The Pennsylvania State University
The Graduate School

GABE: A CLOUD BROKERAGE SYSTEM FOR SERVICE SELECTION, ACCOUNTABILITY AND ENFORCEMENT

A Dissertation in
Information Sciences and Technology
by
Smitha Sundareswaran

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

May 2014

The Dissertation of Smitha Sundareswaran was reviewed and approved* by the following:

Anna C. Squicciarini
Assistant Professor, College of Information Sciences and Technology
Dissertation Advisor, Chair of Committee

Dinghao Wu
Assistant Professor, College of Information Sciences and Technology

Dongwon Lee
Associate Professor, College of Information Sciences and Technology

John J. Metzner
Professor, Computer Science and Engineering, Electrical Engineering

Peter Forster
Assistant Dean for Online Programs and Professional Education
Chair of the Graduate Program

*Signatures are on file in the Graduate School.

Abstract

Much like its meteorological counterpart, *Cloud Computing* is an amorphous agglomeration of entities. It is amorphous in that the exact layout of the servers, the load balancers and their functions are neither known nor fixed. Its an agglomerate in that multiple service providers and vendors often coordinate to form a multitenant system using virtualization. This complex environment offers great potential to providers and adopters, but also introduces great challenges in managing, combining and providing a variety of highly heterogeneous services. In particular, users interaction with these providers is often cumbersome, as the details of a cloud system are often abstracted away and unclear to most adopters. Further, cloud computing does not offer strong security guarantees, or traceability of data, and its indeterminate nature makes accountability of providers and users operations difficult[1]. This nebulous nature and the lack of security assurances of Cloud services together form the foremost barriers to its adoption.

The ambiguous nature of Cloud services also makes choosing the best service challenging. First of all, users may not be aware of all the service providers available to them. Secondly, the users may not be aware of the comprehensive list of options offered by the various service providers even if they know a particular provider. In the scenario where the users know which service option they are looking for, they may not be aware of all the providers supporting the option leading to uninformed choices. Thirdly, users may not be aware of the relationships among service providers, in case the users' requests would be satisfied by a set of providers, and not a single entity. These relationships between the service providers can cause resources, such as storage space, to be overextended. As a result, the process of collecting and analyzing the information required to make a good decision involves a lot of time-consuming computations for consumers. The time-consuming computations involve identifying all the service providers and the relationships between them before making a final choice. As these arduous com-

putations are repeated by multiple consumers who have similar requirements, it is also computationally wasteful. Protecting the data once the service provider is chosen is also major challenge not only because of the changeable relationships between the service providers, but also due to the potentially untrustworthy components of a single service provider. A major feature of the Cloud services is that users' data is usually processed remotely in unknown machines that the users do not own or operate. While enjoying the convenience of remote storage and processing brought by the Cloud services, users' fears of losing control of their own data, particularly financial and health data or any Personally Identifying Information (PII), can become a significant barrier to the wide-spread adoption of Cloud services [2].

In this dissertation, we aim to address some of the most significant barriers to the adoption of Cloud services. We propose a novel brokerage-based architecture called **GABE** - a Cloud brokeraGe system for service selection, AccountaBility and, policy Enforcement. GABE fulfills two major needs of cloud users: helping them understand the Cloud services best suited for them; and providing security assurances on their data. As the core part of the brokerage system, we design a unique indexing technique for managing the information of a large number of Cloud service Providers. Multiple alternatives to the indexing are studied to address specific needs in service selection. We then develop efficient service selection algorithms that rank potential service providers and aggregate them if necessary.

GABE also helps users protect their data by providing a policy driven node selection methodology for map reduce architectures. GABE seamlessly integrates node selection control to the MapReduce framework for increased data security. It leverages data preprocessing techniques and distributed node verification protocols to achieve strong policy enforcement.We further augment GABE by equipping it with accountability features. In order to support accountability, we propose a novel highly decentralized information accountability framework to keep track of the actual usage of the users' data in the Cloud. In particular, we propose an object-centered approach that enables enclosing our logging mechanism together with users' data and policies. We leverage object oriented programming techniques to create a dynamic and traveling object, and to ensure that any access to users' data will trigger authentication and automated logging local to the JARs. We take a policy-driven approach that strongly couples data and content protection policies (CPPs). This approach constitutes an effective and practical solution for content protection for a number of reasons. First of all, both the CPPs and the protection mechanism travel with the content, which is stored in its original form. Secondly, users do not need to rely on any dedicated management system to specify and apply the CPPs. Thirdly, to strengthen users' control, we also provide distributed auditing mechanisms.We provide extensive experimental studies on real

cloud computing testbeds that demonstrate the efficiency and effectiveness of the proposed policy driven node selection, auditing, and service selection approaches with real and synthetic Cloud data.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank many people who have helped me during my time in graduate school. First of all, I must thank Dr. Anna Squicciarini for taking me in as a new graduate student and being my adviser ever since. I would also like to especially thank Dr. John Metzner, who not only acted as my adviser when I started out as a student pursuing her Masters of Science in the Electrical Engineering Department, but also continued to be on my committee when I pursued my Ph.D.

I would also like to thank the rest of my doctoral committee, Dr. Dinghao Wu and Dr. Dongwon Lee, for being gracious with their time and for being especially understanding when I had to postpone my final defense due to medical reasons. At this point I should take a minute to thank the doctoral committee for being so understanding of my raspy voice during my final defense on account of my health problems.

I would also like to thank Dr. Dan Lin, from the Missouri University of Science and Technology, for guiding me on a lot of my research projects, including those that have gone towards becoming an integral part of my dissertation.

I would also like to thank Dr. Kevin Kane, at Microsoft Research, who has become a friend and a mentor to me ever since my internship with him. He has thought me a great deal about networking, demos, presentations and generally helped me with a difficult part of my life.

I would also like to thank all my friends, especially Denise Watts, Linda Tatali-iba, Lori Waters, and Lizandra S. Wyland who have been very encouraging of me through this process. Finally, I would like to thank my sister, Ranjani (Kat) Sundareswaran, for putting up with me when I have been buried in self-doubt and pain, despite her own problems, and keeping on me to power through. Kat, though I often say it teasingly, you are indeed my rock of Gibraltar!

Dedication

Dedicated to my sister, Kat (Ranjani), without whom I would not have ventured this far.

Chapter 1

Introduction

1.1 Background, Motivation and Solution

Cloud Computing is being heralded as the penultimate solution to the problems of uncertain traffic spikes, computing overloads, and potentially expensive investments in hardware for data processing, and backups. It can potentially transform the IT industry, making software and infrastructure both even more attractive as services, by reshaping the way hardware is designed and purchased. However, the understanding of the term remains as unclear as its origins itself. Hence we turn to the National Institute of Standards and Technology (NIST) for a definition. The NIST characterizes Cloud computing as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [1].Cloud computing, also referred to as "Cloud" in what follows, presents a new way to supplement the current consumption and delivery model for IT services based on the Internet, by providing for dynamically scalable and often virtualized resources over the Internet.

