a to 5.7. I separately measure the latency and throughput of fetching all the statements for each instance.

Figure 5.6f shows the throughput for instance creation and fetch for a single MDS with 3 backend Riak servers, which store data on a SSD. With 32 threads, there is not enough parallelism to saturate the MDS, so throughput is lower. With 256 threads, throughput reaches its peak at 2500 create operations/sec and 3600 fetches/sec. Figure 5.6a shows a CDF for fetch request latency, and shows that latencies are generally below 50ms. With fewer threads, there is less queuing in the MDS and hence lower latency. Figure 5.6f shows the impact of layering on throughput. Layering increases latency slightly, as it requires more linking operations and fetches must return more statements. Overall, these results indicate that a single MDS is able to handle thousands of instance creations per second, which is suitable for a large network. Latency for issuing creation statements is generally much lower than the time to start a VM or container, although they may slowdown launch of very small processes.

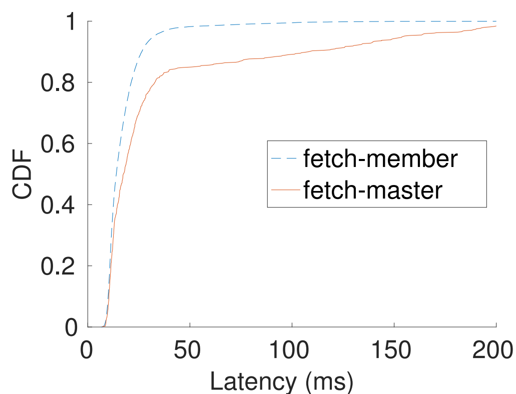
The above results look only at cost of accessing the MDS. On an authorizer, caching can substantially reduce costs. Figure 5.6c shows the impact of caching statements and network address-to-instance mappings. Overall, performance without caching statements is 100x slower (not shown), and without caching network addresses is 5x slower.

Figure 5.6b shows the impact of group membership. I configure a master to create groups of 30 instances and measure the latency to fetch statements for the group master and members. Being a member has little impact on fetch latency, as it only incrementally adds to the number of statements. Fetching statements for a master, though, returns all the member statements and hence leads to larger tail latencies: 20% of fetch requests for a group master take longer 40ms.

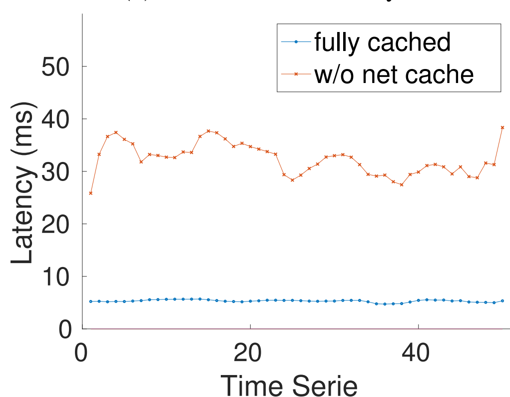
Space usage is very low: each instance or image takes less than 1KB, so even for large networks all data can be kept in memory.



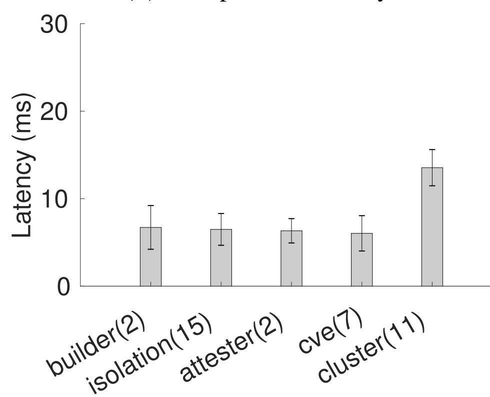
(a) Normal Fetch Latency



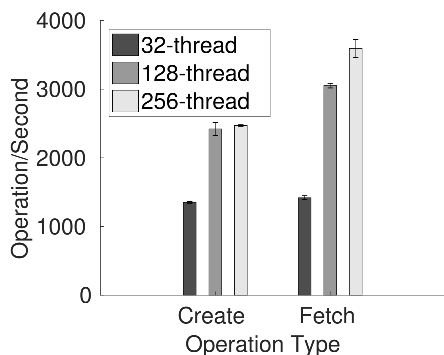
(b) Group Fetch Latency



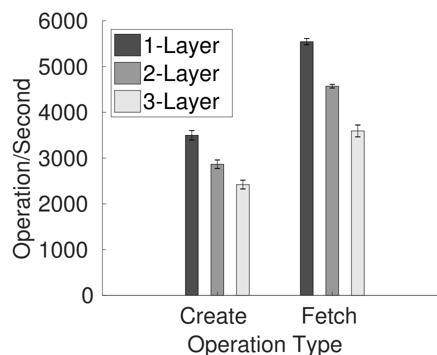
(c) Caching Effect



(d) Time to Check Guards



(e) Throughput



(f) Throughput on Layering

5.8.3 Authorization Cost

I measured the latency to evaluate guard predicates described in previous sections on containers (2 levels) and Spark executors (3 levels). For `builder` I check the build service instance. For `attester` I check a container. For the `cve` I check a single guard (adding more did not change