The concept of the Cloud includes a number of implementations, based on the services they provide, from application service provisioning to grid and utility computing. The most sophisticated implementation of the Cloud is referred to as Software as a Service (SaaS) [2, 3, 4], where the Cloud hosts an application for

the user[1]. Yet another implementation of Cloud is Platform as a Service (PaaS) [5, 6], where the Cloud provides a platform for the user allowing the user to exploit specifications such as the underlying infrastructure and the operating system. A third, even more basic implementation where the Cloud provides only the basic infrastructure on which the user can host everything including their own operating system is referred to as Infrastructure as a Service (IaaS) [7, 8]. With the proliferation of Cloud computing, MapReduce has now become a popular means to process Big Data in the cloud [9, 10, 11]. The MapReduce computing paradigm is an architectural and programming model for efficiently processing massive amounts of raw unstructured data. It provides a seamless distribution of computing tasks among nodes in the cloud, in a way which is transparent to the programmers. Its current design allows users' data to flow between nodes in the cloud, be they part of a public or a private cloud. Regardless of the specific architecture, the overarching concept of this computing model is that customers' data (which can come from individuals, organizations or enterprises) is processed remotely in unknown machines that users do not own or operate. The machines and the services may further be provided by subcontractors to the Cloud. Therefore, there are three major players in the Cloud: the users and organizations who own the data, the Cloud service providers (CSPs) who provide Cloud services to the users, and the subcontractors who provide storage or other services to the CSPs.

The NIST specifications detail five characteristics of Cloud services: 1) They should be on-demand services that the user can procure himself. 2) They should be available on a broad network of devices including remote laptops and mobile phones. 3) They should result in pooling of resources where the resources are assigned dynamically. 4) They should be elastic in that the assigned resources can be rapidly released, and 5) They should provide for some metering capability which allows for measuring the service provided [1].

These characteristics of Cloud services make them highly attractive to businesses of various sizes, ranging from small to large businesses. An attestation to their popularity can be found in the fact that as of date, there are a number of notable commercial and individual Cloud computing services, including Amazon, Google, Microsoft, Yahoo and Salesforce [12]. Also, top database subcontrac-

[1]In this dissertation we use client and user interchangeably

tors, like Oracle, are adding Cloud support to their databases. This new and exciting paradigm has also generated significant interest in the academic world [7, 8, 2, 3, 5, 13], resulting in extensive research. This research has had a great effect on the operations, architecture and security of notable commercial and individual Cloud computing services like those mentioned above [6, 13, 4].

Cloud computing services allow the users and organizations to take advantage of virtually unlimited computing power and storage power. All this storage and computing power is available to organizations and users without them having to make an investment in the requisite hardware. Therefore they are of great value to the users.

1.1.1 Motivation

The ubiquitous nature of Cloud services is confirmed by the statistics from market research which suggest an ever-growing usage of Cloud computing services by organizations and individual users alike. Merrill Lynch puts the Cloud computing industry to be at $160 billion by the year 2013 [14]. In a 2010 survey conducted by Mimecast, a supplier of Cloud-based email management solutions, 72% of the respondents agreed that Cloud services have improved the end-user experience. Gartner predicts that 60% of the server workload will be virtualized by 2014 [15]. While this kind of virtualization has multiple benefits including reduced costs in terms of hardware and the cost of cooling and running servers 24/7, the uptake of Cloud computing services will be immensely affected if the major concerns regarding it are not allayed. In fact a recent LinkedIn survey puts Cloud computing as the top security concern for 54% of its respondents [14].

Experts state that Cloud computing can actually improve the security of organizations [16] not only due to a focus on standard practices, but also due to the resilience and reliability of a Cloud in general. For instance, the distributed nature of the Cloud makes it more resilient against attacks like DDoS, which exploit a single point of failure for non-distributed system [17]. Not only can part of the infrastructure which is under attack be cordoned, but the back-ups on the other systems in the Cloud reduce the probability of data loss [16, 7, 8]. However, they also concede that Cloud computing calls for a different model of security. This is

due to several reasons. The processing and handling of the data on these servers is often provided by third party subcontractors. Details of the services provided are abstracted from the users who no longer need to be experts of technology infrastructure. This abstraction while being the key strength of Cloud services, also forms the main point of contention for the users. Most of the data stored on the Cloud platforms is quiet sensitive, considering that it could consist of Personally Identifiable Information (PII), which does not lend itself well to such abstraction. With sensitive PII, users would like to know how their data is handled, and by whom it is handled. This is especially the case when the organization owning the data is some health-related organization such as a hospital or an insurance company, or when it is a financial company [16, 18]. The threat of PII from financial organizations is especially relevant given that Deutsche Bank and financial services firm JPMorgan Chase are two of the major players in a group founded for regulating Cloud computing practices [19]. Further, when individual users are signing up for Cloud services, they are often required to divulge PII such as name, email address and credit card information. For instance, both Windows Azure and Amazon EC2 require credit card information even when the user is signing up for a trial [20, 21]. Besides, even if the data stored itself is not very sensitive, the usage of remotely stored data can reveal a lot of sensitive information about the owner, such as their daily schedule, their preferences, their location and thereby their travel patterns. Admittedly, the information revealed by the data usage patterns may be more privacy-sensitive to individuals than to large organizations with many people, as it might be hard to pinpoint the single user of the organization of whom the usage patterns are indicative. However, the massive amounts of data stored by the organizations can provide a goldmine of data about its customers, making the privacy concerns about its users critical.

The privacy, confidentiality and integrity of this sensitive data is especially called into question given that the users, the CSPs, and the subcontractors often have conflicting goals. While subcontractors and CSPs view client data as a tradeable asset that can be used for creating targeted advertising, marketing and even service oriented tools, clients are often concerned about data privacy. In fact there are significant statistics from both market research and academic research that point to a major lack of security as one of the chief concerns of clients re-

garding data and applications in the Cloud [16, 18, 7, 13]. Statistics from a study conducted by the Poneman Institute and sponsored by Dome9 [18] revealed that 67% of the respondents claim that their organization is left vulnerable to hackers due to lax port and firewall security for Cloud computing. 27% of respondents rated their organization's overall Cloud server security management as fair while 25% rate it as poor, bringing the total percentage of votes indicating a mediocre or worse Cloud server security management to 52%. Further, 54% of the respondents stated their IT staff had no knowledge about the potential risks of open firewall ports in their Cloud environments.

An extra dimension of privacy concerns is added to MapReduce when it is deployed on a Cloud. Initial constructions of MapReduce only ran in a single trusted data center. However Cloud based MapReduce may use some nodes in the public clouds that cannot be fully trusted [22, 23]. In a MapReduce Cloud, as data flows freely to public servers that are not within the data owners' control, MapReduce does not offer guarantees on the nodes managing (possibly sensitive) user data. On public or even hybrid clouds, a user submitting a job knows very little about the nodes doing the computation, such as whether they are private or public, their location, and their security configuration. Also, it is difficult to request for a specific data processing setup that would provide some guarantees on the quality of the nodes processing users' input. For example, in the classic word count example, a user may request that *words from sensitive input files except stop words (like "the", "is", "are") be processed by nodes capable of file level access control located in the U.S. domain.* This request cannot currently be satisfied without relying on a combination of cryptographic protocols and manual configuration of the workers to ignore certain words.

With the exception of private clouds, the user cannot regain control by offloading these controls to the MapReduce Master node. The Master node, which is the node coordinating the MapReduce computation, is only aware of the methods called by the workers. In case of Azure, the Master is aware of the methods called by the worker (e.g. the *context* method), and the properties of the method (e.g. *JobParameters*, *MapperType*), but not of their implementation (it does not know how the *MapperType* is implemented) [10]. In case of Amazon's Elastic MapReduce (EMR), the Master may or may not have the insight into the mapper and reducer

classes and methods, depending on how the particular application is created [9]. That is, it is possible to create applications where the workers' classes are either totally visible or totally abstract to the Master. For instance, in the Cisco Nexus 1000V InterCloud [24], not only are the virtual machines' environment heterogeneous, but the actual physical hosts are also geographically distributed, and offer different degrees of trust and security. In this case, the master node does not have a complete control over any of the more secure nodes, unless the master node is a part of the trusted network, in which case, it loses control over the non-secure nodes. Another cloud computing platform where a single instance of a database can be federated over globally distributed cloud computing environments, with no robust trust guarantees, is the TransLattice Elastic Database 3.0 [25]. The trans lattice platform is the the world's first geographically distributed database, making it very similar to a Cloud Service, if not a Cloud service itself. While it provides data governance, and granular data location control, it does not provide any guarantees on the node's security or processing capabilites. The Google MapReduce design is similar, in that the classes of the workers are abstracted from the Master. The Master is mainly concerned with sharding, and performance tracking [11].

Also, though methods such as homomorphic encryption [26, 27] or outsourced private computation [28] can protect the data by processing them in the encrypted domain, such approaches are typically computationally expensive and hence are only feasible for selected applications [29]. An alternative approach is to ensure that sensitive data are never stored and processed in the public cloud. Under this approach, the input data are divided into pieces that are classified as either sensitive or non-sensitive, and mechanisms are then in place to prevent leakage of sensitive information to the public cloud during execution. The challenge then is how to ensure that public cloud resources can be used efficiently in a cost effective manner, and how to control that the assignment on public nodes is compliant with the users' requests.

Therefore, to make the data usable, and to take advantage of the computing power, and portability offered by Cloud computing services it is essential to have some sort of broker who will enable the user to interact with the CSPs. This service can be provided by a Cloud brokerage system. The importance of such a service is stressed by Gartner [30, 31], that defined different types of Cloud

brokerage, including arbitrage, aggregation and intermediation. Similarly, other recent work has acknowledged the increasingly important role of Cloud brokers, [32, 33] and their multiple responsibilities, which range from service composition to monitoring. Even Dell has recently claimed an interest in Cloud services brokering, and has been working in partnership with VMWare to push out the same. One of the best established brokers is CloudSwitch. Established in 2008 with service for only Amazon EC2, it has the ability to provide federated services on demand and make the cloud a secure and seamless extension of the enterprise data center by working as the middleman [34]. RightScale is another cloud broker that offers a cloud management platform that enables organizations to deploy and manage applications across multiple clouds [35]. However, these brokers go no further than allowing users to manage applications. They do not provide security assurances or auditing, nor do they deal with the complexity of mode selection in MapReduce Clouds.

1.1.2 Proposed Solution

GABE aims to bridge these gaps. GABE not only aims to empower users with service selection but also provide them with tools for node selection, security and accountability. GABE will empower users to select the CSPs appropriate to their specific needs by providing them with a ranked list of CSPs by taking into consideration not only their queries but also hidden relationships between the CSPs themselves in form of shared subcontractors. GABE will address the lack of controls in MapReduce node selection by means of a policy-based node selection framework. This framework is designed for node selection in any distributed system, but GABE employs it specifically in the context of MapReduce. Accordingly, the broker's MapReduce module is referred to as PARiNgS: **P**olicy driven m**A**p**R**educe **N**ode **S**election. It is a MapReduce extension to allow policy enforcement on the nodes. The core idea underlying PARiNgS is to enable the enforcement of security requirements specified by users on the processing of their data by MapReduce functions. These security requirements may express conditions, for example, on the nodes' functional capabilities, their locations and their cryptographic capabilities. Requirements are defined in terms of simple policy rules against the nodes

that process the users' sensitive data. To support an efficient and effective policy driven node selection mechanism, we focus on attribute-based access control policies, wherein the attributes specify the properties against which some conditions are specified.

The enforcement of the security requirements is elegantly interleaved with the scheduling process of the tasks, performed by the Master nodes, and with other intermediate steps of the task execution. Enforcement primarily deals with verifying the properties of workers, before allowing them to access the data they will be processing. Note that, extracting, and, most importantly, verifying properties of workers (e.g. location, supported cryptographic, and file level access control) is in fact non-trivial in cloud computing infrastructure, given the possibly high-level of virtualization and the lack of a centralized authority that can verify such properties in an efficient fashion. The Master, for example, although in charge of partitioning and completing other tasks, typically has only selected information about the nodes, and much of such information is not verified. To this end, we provide a collaborative verification protocol, that allows remote verification of workers' properties given a policy in a collaborative, yet secure fashion with limited overhead. To minimize the risk of collusion, the workers' verification is dynamic, and involves different set of workers at each time, selected according a probabilistic algorithm. Our framework also includes a tainting module, that taints any sensitive input data before it is processed by the mappers, and at other intermediate steps of the computation, as needed. The tainting module addresses the problem of data tracking, that arises when the application transforms the input at intermediate computation steps, so that the intermediate results are not of the same type as the original input, thus challenging the the policy applicability during these steps.

To improve the security of the users' data, GABE will also function as a distributed policy enforcement mechanism. Specifically, our approach binds the users' policies with the data, so that both the policies and the data travel together in a distributed environment. We achieve this strong coupling of policies and data by leveraging object oriented programming techniques. Using encapsulation techniques also allows us to bind a distributed policy enforcement mechanism to the traveling data, which works in tandem with the accountability framework to provide a strong security mechanism for the user. GABE will provide accountability

information to the users, showing the usage of the users' data by the CSPs, and by other subcontractors. A key feature of GABE is that while it is designed and deployed for Cloud computing, it can be deployed on any distributed computing environment, specifically email systems, with minor modifications.

1.2 Contributions of this Dissertation

The first contribution of this dissertation is the design and development of a Cloud brokerage system. Analogous to a stock broker, a Cloud broker is essentially an intermediary between the user and the service providers, who helps the user with the task of choosing services tailored to his need. Not only is the Cloud brokerage system is responsible for matching up a user with a suitable service provider, it is also responsible for enabling the transactions between parties that potentially do not know each other. To achieve these two goals, the Cloud brokerage in its very basic form attempts to obtain the privacy and usage policies of all the players involved in a data transaction, and carries out policy matching to enable the data transaction, while providing the parties involved with some proof of each other's reliability. However, a simplistic policy matching does not enable the best possible use of the users' data - both the data uploaded by users, and the usage data and any other PII obtained by the CSP when the user signs up for the service. For instance, some user requests can only be fulfilled by multiple CSPs. Providing a brokerage system that identifies multiple CSPs to satisfy a single request poses several security challenges. The process of selection itself can reveal the privacy preferences of the users to the other two parties, as the heightened privacy for a particular data item can be considered indicative of its sensitive nature. Further, when multiple CSPs are involved, there is a chance that they connive to derive sensitive information about the user, including the user's usage patterns and privacy preferences. Hence, the process of CSP selection is a challenging job. In this dissertation, we explore the major issues in CSP selection as the first challenge faced by a Cloud brokerage system. Accordingly in this dissertation we design a service selection algorithm that empowers clients to select the CSP or CSPs most suitable to their needs. The algorithm selects and ranks CSPs based on the client's needs which are stated in a query, and aggregates the CSPs if necessary.

Once the CSPs are selected, it is quite possible that the users have had no previous transaction history with the CSPs in question. Therefore the reliability of the CSPs is uncertain from the users' perspective. This is turn calls the reliability of subcontractors into question as they are the ones often providing crucial services to the CSPs. Further, users have a lack of trust in allowing the service providers to share the data with their subcontractors because once the data is given out by the service providers to the subcontractors, the users no longer have any control on this data. For example, users need to be able to ensure that their data is handled according to the service level agreements made at the time they sign on for services in the Cloud. Furthermore, the users need to be able to enforce any access policies that have been agreed upon. Hence the data brokerage service also seeks to provide aspects of policy enforcement and accountability. The importance of some form of accountability is underscored by fact that the 42% of the respondents of the Poneman study fear they wouldn't know if their data or applications on their Cloud were actually compromised or if a data breach occurred involving an open port on a Cloud server [18].