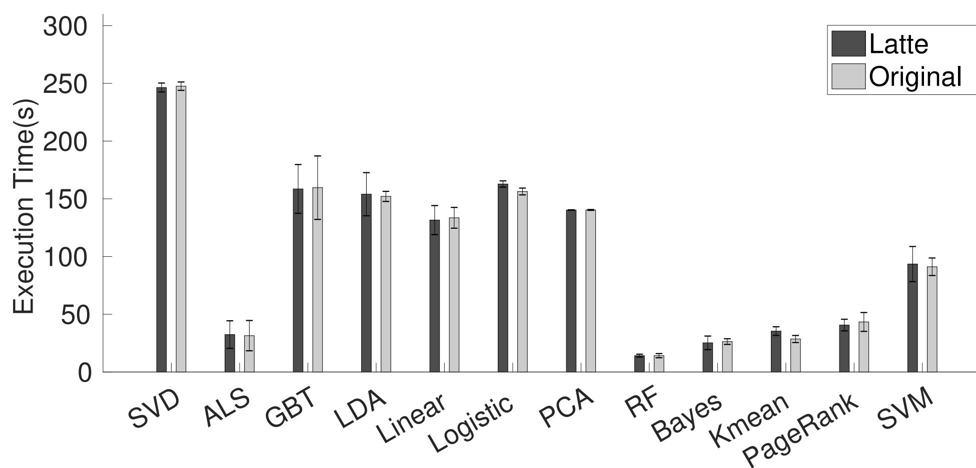


Figure 5.7: HiBench Spark Suite

the results). The isolation guard add more isolation conditions to the one in Listing 5.4. And the cluster guard refers to Listing 5.9.

Figure 5.6d shows the average latency and standard deviation for each of these guards when cache hits. Cache miss effect is consistent with Figure 5.6c. The number of rules is shown after the guard's name. Overall, guards took between 5-10ms except for the cluster guard, which took 13ms. The time for logical inference alone is 1ms for most guards, and 8ms for the cluster guard due to its complexity (isolation guard has longer rules but fewer statements involved and used). The time above inferences comes from unoptimized guard parsing. To put these time in perspective comparison, AWS S3 access latencies are generally higher than 10ms. In the case of coarse-grained access, such as connecting to a database, the authorization check is only needed once. For storage services, the results of authorization can be cached and used for any object with the same guard (not implemented). Besides, the IaaS network can implement a time to live counter for each tenant. The counter indicates the next time to reuse an IP address that was previously applied to another instance. Caching statements in the TTL window is an interesting optimization.

5.8.4 Applications

Finally, I evaluate application performance. For SpatialSpark, I used new york taxi routes [163] and district block [162]. The performance does not differ much. SpatialSpark on original platform

runs for 703.2 second and 712.3 second for modified Kubernetes. The input data contains about 2 million points of taxi pickup location, while the census block is mostly static, about 23MB. For the email filtering, it takes 595 second on original Kubernetes, and 613 second for Latte Kubernetes. For Spark, I use Intel's HiBench [89] bigdata benchmark, and compare SafeSpark and HDFS performing authorization against their native counterparts. The execution time of "Large" dataset is shown in Figure 5.7. Overall, Latte performed identically to the native system. This is explained by the nature of analytics jobs, which are not storage bound.

For Keystone, I measure the latency to obtain the credential and use it with an unmodified OpenStack Swift storage service. Latte added 40ms, which is a 6% slowdown on small object access and less for large objects. Similarly, I compare the result of an OLTP benchmark dbt2 [127] with modified MYSQL proxy and no proxy case. The transactions per minute reduced by 2% with Latte, within the standard deviation of 3%.

5.9 Summary

Many computing settings require high assurance in the code being run, which cannot be provided by current cloud computing systems. I propose that code attestation is a suitable primitive for establishing this trust. I show how it can be applied for flexible authorization policies. While Latte roots its trust in the IaaS platform, the same architecture can be used with trusted hardware such as SGX, by seeding the metadata service with statements that the processor is trusted.

Chapter 6

Discussion

To verify the Latte's design is compatible with current engineering practices, I conducted an end user survey as shown in table 6.1. The sample size is small (37) but each sample reflects a different company. The result supports my assumptions in section 1.2.

First of all, the public continuous integration and continuous delivery (CI/CD) tools are used by approximately one third of the respondents. Latte relies on public CI/CD tools as a foundation (builder instances in section 5.4.4) for source based attestation, and source based attestation is important to the overall usefulness of code based authorization for third parties. For CI tools, which is most relevant for code building, a different market report [52] shows public tools like Jenkins [90] are leading in the total share.

Secondly, service providers already adopt the immutable infrasturcture practice. In my survey, about 20% of the participants agree that administrators do not log into production machines. For containers, almost 80% of the companies who uses containers run immutable containers, and launch new containers on configuration changes. This means the way that these administrators manage their services is compatible with Latte's design about sealed instances (which has enforced immutability). Although there are still many administrators (70%) log into productions machines, the number is likely to shrink in future. Based on another report from Sonatype [159] for about 1400 respondants from companies with mature DevOps practices and companies currently trying to improve DevOps operations (these company occupies 77% of all companies), 72% of them agree on usage of immutable infrastructure.

Lastly, my survey shows that 92% of the respondents use open source projects world services. Additional reports from other sources reveal the similar fact [123, 124]. At the same time, code

Questions	Responses (Percentage)
Do your company use public clouds?	Yes (71%) No (29%)
Do your company use public CI/CD tools	Yes (35%) No (65%)
When do your admin log in to production machines	Debugging (70%) Reconfiguring (51%) Monitoring (43%) Admins do not log in(19%) Other (3%)
Do your company use containers? If yes, how do you update container configurations?	We launch new containers (54%) We update containers in place (16%) We do not use containers (30%)
How and why will you trust third party service for private data?	We remove secrets first (59%) We require compliance certificates(56%) The provider has a good reputation (44%) The provider has a reasonable agreement (41%) Other (5.9%)
If you are to provide services to others, will you use auditing services and how?	We have auditors check our system behaviors(81.5%) We have auditors check our system configuration (48.1%) We have auditors interview our staff (33.3%) We have auditors check our source code (25.9%) Others (7.4%)
Do you use open source project, and how?	We use LTS releases of open source projects(32.4%) We maintain our private repository of open source projects (59.5%) We don't use open source project (8.1%)

Table 6.1: Survey about how end users/operators set up the cloud services. There are 37 samples coming from different organizations.

auditing is already happening at service providers' (25%), and more than 80% of companies would have auditors to test their system behaviors. These numbers directly support my assumption about widely available endorsements and source based properties, which are used extensively in Latte authorization.

In general, the survey indicates that Latte is already practical to benefit a fraction of companies. As modern DevOps and containerization are becoming popular, this fraction is likely to grow higher in near future.

Chapter 7

Lessons and Future Works

7.1 Lessons

My research covers a broad range of materials about public clouds, from the underlying IaaS infrastructure to end user applications. I spent quite some time exploring the proper abstraction and design for the Latte framework. Working with complex systems are challenging, but also fun. I would like to discuss about the lessons learned during this process.

Two years to one week. To conquer Openstack, the first big system I touched during my Ph.D study, I spent two years from the fall of 2013 to the summer of 2015 to finish the coding. A few months ago, to get Spark working, I spent roughly a week: five days to read the source code, and two days to implement what is needed. Granted, the lines of code are different, but to myself this really explains how valuable the experiences in system research are. It took me a very long time to get familiar with one system, but later on things are becoming easier and easier, as complex systems share many faith in their designs. For example, a golden rule for me to play with these systems are to dig out the complete flow about how a client interacts with the server. It works on Openstack, on Docker, on Linux kernel, on Kubernetes, on Spark, and on HDFS. Before I started the research, I never imagined that I am going to face these many complex systems. But it turns out, as long as you are not struck down by the difficulties, you can always see the sun rising.

Stay organized. When things grow big, it requires better organization. This is a painful lesson I learnt from this research. Even now I can not tell how many times I can not remember where to find a useful document I created five months ago. I also can not tell how many times in discussion

with my adivsor and other collaborators we circled back to old problems that have been discussed over and over just because the discussed materials are not well recorded. If I could do it again, I would like to review more to have clearer records about the research, and avoid wasting times in matters like this.

Don't assume too much about your audiences. During my Ph.D life in Madison, I find one of the most difficult things is to explain the idea in my mind to others. Not everyone has spent the same amount of time in the fields as I do, thus it is very important to explain necessary background, and it is also critical to stay away from messy details. As a geek, the fun comes from hacking, and it is always tempting to explain how every byte in my masterpiece means. However, this distracts the audiences. They can not see the same picture as I do. What helps more is the high level picture, i.e., what is the problem, why you do it, and what outcomes do you achieve. This is one of the biggest take aways for the Ph.D study.

Applications come first. Find a nail first, instead of holding a hammer and run desperately to find a nail. As a system developer, it is tempting to build a big system first, and then think about who can use it. It's wrong. In fact, without reasonable applications to demonstrate the values of my systems, it can hardly get recognized by others for being useful. Meanwhile, application scenarios serve great to guide optimizations and focuses of the underlying support systems. Without proper restrictions such as application demands, for countless times I would fall into the trap of creating a big pile of junk that tries to address every problem in system design. It is not how things work. The lesson here is to talk to other people more, to connect with application developers to understand what is really needed before starting to write the code.

Learn from others. Every Ph.D is an expert in certain field. It is really beneficial to talk more with collaborators, lab mates, and people I met during different events. For example, originally I did not understand logics at all. I thought logic languages were no different than daily programming languages that I used. But as I talk more with my collaborators, Jeff and Qiang, both experts in logics, I realized that I was quite wrong before. Unlike common tools I used, logics

have stronger mathematical background. Designs using different logic language can have different implications and properties. From Jeff and Qiang I learned a lot, and eventually I am able to finish Latte with logics as the core element.

7.2 Future Works

There are a few research directions enabled or based on Latte. One category of opportunities are to further extending Latte, such as extending to multiple cloud, working with secure hardware. Another type of research is to use Latte for building trusted applications, such as using Latte to defend against side channel attacks.

7.2.1 Secure Hardware and Multi-Clouds

In this dissertation I just studied the case with one cloud provider. What becomes interesting is that how secure hardware can connect with the software trust, and in addition, how to address the scenario where multiple clouds are involved.

The logical structure of Latte has no problem adapting multiple root of trusts. The definition of `attester` property can easily cover the case where multiple clouds and/or a special enclave ceritificate are defined as trust anchors, so that whenever an instance is checked for attestations, as long as its host is verified to be an attester, then nothing is different from the case of a single cloud. For SGX secure enclaves, it may need a different protocol to issue statements though, as an enclave won't actively talk to the MDS of Latte. Moreover, the hosting environment of an enclave can also issue statements about the enclave, which provides a link for an authorizer to also check its statements when an enclave application makes requests.