Precisely, the second contribution of this dissertation is an accountability framework. Accountability focuses on keeping the data usage transparent and trackable. In this dissertation, we explore the aspects of reliability and accountability as the second challenge faced by a Cloud brokerage system. The Cloud brokerage service proposes to provide end-to-end accountability by combining the aspects of authentication and access control. Conventional access control approaches developed for closed domains such as databases and operating systems, or approaches using a centralized server in distributed environments are not suitable due to the following features characterizing Cloud environments. First, data handling can be outsourced directly by the CSP to other entities in the Cloud and these entities can also delegate the tasks to others, and so on. Second, entities are allowed to join and leave the Cloud in a flexible manner. As a result, data handling in the Cloud goes through a complex and dynamic hierarchical service chain which does not exist in conventional environments. This need is best met by a dynamic, distributed accountability system which is based on the notion of information accountability [36]. Unlike privacy protection technologies which are built on the hide-it-or-lose-it perspective, information accountability focuses on keeping the data usage trans-

parent and trackable. Accordingly, GABE incorporates aspects of information accountability to provide the users a sense of reliability of the Cloud.

The distributed access control and information accountability framework proposed in this dissertation is suitable for being deployed on the Clouds. Our proposed Cloud Information Availability (CIA) framework provides end-to-end accountability in a highly distributed fashion. One of the main innovative features of this framework lies in its ability of maintaining lightweight and powerful accountability that combines aspects of access control, usage control and authentication. Through it, data owners can track not only whether or not the service level agreements are being honored, but also enforce access and usage control rules as needed.

The final contribution of this dissertation is the development of the PARiNgS framework as a part of GABE for policy driven node selection in Cloud service. PARiNgS is studied specifically in the context of MapReduce in this dissertation, but it can be applied as is for node selection in any distributed computing environment. It seamlessly integrates node selection with the scheduling process of the tasks, performed by the Master nodes, and with other intermediate steps of the task execution. The policy enforcement is achieved using a combination of data tainting, pre-processing, and requirement checking. With regards to MapReduce the key contributions of this dissertation are as follows:

- We propose the very first work to seamlessly integrate policy-driven node selection in a MapReduce framework.
- We develop a synoptic yet eloquent policy language for users to express their policies.
- We propose data pre-processing methods based on a tainting strategy. Our tainting strategy ensures that the policy is applied to the sensitive data as it gets processed at various nodes.
- We deploy PARiNgS on top of an actual MapReduce implementation and demonstrate the scalability of our framework in terms of the additional time taken for processing.

1.3 Layout of this Dissertation

The rest of the Dissertation is organized as follows. Section 2 reviews the background and related work. Section 3 presents a brief overview of the workflow of the entire framework. Section 4 discusses in detail the process of Cloud Service Selection. Section 5 discusses the details of the PARiNgS framework for MapReduce. Section 6 discusses in detail the information accountability framework. Section 7 presents the experiments conducted for the Cloud brokerage system. Section 8 contains the concluding remarks and possible directions for future work. Appendix A presents a application scenario for the CIA architecture beyond the Clouds.

Chapter 2

Background and Related work

Cloud computing raises a range of important *privacy* issues as acknowledged by a number of recent works [37, 38, 39, 40, 41, 42]. Concerns arise since in the Cloud it is not clear to users why their personal information is requested or how it will be used or passed on to other parties.

Despite increased awareness of the privacy issues in the cloud, little work has been done in this space. Most of the work done to ensure privacy comprises of general outsourcing techniques that have been investigated over the past few years [43, 44, 45]. Even in the proposed outsourcing techniques, only [46] is specific for the Cloud. However, some of the outsourcing protocols may also be applied in this realm. Several cryptographic-based approaches for ensuring remote data integrity have also been proposed.

In this section, we review related works addressing the privacy and security issues in the Cloud. Specifically, we discuss works related to each aspect of our research work, and conclude the section with a discussion of works focused on distributed policy enforcement beyond the Cloud computing domain.

2.1 Work related to Cloud Brokerage and Service Selection

Some high level discussion on service provider selection and brokerage-based frameworks in the Cloud can be found in [47, 48, 33, 49]. In particular, Gartner [31]

defined different types of Cloud brokerage, including arbitrage, aggregation and intermediation. Similarly, other recent work has acknowledged the increasingly important role of Cloud brokers, [48, 49] and their multiple responsibilities, which range from service composition to monitoring. As acknowledged by this body of work, the first step for a broker to fulfill these responsibilities is to rank and select the appropriate Cloud service providers, based on the user's requirements. However, to the best of our knowledge, there is not any existing work that provides a specific solution. The only academic effort in this direction is from Buya et al. [50] who provide a general description of the key role of Cloud broker services for a market-oriented Cloud service. In addition, although not specific to Cloud brokers, Xin et al. [51] have considered collaborative protocols among Cloud service providers for resource sharing. Xin et al. use trust as the criteria to select service partners.

While Cloud brokerage and selection is a relatively unexplored topic, ranking and selection algorithms have been studied in great depth in the context of Web-Services. To date, most of the works on Web-Service selection are based on QoS (Quality of Service) [52]. For instance, Kalepu et al [53] propose an objective measure of QoS based on the extent up to which the Web-Service meets its service level agreements. Similarly, Zeng et al. [54] present a middleware aimed at selecting Web-Services for composition such that user satisfaction is maximized when it is expressed as a utility over QoS functions, while satisfying user and service provider constraints at the same time. The authors first identify five generic quality criteria for elementary services, ranging from price to availability.

Web-Services composition from the standpoint of matching the semantics between the services requested have also been explored [55]. Paolucci et al. provided a solution based on DAML-S, a DAML-based language for service description. The authors perform a semantic matching between the request and an advertisement. Their algorithm sorts the matches found, but it does not define in detail the criteria for the ranking. Web-Service composition approaches relevant to our effort are those focused on business collaboration [56]. Bentallah et al. consider consider fast composition, scalable composition and distributed composition as the main issues surrounding Web-Services composition. They assume the existence of a ranking algorithm and concentrate on how to compose the Web-Services given

the participants.

Additionally, as we explain in Section 4.2.2.3, our work employs a recent high-dimensional data indexing technique, i.e., the iDistance [57]. High-dimensional data indexes [58, 59, 60, 61, 62, 63] are typically used for retrieval of similar multimedia objects such as images, and videos. It is not trivial to adapt it to the Cloud service provider selection. We propose a new encoding technique that represents multiple properties of service providers. We also design novel querying techniques that take into account unique characteristics in the Cloud such as the relationships among service providers, which do not exist in conventional high-dimensional data sets.

2.2 Accountability in Distributed Systems

One of the works most similar to ours in the goal of tight coupling of data and policies is that on sticky policies by Pearson et al. The authors have proposed accountability mechanisms for identity management to address privacy concerns of end users [42] and then develop a privacy manager [64]. Their basic idea is that the user's private data is sent to the Cloud in an encrypted form, and the processing is done on the encrypted data. The output of the processing is de-obfuscated by the privacy manager to reveal the correct result.

In [65], Chun et al. present a layered architecture for addressing the end-to-end trust management and accountability problem in federated systems. The authors mainly leverage trust relationships for accountability, along with authentication and anomaly detection. Their solution requires third party services to complete the monitoring and focuses on lower level monitoring of system resources.

In general, researchers have investigated accountability mostly as a provable property through cryptographic mechanisms, particularly in the context of electronic commerce [66, 67]. A representative work in this area is given by Corin et al. [68]. The authors propose the usage of policies attached to the data and present a logic for accountability data in distributed settings. Similarly, Jagadesaan et al. recently proposed a logic for designing accountability-based distributed systems [69]. In [67], Crispo and colleagues proposed an interesting approach related to accountability in case of delegation. In summary, all these works stay at a theoretical

level and do not include any algorithm for tasks like mandatory logging.

To the best of our knowledge, the only work proposing a distributed approach to accountability is from Lee and colleagues [70]. The authors have proposed an agent-based system specific to grid computing. Distributed jobs, along with the resource consumption at local machines are tracked by static software agents. The notion of accountability policies in [70] is related to our algorithm which specifies what accountability information is to be collected and whether a content protection policy is to be enforced, but it is mainly focused on resource consumption and on tracking of sub-jobs processed at multiple computing nodes, rather than access control.

The accountability provided by GABE and the enforcement of content protection policies depends on the usage of object oriented programming techniques. With respect to using object oriented programming techniques for security, our methods are related to self-defending objects [71]. Self defending objects (SDO) are an extension of the object oriented programming paradigm, where software objects that offer sensitive functions or hold sensitive data are responsible for protecting those functions/data. The authors rely on a centralized database to maintain the access records, while the items being protected are held as separate files.

In previous work, we provided a Java-based approach to prevent privacy leakage from indexing [72], which could be integrated with the CIA framework proposed in this work since they build on related architectures.

In terms of authentication techniques used during enforcement and auditing, Appel and Felten [73] proposed the Proof-Carrying authentication framework (PCA). The PCA includes a high order logic language that allows quantification over predicates, and focuses on access control for web-services. Using PCA helps maintaining safe, high-performance, mobile code, it focuses on validating code, rather than monitoring content.

In addition, our work may look similar to works on secure data provenance [74, 75, 76], but in fact greatly differs from them in terms of goals, techniques and application domains. Works on data provenance aim to guarantee data integrity by securing the data provenance. They ensure that no one can add or remove entries in the middle of a provenance chain without detection, so that data is correctly

delivered to the receiver.

Finally, general outsourcing techniques have been investigated over the past few years [45, 44]. Although only [46] is specific to the Cloud, some of the outsourcing protocols may also be applied in this realm. In this work, we do not cover issues of data storage security which are a complementary aspect of the privacy issues.

2.3 Background of MapReduce

MapReduce is a functional programming paradigm. It enables parallel programming of large data efficiently using multiple nodes. Its programming model is built upon a distributed file system (DFS) which provides distributed storage. Programmers specify two functions: *Map* and *Reduce*. The Map function receives a key/value pair as input and generates intermediate key/value pairs to be further processed. The Reduce function merges all the intermediate key/value pairs associated with the same (intermediate) key and then generates final output. On a cloud computing setting, these functions are orchestrated by the Master, and carried out by the mappers, and reducers. The Master acts as the coordinator responsible for task scheduling, job management, etc.

A Master's module (typically the data partitioner) splits input data into a set of M blocks, which will be read by M mappers through DFS I/O. The execution of map and reduce tasks are automatically distributed across all the nodes in the cluster. The *map* function takes as input one of the M blocks, which is defined as a key-value pair, and produces a set of intermediate key-value pairs. The intermediate result is sorted by the keys so that all pairs with the same key will be grouped together (the shuffle phase). If the memory size is limited, an external sort might be used to deal with large amounts of data at one time. The locations of the intermediate results are sent to the Master who notifies the reducers to prepare to receive the intermediate results as their input. Reducers then use Remote Procedure Call (RPC) to read data from mappers. The user defined reduce function is then applied to the sorted data; basically, key pairs with the same key will be reduced in some way, depending on the user defined reduce function. Each mapper will process the data by parsing the key/value pair and then generate the intermediate result that is stored in its local file system. Finally, the output will

be written to DFS.

2.3.1 Security and Privacy in MapReduce

There is growing interest in security of MapReduce [77, 78, 79, 80, 81, 26, 27, 28]. The Sedic framework [78], is the closest effort to ours. Sedic aims to partition the data according to the inputs sensitivity level. If a data piece is sensitive, it is sent to a sensitive mapper. For reducer computations, Sedic modifies the reducer routines by checking whether they contain certain loop dependent variables: if they do then the partition of sensitive and non-sensitive data is affected else data from sensitive mappers can be pushed to non sensitive reducers. Sedic achieves its goals by modifying how the data is read: normally, the entire data is read using a single pointer, however with Sedic only a block of data is read using a given pointer. As we discuss in Section 5.3.1, PARiNgs also includes algorithms for data partitioning, in addition to checking that the workers satisfy user-specified conditions before they are allowed to process the data.

Also closely related is the Airavat [79] project. Airavat is a security and privacy framework for MapReduce systems. Airavat aims to enforce differential privacy, i.e. it aims to ensure that the output of aggregate computations does not violate the privacy of individual inputs. It does so by modifying the Java Virtual Machine and the MapReduce framework by adding SELinux-like Mandatory Access Control to the DFS. Not only does the methodology of Airavat differ from that of PARiNgS, but the end goals of the frameworks are also different: While Airavat tries to prevent the processing of the data by untrusted code, PARiNgS tries to prevent the processing of the data by untrusted nodes.

Another related work, which, similar to our work, relies on distributed verification (see Section 5.4.1) is the SecureMR framework [81]. The framework is intended to be a practical service integrity assurance framework for MapReduce. It allows mappers to examine the integrity of data blocks from the DFS; verify the authenticity and correctness of the mappers' results; and allows users to check the authenticity and correctness of the reducers' final results.

Finally, we are loosely related to the body of work focusing on cloud computing integrity of computation [82, 80, 83]. For example Moca's [80] proposal deals with

distributed results checking for MapReduce. The work relies on the distributed voting method to check the correctness of the results produced by MapReduce. This work is complementary to ours in that while it relies on a distributed approach, it verifies the correctness of the computation, rather than whether a user's requirements of the nodes are satisfied.

Chapter 3

Overview

In this dissertation, we present GABE, a brokerage system designed to provide users a practical, easy to use framework to manage their interactions with the CSPs from selection, and safeguard their data on the Cloud. The brokerage system manages the interactions of the users with the CSPs from the first step of selecting the appropriate service providers to the last step of using the experience with a given CSP as a feedback for any future dealings. In Figure 3.1 , we sketch the flow of operations in GABE.

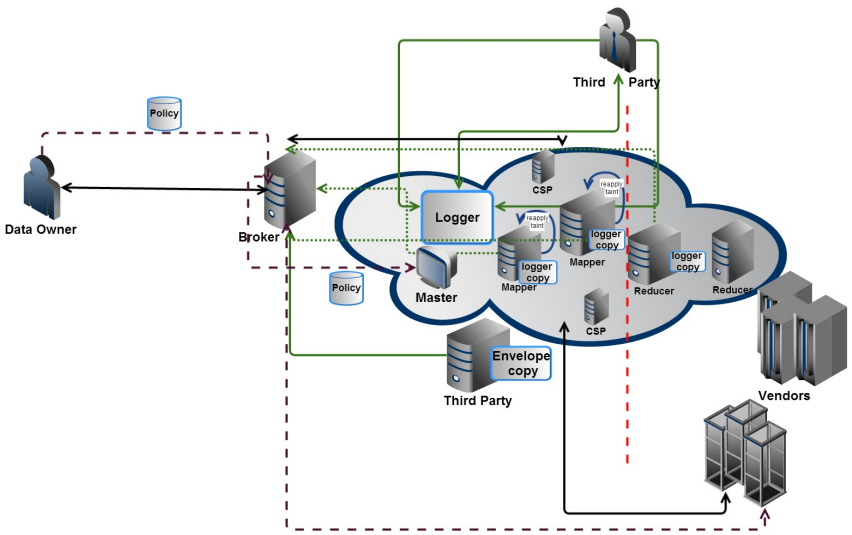

Figure 3.1. Overview of the flow of operations in GABE

As seen in Figure 3.1, the current architecture of GABE consists of three main

parts. The first part of the framework is in charge of helping the users select the CSPs, and aggregating CSPs to satisfy the users' requests if needed. We refer to this part as the Cloud Service Selection (CSS) Framework. The second part of the framework is responsible for performing the auditing mechanisms and enforcing the policies which have been set by the users. We refer to the part of the brokerage system that is responsible for accountability as the Cloud Information Accountability (CIA). The third part deals with policy-driven node selection and is used when data storage, and more specifically data computation present certain functional requirements. Examples of such requirements are capability to execute a specific algorithm, and non-functional requirements such as location of the provider's nodes, and their security and privacy guarantees. This part of the framework is referred to as PARiNgS: **P**olicy driven m**A**p **R**educe **N**ode **S**election. Though PARiNgS is specifically designed for MapReduce as suggested by its cronym, it can be used to select nodes in any Cloud or distributed environment.