However, there are a few research questions that may be valuable to answer. The first is how to manage the MDS when there are multiple cloud providers. Authorizers may not trust all the cloud providers, thus it can not take the Latte approach to let a single cloud provider manage this service. Should it be delegated to a trusted third party, or should it be handled by each cloud provider? A related question is how to use decentralized storage. For example, can each attester manage its own

statements? In this way single point failures can be avoided. But how can an authorizer effectively fetch statements of a layered instance in this case?

Another question is how to design trusted applications with both public clouds and SGX. Current SGX usage scenarios often assume a powerful adversary like cloud provider. This can make design ultra complex to properly handle the secrets, and it needs extensive work to recreate existing wheels again in SGX flavor. But with Latte, can the design be simplified? What is the trust implication when a SGX application also trusts the underlying cloud provider?

7.2.2 Colocation and Side Channel Attacks

Latte makes it possible to enumerate any related software through logical links. For example, from a requesting instance, it is also possible for an authorizer to retrieve statements about any colocated instances. Such capability makes Latte a promising candidate to address side channel attacks. To adapt Latte for such usage, there are two possible factors to consider.

The first factor is the scheduling. An instance may declare a policy for preference of colocated instances. For example, an executor of a Spark job may prefer to stay closer to a HDFS data node, while trying not to compete with a peer executor. The question to ask here is how to use such information to pack and schedule workloads. Is it fair and responsive? Can it prevent DoS attacks that try to drive other workload off? And what is the performance implication?

The second is to think about how to logicalize and describe a co-location attacker with logic, so that an authorizer can use to construct preference above. What should be used to measure an attacker? Does performance profile work for this purpose? Can we apply machine learning in this scenario to identify side channel attackers? What is the chance for false positive? Typical solutions can introduce large overhead, does an approach based on code trust scale?

7.2.3 Network Authentication and Layered Networks

Beyond the IaaS network, there can often be tenant managed network, which uses kernel routing table or IaaS assisted routing such as Google Cloud Platforms advance routing mechanism to direct application traffic. An example is the cluster network of Kubernetes, which provisions a flat

network for all the pods on this network. Such networks often use source NAT to initiate client traffic, so that the outgoing packet uses the virtual machine IP (or even through a second NAT). To make network authentication work, I chose a different way to set up such network in Latte, i.e., to allocate an unique IaaS network address to each container. The address is assigned by the VM instance. To enable such functionality, IaaS provides statements to certify that a VM for owning certain IPs. In addition, the IaaS provider either reserves a block of IP addresses for each VM, or IaaS maintains a DHCP service that allows a VM to request for multiple IP addresses. Both of the approaches for provisioning additional IP blocks require enough IP addresses, which again calls for the usage of IPv6 network. The question to ask here is: is it possible to isolate the cluster network, i.e., not exposed on the public network, while allowing the instances on cluster network to be attested and verified uniformly as the other instances on the IaaS network?

7.2.4 High Order Logic Trust

The configuration of applications can be very complex. For example, on Kubernetes each pod is associated with a file, which can contain more than 30 properties. For a software stack that I used for joint analysis, there can be even more options. How to understand and verify these configuration for trust in general remains an open question. On the other hand, the policy I used so far is still first order logic. This means it can not effectively handle configurations that defines functions, such as a guard policy to check usage of another guard. For these purposes, which is the best logic for verifying a configuration? How to use it to check these configurations?

Chapter 8

Conclusion

Securely sharing private data across tenants is becoming more important in current public clouds. To enable trustworthy collaborations on private data, it is necessary for the collaborators to authorize access from other parties faithfully. Current approaches to establish trust are inaccurate, slow, inflexible, and lack defense in depth. Software trust is the key to help address these problems, and the central question of adapting software trust is to adapt code identities of cloud instances.

In this dissertation I presented the authorization framework Latte, which securely leverages code identities to help authorizers to assess the trustworthiness of a requester. Latte is a general framework that works for all layers of public clouds. Using Latte, an authorizer can securely learn about the entire running environment of the requester. Its usage of logic provides a powerful tool for authorizers to reason about security properties of a remote requester.

The mechanisms in Latte capture different aspects of code trust; Layered execution makes it possible to reason about the software stack of a running application; Source based attestation attaches the source repository URL to generated images, offering a better place to verify the code for trust; Group mechanism gives authorizers an approach to delegate verification tasks to a group master; Endorsers can certify properties on code repository, which can be later manipulated by an authorizer in a guard policy. To make Latte work, I implemented necessary changes on Openstack(CQSTR), Docker(TapCon), Kubernetes, etc to provide examples of trusted software, and authorize these trusted software.

With Latte, it is easier to run trusted software, and allows a different organization to check and trust them regardless of who owns and runs them. This enables trusted collaborations that we seek in the first place.

LIST OF REFERENCES

[1] Cloudlab. https://www.cloudlab.us.

[2] Email data set. https://aws.amazon.com/datasets/917205.

[3] Ists competition 12. http://www.netresec.com/?page=ISTS.

[4] Linux containers. https://linuxcontainers.org/.

[5] Linux containers. http://linuxcontainers.org/.

[6] Openstack trusted computing pool. https://wiki.OpenStack.org/wiki/TrustedComputingPools.

[7] Reproducible build project. https://reproducible-builds.org/.

[8] Martín Abadi and Boon Thau Loo. Towards a declarative language and system for secure networking. In *NetDB*, 2007.

[9] Jay Aikat, Aditya Akella, Jeffrey S Chase, Ari Juels, Michael K Reiter, Thomas Ristenpart, Vyas Sekar, and Michael Swift. Rethinking security in the era of cloud computing. *IEEE Security & Privacy*, 15(3):60–69, 2017.