GABE's first steps consist of collecting the privacy and usage policies of all possible CSPs. It then tries to carry out a policy matching to enable the data transaction by checking whether any of the terms of use of either the subcontractors or the CSPs are conflicting with the requests of the user. However, this simplistic policy matching does not enable the best use of the data possible. For instance, multiple service providers could be more optimal for a user as opposed to a single service provider, as they may prove to be more cost effective. GABE therefore factors in the user's preferences specified in his request, and accordingly ranks the CSPs to reflect the same. Therefore, in case multiple CSPs are more cost effective, and cost is one the user's primary preferences, the multiple CSPs are ranked above a single CSP.

Once the CSPs are agreed upon, the users begin the process of sharing their data with the service providers and sending a request for data processing or storage. The service chosen may simply be a storage service or may require computing, both of which are done on nodes which meet policy specifications. GABE enables the users to create an envelope for their data which provides accountability, with optional policy enforcement. The accountability system encapsulates users' data in a distributed policy enforcement engine. The enforcement engine, which forms one of the main components of the CIA is strongly bound with the data through

executable policies, and travels with it. Strong binding is ensured even when the data is copied and/or distributed to remote third party users. The engine also performs logging of all the accesses made to the data. While logging, it sends redundancy information to help with the recovery of logs back to the broker. Since the enforcement and logging is performed by a distributed engine, the log files need to be aggregated when they are sent to the user. The process of amalgamating the log files is carried out by the component of the CIA that is resident on the broker. Hence, the log files are actually collected at the broker, who sanitizes and formats them before sending them back to the user. If the service chosen requires computing on nodes which meet policy specifications, the broker takes the additional steps of inserting the policy driven node selection logic into the Master nodes of the architecture. This insertion is done using a software shim, which equips the Master with the policy administration point (PAP) and a policy information point (PIP) which are components of the PARiNgS framework. The PAP and PIP are installed as a part of the users' request.

Unlike the brokerage system defined in [84], the system in this dissertation does not focus on service composition for the end user. That is, while GABE helps the user select multiple CSPs either to satisfy his requirements, it does not merge the services provided by the CSPs to present them as a single unit to the user. Solving service selection is an important first step in making Cloud services more approachable, because despite their advantages, the sheer number of choices can be confusing, especially for key decision makers who are not technically inclined. While service composition arguably has its benefits such as providing a seamless experience to the user [52], it is often best for the user to know all the entities involved in handling of his data. Further, the user has the ability to portion his data according to his requirements and previous experiences, when the services are not presented in a unified manner.

Chapter 4

Cloud Service Selection

4.1 Introduction

Cloud services offer an elastic and scalable IT task force in terms of storage space and computing capabilities which is essential to most business owners, especially small and medium sized businesses [24]. While this has fueled the large growth in Cloud services, the growing number of Cloud services make it difficult for the potential users to weigh and decide which options suit their requirements the best. There is clearly a need of an additional computing layer on top the base service provisioning to enable tasks such as discovery, mediation, and monitoring. This is the first task accomplished by GABE.

The selection of Cloud providers requires addressing a number of interesting questions raised by the unique characteristics of the Cloud computing environments. First, Cloud services may seem to resemble but are in fact very different from Web-Services. For example, There is no standardized representation of the Cloud providers' properties. Also, the Service Level Agreements (SLAs) of Cloud providers often vary in format and content. Therefore, Web-Service selection algorithms [53, 85, 86] cannot be directly applied to the Cloud domain. Secondly, a Cloud user may have a service requirement that cannot be fulfilled by any single service provider, thus requiring an aggregation of service providers. Aggregating service providers is very challenging in the Cloud due to complex relationship among Cloud service providers that are built via subcontracting. For example, when aggregating service providers that rely on the same contractor for storage

space, we should be careful so as to avoid overextending the actual storage space.

Bearing these challenges in mind, in this dissertation we propose a comprehensive brokerage-based architecture to support Cloud service selection. In particular, we propose an efficient indexing structure, the CSP (Cloud Service Provider) index, to manage the potentially large number of service providers. The CSP-index is built based on a novel encoding technique that captures similarity among various properties of service providers. With the aid of the CSP-index, we further design the service selection algorithm that considers aggregation of services and provides rankings of potential service providers. To evaluate our approach, we have collected real data from top 10 Cloud providers listed by SearchCloudComputing in [47]. Our experimental study, presented in Chapter 7, demonstrates both efficiency and effectiveness of our approach.

4.2 Overview of the CSS Framework

The overall architecture of the CSS Framework for a brokerage-based Cloud service system is illustrated in Figure 4.1. There are three main entities in the system excluding the subcontractors: Cloud service provider, Cloud broker, and end users. For all practical purposes, the subcontractors can be treated as a part of the CSPs, since they are directly involved only with the CSPs. The Cloud broker, which has a contract with the Cloud service providers, collects their properties (e.g, service type, unit cost, and available resources), and the consumer's service requirements. The Cloud broker analyzes and indexes the service providers according to the similarity of their properties. Upon receiving the service selection request from an end user, the Cloud broker will search the index to identify a ranked list of candidate providers based on how well they match the user requirements. This list forms the basis of the end users' final decision. The realization of the architecture includes two key technical issues. One is the construction of the index for managing the service providers. The other is the query algorithm for the service selection.

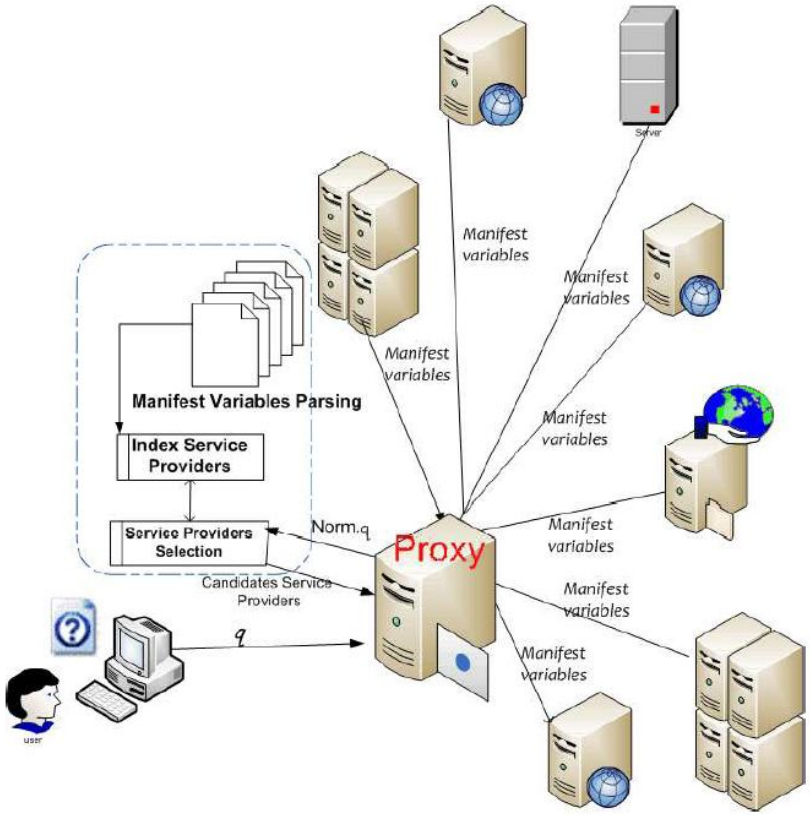

Figure 4.1. Service Selection Framework

4.2.1 Types of User Requests

A user sends a service selection query to the broker which specifies what properties and values he/she expects from the service providers. Our brokerage system supports two types of queries as defined in the following.

Definition 1. An **exact query** on cloud services is in the form of $Q = \langle (QP_1 : V_1), (QP_2 : V_2), \ldots, (QP_k : V_k) \rangle$, where $k \leq 10$, QP_i $(1 \leq i \leq k)$ is the property that the user requests the service provider to possess, and V_i describes the user expected value of property QP_i.

Definition 2. An **interval query** on cloud services is in the form of $Q = \langle (QP_1 : I_1), (QP_2 : I_2), \ldots, (QP_k : I_k) \rangle$, where $k \leq 10$, QP_i $(1 \leq i \leq k)$ is the property that the user requests the service provider to possess, and I_i describes the range of user expected values of property QP_i.

4.2.2 Indexing Cloud Service Providers

In face of a large number of Cloud service providers, it is important to design an efficient index structure to facilitate information management and retrieval.

In face of a large number of cloud service providers, it is important to design an efficient data structure to facilitate information management and retrieval. The specific design goals are the following.

- If the broker needs to update a CSP's information such as change of properties and available resources, the broker should be able to quickly locate where the CSP is stored rather than scanning the whole database.

- Given a user's service selection request, the broker should be able to narrow down the search within a small group of eligible CSPs rather than checking all CSPs against the user's request.

To achieve the above goals, we propose a B^+-tree which provides quick access to individual CSPs and also groups CSPs with similar properties to facilitate queries. The CSP-index is developed using the B^+-tree as the base structure since the B^+-tree is widely adopted in commercial database systems and provides the great foundation for our new index structure to be easily integrated to existing systems. We elaborate on our initial design of the B^+-tree, by designing what we refer to as a B^{cloud}-tree.

The B^{cloud}-tree is a multi-level balance tree as shown in Figure 6.1. Each node in the B^{cloud}-tree is of the same size (typically the size of a disk page), and contains a number of equal-size entries. An entry in the leaf node stores the following information of a CSP: $\langle \kappa, ID, p_1, p_2, ..., p_{10} \rangle$, where κ is the indexing key of the CSP whose identity is ID, and p_i is the i^{th} property of the CSP ($0 \leq i \leq 10$). The internal nodes of the B^{cloud}-tree serve as the search directory which contain indexing keys and pointers to children nodes.

The B^{cloud}-tree has a similar structure as the B^+-tree. The biggest advantage of basing the B^{cloud}-tree on the B^+-tree is that the B^+-tree provides the great foundation for our new index structure to be easily integrated to existing systems since the B^+-tree is widely adopted in commercial database systems. Moreover, we can also leverage the same suite of efficient B^+-tree's algorithms to search, insert,

delete and update the information of a CSP (i.e., an entry in a leaf node). For example, to search the properties of a CSP with a known indexing key, we start from the root of the B^{cloud}-tree, and look for the internal pointer that leads to the range of keys including the CSP's indexing key. The search just needs to access h nodes to locate any given CSP where h is the height of the tree.

In the design of the B^{cloud}-tree, the main challenge is the generation of the indexing key. Specifically, the indexing keys determine the storage order of CSPs. How the CSPs are stored has a great impact on the efficiency of answering a user's service request. The B^{cloud}-tree just needs to access one node while the traditional B^+-tree needs to check all the leaf nodes which ends up a linear search. Therefore, the desired indexing keys in the B^{cloud}-tree should be generated in the way that CSPs with similar properties receive nearby indexing keys.

In what follows, we first describe the alternative data structures of the CSP-index and then present the index construction algorithms.

4.2.2.1 Data Structure

The internal nodes of the CSP-index have the similar format for both the B^+-tree and the B^{cloud}-tree, and serve as the search directory. Each entry in the leaf node of the CSP-index has the following format: $\langle Key_{sp}, SID, p_1, p_2, ..., p_{10}\rangle$, where Key_{sp} is the indexing key, SID is the identity of the service provider, p_i is the i^{th} property of the service provider. The generation of Key_{sp} will be elaborated in the next subsection. This work focuses on the ten properties most commonly acknowledged as relevant properties of CSPs. We discuss the approach used to identify these properties in Section 7.3.

- `Service Type` (p_1). This denotes the type of service provided, which could vary between service on-demand, and reserved instances, or refer to specialized services such as custom IPs in case of Amazon or caching in case of Windows Azure.
- `Security` (p_2). This denotes the level of security and/or privacy that can be achieved using the various options provided by the Cloud provider. If the service provider satisfies three or more of the Information Security guidelines listed in the Security Guidance for Critical Areas of Focus in Cloud

Computing published by the Cloud Standards group [19], apart from the compliance or risk management guidelines, the service providers are classified as having high security. If they satisfy only the compliance, legal or risk management guidelines they are classified as having medium security. Else they are classified as having low security. Accordingly, when the service provider offers advanced security services such as Access Control, offered by Windows Azure, or it adopts a number of security standards (e.g., secure connections), the level of security is labeled as *high*. When there are security options, but these options are limited to secure passwords and encryption as in case of Rackspace, the level of security that can be achieved is considered to be *low*. When the features do not include detailed access control options, but still provide improved security through automatic security updates, as Google does for its Clouds, then the security level is considered to be *medium*.

- `Quality of service` (QoS, p_3). QoS is determined by the Cloud broker which analyze the collected information about service providers over time, and ratings provided by other vendors. It is briefly represented using value high, low or medium.
- `Measurement units` (p_4). This represents in what terms the service can be charged. Measurement can be in terms of memory used, the number of the transactions, the number of connections or data transfers, or the time taken for the data transfer.
- `Pricing Units` (p_5). This indicates how long a service is reserved for. For example, the price could be charged per hour, per month or per year.
- `Instance sizes` (p_6). This refers to the amount of resources used at a given instant by the user. The size may vary from micro (in case of Amazon EC2) to small, medium, large, or extra large to something such as quadruple extra large provided by Amazon EC2.
- `Operating system` (p_7). This indicates that the operating system could be Linux or Windows.
- `Pricing` (p_8). This is the actual price for the usage of the cloud service.

- `Pricing sensitivity` to regions (p_9). This denotes if the price varies by region.
- `Subcontractors` (p_{10}). This indicates if subcontractors are present, and if so, what kind of services they provide. We represent this property using a binary bit array with three bits. The first bit is set to 1 if subcontractors are present. The second bit is set to 1 if the subcontractor provides computational or storage services. The third bit is set to 1 if the subcontractor provides security, privacy, or search related services.

While the same properties are considered for the B^{cloud}-tree, the property encoding in the data structure is slightly modified as follows:

4.2.2.2 Property Encoding

The property encoding transforms various types of CSP properties into decimal values. The encoding algorithms differs according to the types of the properties.