[10] Amazon Web Service. AWS Code Commit. https://aws.amazon.com/codecommit.

[11] Amazon Web Service. AWS Code Pipeline. https://aws.amazon.com/codepipeline.

[12] Aws compliance. https://aws.amazon.com/compliance/.

[13] Amazon Web Service. Life Without SSH: Immutable Infrastructure in Production. https://www.slideshare.net/AmazonWebServices/aws-reinvent-2016-life-without-ssh-immutable-infrastructure-in-production-sac318.

[14] Amazon Web Services. Vpc security capabilities. https://aws.amazon.com/answers/networking/vpc-security-capabilities/.

[15] Amazon Web Services. Amazon web services: Overview of security processes, 2014. available at http://www.utdallas.edu/~muratk/courses/cloud11f_files/AWS_Security_Whitepaper.pdf.

[16] Inc. Amazon Web Services. Aws identity and access management (iam). https://aws.amazon.com/iam/.

[17] Amazon Web Services, Inc. Amazon cloudtrail. https://aws.amazon.com/cloudtrail/.

[18] Amazon Web Services, Inc. Amazon cloudwatch. https://aws.amazon.com/cloudwatch/.

[19] Amazon Web Services, Inc. Amazon machine learning. https://aws.amazon.com/machine-learning/.

[20] Amazon Web Services, Inc. Amazon virtual private cloud. https://aws.amazon.com/vpc/.

[21] Amazon Web Services, Inc. AWS api gateway. https://aws.amazon.com/api-gateway/.

[22] Amazon Web Services, Inc. AWS config. https://aws.amazon.com/config/.

[23] Amazon Web Services, Inc. Ipv6 support for ec2 instances. https://aws.amazon.com/blogs/aws/new-ipv6-support-for-ec2-instances-in-virtual-private-clouds/.

[24] Amazon Web Services, Inc. VPC endpoints. http://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/vpc-endpoints.html.

[25] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative Technology for CPU-based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, volume 13, 2013.

[26] David G. Andersen, Hari Balakrishnan, Nick Feamster, Teemu Koponen, Daekyeong Moon, and Scott Shenker. Accountable Internet Protocol (AIP). In *the ACM SIGCOMM Conference on Data Communication*, pages 339–350, 2008.

[27] Ansible is simple it automation. https://www.ansible.com.

[28] Apache Foundation. Spark. https://spark.apache.org/.

[29] Apache Foundation. Welcome to apache pig. https://pig.apache.org/.

[30] Apache Software Foundation. Prediction io. http://prediction.io.

[31] Apparmor. https://wiki.ubuntu.com/AppArmor.

[32] Andrew W Appel and Edward W Felten. Proof-carrying authentication. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, pages 52–62. ACM, 1999.

[33] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel OKeeffe, Mark L Stillwell, et al. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[34] Philippe Arteau. Find security bugs. https://find-sec-bugs.github.io/.

[35] Lessons on ashley madison data breach. https://www.pandasecurity.com/mediacenter/security/lessons-ashley-madison-data-breach/.

[36] Asylo. An Open and Flexible Framework for Enclave Applications. https://asylo.dev, 2018.

[37] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, pages 267–283, 2014.

[38] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. The million song dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[39] Karthikeyan Bhargavan, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, Santiago Zanella-Beguelin, and Cdric Fournet. Proving the TLS Handshake Secure (As It Is). In *Advances in Cryptology – CRYPTO 2014*, pages 235–255, July 2014.

[40] Bigml. http://bigml.com/.

[41] Arnar Birgisson, Joe Gibbs Politz, Ulfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *NDSS*, 2014.

[42] Nikita Borisov, Rob Johnson, Naveen Sastry, and David Wagner. Fixing races for fun and profit: How to abuse atime. In *Proceedings of the 14th Conference on USENIX Security Symposium*, pages 20–20, 2005.

[43] Benjamin Braun, Ariel J Feldman, Zuocheng Ren, Srinath Setty, Andrew J Blumberg, and Michael Walfish. Verifying computations with state. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 341–357, 2013.

[44] Andrew Brown and Jeffrey S Chase. Trusted platform-as-a-service: a foundation for trustworthy cloud-hosted applications. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 15–20. ACM, 2011.

[45] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, page 991–1008, Baltimore, MD, 2018. USENIX Association.

[46] Qiang Cao, Vamsi Thummala, Jeffrey S. Chase, Yuanjun Yao, and Bing Xie. Certificate Linking and Caching for Logical Trust. `http://arxiv.org/abs/1701.06562`, 2016. Duke University Technical Report.

[47] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE transactions on knowledge and data engineering*, 1(1):146–166, 1989.

[48] Ramaswamy Chandramouli, Michaela Iorga, and Santosh Chokhani. Cryptographic key management issues and challenges in cloud services. In *Secure Cloud Computing*, pages 1–30. Springer, 2014.

[49] Stephen Checkoway and Hovav Shacham. *Iago attacks: why the system call API is a bad untrusted RPC interface*, volume 41. ACM, 2013.

[50] Chef automation. `https://www.chef.sh`.

[51] Liqun Chen, Rainer Landfermann, Hans Löhr, Markus Rohe, Ahmad-Reza Sadeghi, and Christian Stüble. A protocol for property-based attestation. In *Proceedings of the first ACM workshop on Scalable trusted computing*, pages 7–16, 2006.

[52] Best Integration Software. `https://www.g2crowd.com/categories/continuous-integration?order=popular`.

[53] cilium.io. API-aware Networking and Security. `https://www.cilium.io/`.

[54] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In *International Conference on Theorem Proving in Higher Order Logics*, pages 23–42, 2009.

[55] Wei Dai, Yarkın Doröz, and Berk Sunar. Accelerating ntru based homomorphic encryption using gpus. Cryptology ePrint Archive, Report 2014/389, 2014. `https://eprint.iacr.org/2014/389`.

[56] Deep ai. `https://deepai.org/enterprise-machine-learning`.

[57] Deep cognition. https://deepcognition.ai/.

[58] John DeTreville. Binder, a logic-based security language. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 105–113, 2002.

[59] The 2018 DevOps Pulse Survey Results. https://dzone.com/articles/the-2018-devops-pulse-survey-results.

[60] Docker. https://www.docker.com/.

[61] Docker. Docker Notary. https://docs.docker.com/notary/.

[62] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the Third Conference on Theory of Cryptography*, pages 265–284, 2006.

[63] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazieres, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. *ACM SIGOPS Operating Systems Review*, 39(5):17–30, 2005.

[64] Justin Ellingwood. Understanding IP Addresses, Subnets, and CIDR Notation for Networking. https://www.digitalocean.com/community/tutorials/understanding-ip-addresses-subnets-and-cidr-notation-for-networking, March 2014.

[65] Centers for Disease Control, Prevention, et al. Hipaa privacy rule and public health. guidance from cdc and the us department of health and human services. *MMWR: Morbidity and mortality weekly report*, 52(Suppl. 1):1–17, 2003.

[66] Bryan Ford and Russ Cox. Vx32: Lightweight User-Level Sandboxing on the x86. In *USENIX Annual Technical Conference*, pages 293–306, 2008.

[67] The Apache Software Foundation. Spamassassin. http://spamassassin.apache.org/.

[68] Keke Gai and Meikang Qiu. Blend arithmetic operations on tensor-based fully homomorphic encryption over real numbers. *IEEE Transactions on Industrial Informatics*, 14(8):3590–3598, 2018.

[69] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proc. Symposium on Operating Systems Principles*, 2003.

[70] the gnu compiler collection. http://gcc.gnu.org/.

[71] Roxana Geambasu, Steven D. Gribble, and Henry M. Levy. Cloudviews: Communal data sharing in public clouds. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, 2009.

[72] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[73] Github. https://github.com/.

[74] Gmail. https://mail.google.com/.

[75] Google Inc. Google Cloud Alias IP Range Overview. https://cloud.google.com/vpc/docs/alias-ip.

[76] Google, Inc. Google prediction api. https://cloud.google.com/prediction/.

[77] Google, Inc. Kubernetes. https://kubernetes.io/.

[78] HashiCorp. HashiCorp Vault: A Tool for Managing Secrets. https://www.vaultproject.io/.

[79] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, pages 1–17, 2015.

[80] Chris Hawblitzel, Jon Howell, Jacob R Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, pages 165–181, 2014.

[81] Keqiang He, Alexis Fisher, Liang Wang, Aaron Gember, Aditya Akella, and Thomas Ristenpart. Next stop, the cloud: Understanding modern web service deployment in ec2 and azure. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 177–190. ACM, 2013.

[82] Heroku addons. https://elements.heroku.com/addons.

[83] Galen Hunt and Jim Larus. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review*, pages 37–49, April 2007.

[84] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proceedings of the 12th USENIX Symposium Operating Systems Design and Implementation*, 2016.

[85] Google Inc. Shielded vms on google cloud platform. https://cloud.google.com/shielded-vm/.

[86] IBM inc. Is your docker container secure? ask vulnerability advisor. https://www.ibm.com/blogs/bluemix/2015/07/vulnerability-advisor/.

[87] Sysdig Inc. Sysdig falco. https://sysdig.com/opensource/falco/.

[88] Tigera Inc. Calico. https://www.projectcalico.org/.

[89] Intel Corp. Intel hibench suite. https://github.com/intel-hadoop/HiBench.

[90] Jenkins. jenkins.io.

[91] Trevor Jim. Sd3: A trust management system with certified evaluation. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 106–115, 2001.

[92] Charles Killian, James W Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, 2007.

[93] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[94] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. *ACM SIGOPS Operating Systems Review*, 41(6):321–334, 2007.

[95] Kubernetes cluster networking. https://kubernetes.io/docs/concepts/cluster-administration/networking/.

[96] Ang Li, Xin Liu, and Xiaowei Yang. Bootstrapping accountability in the internet we have. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, page 155, 2011.

[97] Ninghui Li and John C. Mitchell. Datalog with Constraints: A Foundation for Trust Management Languages. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, PADL '03, 2003.

[98] Yanlin Li, Jonathan M McCune, and James Newsome. MiniBox: A Two-Way Sandbox for x86 Native Code. In *Proceedings of the Usenix Annual Technical Conference*, 2014.

[99] Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. Modist: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, 2009.

[100] Yehuda Lindell and Benny Pinkas. Secure multiparty computation for privacy-preserving data mining. *Journal of Privacy and Confidentiality*, 1(1):5, 2009.

[101] Linq (language-integrated query). `https://msdn.microsoft.com/en-us/library/bb397926.aspx`.

[102] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, pages 605–622, 2015.

[103] Justin Lute. Qualys virtual scanner appliance. `https://community.qualys.com/docs/DOC-3452-reference-qualys-virtual-scanner-appliance)`.

[104] Haohui Mai, Edgar Pek, Hui Xue, Samuel Talmadge King, and Parthasarathy Madhusudan. Verifying security invariants in expressos. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 293–304, 2013.

[105] William R. Marczak, David Zook, Wenchao Zhou, Molham Aref, and Boon Thau Loo. Declarative Reconfigurable Trust Management. *Computing Research Repository*, September 2009.

[106] Maste card. `https://www.mastercard.us/en-us.html`.

[107] Apache maven project. `http://maven.apache.org/`.

[108] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328, 2008.

[109] Frank Mcsherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 19–30. acm, 2009.

[110] Microsoft Corp. Caching access checks. `https://msdn.microsoft.com/en-us/library/windows/desktop/ff394767(v=vs.85).aspx`.

[111] Microsoft Corp. Guarded Fabric and Shielded VMs on Azure. `https://docs.microsoft.com/en-us/windows-server/virtualization/guarded-fabric-shielded-vm/guarded-fabric-and-shielded-vms-top-node`.

[112] Andrew G Miklas, Stefan Saroiu, Alec Wolman, and Angela Demke Brown. Bunker: A privacy-oriented platform for network tracing. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, pages 29–42, 2009.

[113] Prateek Mittal, Vern Paxson, Robin Sommer, and Mark Winterrowd. Securing mediated trace access using black-box permutation analysis. In *HotNets*. Citeseer, 2009.

[114] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proc. Symposium on Operating Systems Principles (SOSP)*, 1997.

[115] Mysql database. `https://www.mysql.com`.

[116] Arvind Narayanan and Vitaly Shmatikov. How to break anonymity of the netflix prize dataset. *arXiv preprint cs/0610105*, 2006.

[117] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an os kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 252–269, 2017.

[118] B Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications magazine*, 32(9):33–38, 1994.

[119] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73, March 2015.

[120] OASIS. Saml2.0. `http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0.html`.

[121] Tj OConnor, William Enck, W. Michael Petullo, and Akash Verma. Pivotwall: Sdn-based information flow control. In *Proceedings of the Symposium on SDN Research*, SOSR '18, pages 3:1–3:14, New York, NY, USA, 2018. ACM.

[122] Openid. `https://openid.net/`.

[123] Open Source is Everywhere. `https://blog.tidelift.com/open-source-is-everywhere-survey-results-part-1`.

[124] Open Programs are a Best Practice Among Large Companies. `https://thenewstack.io/survey-open-source-programs-are-a-best-practice-among-large-companies/`.

[125] Openstack. `http://www.OpenStack.org/`.

[126] Oracle Corporation. Mysql-router. `https://dev.mysql.com/doc/mysql-router/2.1/en/`.

[127] Oracle Inc. Mysql benchmark tools. `https://dev.mysql.com/downloads/benchmarks.html`.

[128] Packetloop. Packetpig. `http://blog.packetloop.com/search/label/packetpig`.

[129] PCI Security Standards Council. Official PCI Security Standards. https://www.pcisecuritystandards.org/document_library.

[130] Private biometrics. https://www.privatebiometrics.com/.

[131] Puppet: Let's talk automation, devops and cloud. https://puppet.com.

[132] Safespark. https://users.cs.duke.edu/~qiangcao/publications/safespark.pdf.

[133] Rackspace. OpenStack Congress. https://wiki.openstack.org/wiki/Congress.

[134] Red Hat, Inc. Coreos clair. https://coreos.com/clair.

[135] Red Hat, Inc. Project Atomic. https://www.projectatomic.io/.

[136] IBM X-Force Research. Ibm x-force threat intelligence index 2017, 2017.

[137] Bruno F Ribeiro, Weifeng Chen, Gerome Miklau, and Donald F Towsley. Analyzing privacy in enterprise packet trace anonymization. In *NDSS*, 2008.

[138] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM, 2009.

[139] Rkt container. https://coreos.com/rkt/.

[140] Rami Rosen. Resource management: Linux kernel namespaces and cgroups. *Haifux, May*, 186, 2013.

[141] Indrajit Roy, Srinath TV Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for mapreduce. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*, pages 297–312, 2010.

[142] Ahmad-Reza Sadeghi and Christian Stüble. Property-based attestation for computing platforms: caring about properties, not mechanisms. In *Proceedings of the 2004 workshop on New security paradigms*, pages 67–77. ACM, 2004.

[143] Reiner Sailer, Trent Jaeger, Xiaolan Zhang, and Leendert Van Doorn. Attestation-based policy enforcement for remote access. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 308–317. ACM, 2004.

[144] Nuno Santos, Rodrigo Rodrigues, Krishna P Gummadi, and Stefan Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *USENIX Security Symposium*, pages 175–188, 2012.

[145] Joshua Schiffman, Yuqiong Sun, Hayawardh Vijayakumar, and Trent Jaeger. Cloud verifier: Verifiable auditing service for iaas clouds. In *IEEE Ninth World Congress on Services*, pages 239–246, 2013.

[146] Fred B Schneider, Kevin Walsh, and Emin Gün Sirer. Nexus authorization logic (nal): Design rationale and applications. *ACM Transactions on Information and System Security (TISSEC)*, 14(1):8, 2011.

[147] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *36th IEEE Symposium on Security and Privacy*, May 2015.

[148] Seccomp. `https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt`.

[149] Amazon Web Service. Automated Reasoning and Amazon s2n. `https://aws.amazon.com/blogs/security/automated-reasoning-and-amazon-s2n/`, 2016.

[150] Srinath Setty, Andrew J Blumberg, and Michael Walfish. Toward practical and unconditional verification of remote computations. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, 2011.

[151] Noam Shalev, Idit Keidar, Yaron Weinsberg, Yosef Moatti, and Elad Ben-Yehuda. Watchit: Who watches your it guy? In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 515–530, 2017.

[152] Rian Shambaugh, Aaron Weiss, and Arjun Guha. Rehearsal: A configuration verification tool for puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 416–430, 2016.

[153] Elaine Shi, Adrian Perrig, and Leendert Van Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *2005 IEEE Symposium on Security and Privacy*, pages 154–168, 2005.

[154] Daniel R. Simon, Aydan Yumerefendi, Ted Wobber, Martin Abadi, and Andrew Birrell. Authorizing applications in singularity. In *Proceedings of the 2007 Eurosys Conference*, March 2007.

[155] Rohit Sinha, Sriram Rajamani, Sanjit A. Seshia, and Kapil Vaswani. Moat: Verifying confidentiality of enclave programs. In *The ACM Conference on Computer and Communications Security (CCS)*, October 2015.

[156] Emin Gün Sirer, Willem de Bruijn, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B Schneider. Logical Attestation: an Authorization Architecture for Trustworthy Co mputing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 249–264, 2011.

[157] Emin Gün Sirer, Willem de Bruijn, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B Schneider. Logical attestation: an authorization architecture for trustworthy computing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 249–264, 2011.

[158] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing selinux as a linux security module. *NAI Labs Report*, 1(43):139, 2001.

[159] Sonatype DevSecOps Survey. `https://www.sonatype.com/devops-survey-report`.

[160] Cloud Service Usage Survey. `https://yanzhai.typeform.com/report/sZ0Yjm/Num74k0R4kruPwl3`.

[161] Paul Tarau. Styla - a prolog in scala,. `https://code.google.com/archive/p/styla/`, 2012.

[162] New york census block. `https://data.cityofnewyork.us/City-Government/2010-Census-Blocks/v2h8-6mxf`.

[163] New york taxi trip data. `http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml`.

[164] Basho Technologies. Riak is a Distributed, Decentralized Data Storage System. `https://github.com/basho/riak`.

[165] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.

[166] Andrew Tucker and David Comay. Solaris zones: Operating system support for server consolidation. In *Virtual Machine Research and Technology Symposium*, 2004.

[167] Valohai. `https://valohai.com`.

[168] Marten Van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 24–43, 2010.

[169] Venkatanathan Varadarajan, Thawan Kooburat, Benjamin Farley, Thomas Ristenpart, and Michael M Swift. Resource-freeing attacks: improve your cloud performance (at your neighbor's expense). In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 281–292. ACM, 2012.

[170] VeraCode Independent Audit Services. `https://www.veracode.com/services/independent-audit`.

[171] Visa. `https://usa.visa.com/`.

[172] Paul Voigt and Axel Von dem Bussche. *The EU General Data Protection Regulation (GDPR)*, volume 18. Springer, 2017.

[173] Victor Vu, Srinath Setty, Andrew J Blumberg, and Michael Walfish. A hybrid architecture for interactive verifiable computation. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 223–237, 2013.

[174] Riad S Wahby, Ye Ji, Andrew J Blumberg, Abhi Shelat, Justin Thaler, Michael Walfish, and Thomas Wies. Full accounting for verifiable outsourcing. In *CCS*, 2017.

[175] Liang Wang, Antonio Nappa, Juan Caballero, Thomas Ristenpart, and Aditya Akella. Whowas: A platform for measuring web deployments on iaas clouds. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 101–114. ACM, 2014.

[176] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. Faster secure two-party computation in the single-execution setting. Cryptology ePrint Archive, Report 2016/762, 2016. `https://eprint.iacr.org/2016/762`.

[177] Weaveworks. Weave net. `https://www.weave.works/oss/net/`.

[178] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the ACM SIGPLAN 2015 Conference on Programming Language Design and Implementation,*, pages 357–368, 2015.

[179] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early detection of configuration errors to reduce failure damage. In *OSDI*, pages 619–634, 2016.

[180] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. Crystalball: Predicting and preventing inconsistencies in deployed distributed systems. In *The 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI09)*, 2009.

[181] Latte kubernetes. `https://github.com/jerryz920/kubernetes`.

[182] Yuval Yarom and Katrina Falkner. Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.

[183] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *30th IEEE Symposium on Security and Privacy*, pages 79–93. IEEE, 2009.

[184] Simin You, Jianting Zhang, and Le Gruenwald. Large-scale spatial join query processing in cloud. In *2015 31st IEEE International Conference on Data Engineering Workshops (ICDEW)*, pages 34–41. IEEE, 2015.

[185] Pamela Zave. Using lightweight modeling to understand chord. *ACM SIGCOMM Computer Communication Review*, 42(2):49–57, 2012.

[186] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proc. 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 263–278, 2006.

[187] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazieres. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, volume 8, pages 293–308, 2008.

[188] Yan Zhai, Lichao Yin, Jeffrey Chase, Thomas Ristenpart, and Michael Swift. CQSTR: Securing Cross-Tenant Applications with Cloud Containers. In *the 7th ACM Symposium on Cloud Computing*, pages 223–236. ACM, 2016.

[189] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 305–316. ACM, 2012.