- **Service Type (p_1)**: Since services may be described in different ways, we employ the oneR mining algorithm, where the name of the service types is given as the input for the mining, to identify similar services. According to the mining result, service types falling into the same group will be assigned the same encoding. For example, if there are total 100 types of services identified, the domain of the encoding will be from the numerical value 1 to 100.
- **Properties with continuous values**: This category includes the properties: pricing (p_8), instance sizes (p_6) (e.g., requested storage capacity), pricing unit (p_5) and measurement unit (p_4). For such type of properties, we first partition the domain of the corresponding property into n ranges, where n is a tunable system parameter. Then we represent each range using one bit. If a CSP's property falls into a given range, the respective bit will be set to 1. Finally, we convert the binary representation to a decimal value.

  Example 1. Assume that the domain of storage space provided is divided into four ranges: [10G,∞), [1G,10G), [500M,1G), [0,500M). If the storage

capacity of a service provider is 800M to 2G, the second and the third bits will be set to 1, resulting in the encoding '0110'.

- **Properties with categorical values**: This category includes the following properties: security (p_2), quality of service (p_3), operating system (p_7), pricing sensitivity (p_9). Such properties are typically represented by a categorical value. For each of the categorical value, we assign a distinct numerical value. For example, the property "quality of service" can be described using "high", "medium", "poor". Correspondingly, we convert it to numbers "3"(high), "2"(medium), "1"(poor).

- **Relationship property**: This refers to the specifical property "Subcontractor" (P_{10}) which describes the relationship among the CSPs built upon subcontracting. We represent the relationship using a binary bit array with three bits. The first bit is set to 1 if subcontractors are present. The second bit is set to 1 if the subcontractor provides computational or storage services. The third bit is set to 1 if the subcontractor provides security, privacy, or search related services. Then, the binary value is converted into a decimal value.

In addition, if a service provider does not have specific value for certain properties, the encoding of that property will be set to the default value 0.

4.2.2.3 Index Construction for the B^+-tree

The novelty of the CSP-index lie in the construction of the index keys for service providers that can speed up the query processing. Intuitively, service providers with similar properties should be stored close to each other. In this way, once the broker identifies a candidate service provider in the index, the broker can quickly locate other candidates with closely matching properties since they are stored together. To achieve this, we propose the following key generation method that captures the similarity among service providers accurately while being efficient. The algorithm consists of three major steps. The first step is to encode the properties of the service provider, and the second step is to encode the relationships among service providers built by subcontracts. Finally, the service providers are

to be clustered based on the encoding to construct the index key. We elaborate each step in the following.

Step 1: Initial Grouping of Service Providers.

We first group service providers based on their service types, and functional properties. Precisely, we use algorithms such as the oneR mining algorithm over the entire set of service providers with the `service type` being the attribute over which the algorithm is run.

Step 2: Property Encoding.

For each type of service providers, we encode their properties. The overall idea is to use a bit array to store the values of the service provider's properties. The bit array is of the same size for every service provider. The bit array consists of 9 sections corresponding to the first 9 properties identified in Section 4.2.2.1. The number of bits used for each section is based on the domain of each property. The encoding differs according to the types of the properties.

For properties with continuous values, such as the cost, the storage capacity, we partition its domain into n ranges and represent each range using a bit.

Example 2. Assume that the domain of storage space provided is divided into four ranges: [10G,∞), [1G,10G), [500M,1G), [0,500M). If the storage capacity of a service provider is 800M to 2G, the second and the third bits will be set to 1, resulting in the encoding '0110'.

For properties with categorical values, we use a numerical value to represent it. For example, the property "service quality" can be described using "high", "medium", "poor". Correspondingly, we convert it to numbers "3"(high), "2"(medium), and "1"(poor). A typical example of a descriptive property is the privacy level. In addition, if a service provider does not have specific value for certain properties, the corresponding sections in the bit array will be set to '0'.

Step 3: Relationship Encoding.

The CSP-Index also maps any relationships among the service providers. These relationships often result because of service providers subcontracting to others. To this end, the subcontractors of a service provider are stored using a bitmap hash table. The length of the bitmap table is the same for each service provider. Then, the IDs of subcontractors are hashed and the corresponding bit is set to 1 in

the hash table. Just like the other properties of the service provider, this too is a bit array. It always consists of three bits - the first bit denotes whether the subcontractor is shared with another provider, the second bit denotes whether the services provided by the subcontractor affect the providers' computation or storage, and third bit denotes if the services provided affect the providers' privacy or security.

Step 4: Encoding Integration.

After the encoding, each service provider i has a set of binary strings mapping each property, and a bit-array as the result of the relationship encoding. Then, we generate the integrated encoding by concatenating the bits representing the service type with the XOR-ed results of the remaining property encodings (as shown in Equation 4.1).

$$E_{sp_i} = p_{1_i} || (p_{2_i} \oplus p_{3_i} \oplus p_{4_i} \oplus p_{5_i} \oplus p_{6_i} \oplus p_{7_i} \oplus p_{8_i} \oplus p_{9_i} \oplus p_{10_i}) \tag{4.1}$$

Performing the XOR operations on the strings helps condense the resulting string to a small size, while still preserves the similarity between service providers. The idea is illustrated by the following example.

Example 3. Consider the following small set of properties with divided value domains for example:

- Service type: 0001, 0002,...,1000
- Storage : [10G, ∞), [1G, 10G), [500M, 1G), [0, 500M)
- Cost: [50cents/min, 1 Dollar/min], [10 cents/min, 50 cents/min), [0, 10cents/min)
- Service quality: 3-High, 2-medium, 1-poor
- Privacy protection: 3-High, 2-medium, 1-poor

Suppose that a service provider SP_1 provides service type '0001', 800M to 2G storage space to each end user at 10 cents/min with medium service quality and medium privacy protection. The corresponding encoding of each property is: '0110'(storage), '010'(cost), '010'(service quality), '010'(privacy). The final integrated encoding for the service provider is then computed as follows:

$E_{sp_1} = 0001 || (0110 \oplus 010 \oplus 010 \oplus 010) = 00010100$

Step 5: Index Key Generation

We employ the k-means algorithm [87] to cluster all the service providers based on the Hamming distance between their final encodings, where k is equal to the number of service types. An auxiliary structure is maintained to store the cluster centers. Then, we leverage the idea of the iDistance [88] to generate the indexing key Key_{sp}. Using iDistance allows us to index the service points as data points, on a B^+-tree like index structure, which as described earlier is particularly suited for this problem space due to its wide adoption in commercial database systems.

To generate the index key, we first compute the Hamming distance (denoted as D_h) between the encoding of each service provider and its closest cluster center. Then, we add a scaling value S to D_h to form the index key Key_{sp}. The scaling value is used to partition the dimensional space into regions, where each region holds a cluster of points close to each other. Therefore, the scaling value depends upon the number of regions we aim to generate. The value S is directly proportional to the number of service types encountered, since the number of regions must be proportional to the number of service types. k is the constant used to stretch the index values so that the partitioning of the CSPs is easier. Equation 4.2 summarizes the key generation, where E_{sp_i} is the property encoding of service provider i, and E_{c_k} is the encoding of the cluster center c_k which is closest to the service provider i.

$$Key_{sp_i} = S \cdot k + D_h(E_{sp_i}, E_{c_k}) \tag{4.2}$$

Once the indexing key is generated, the insertion and deletion in the CSP-index resembles that in the B^+-tree. An example CSP-index is shown in Figure 4.2.

4.2.2.4 Indexing Key Generation for the B^{cloud}-tree

After the property encoding, a CSP is described by 10 decimal values. Next, we aim to generate the indexing keys based on the encoded properties of the CSP. Recall that the indexing key should have the property that CSPs with similar properties (i.e., the encoding value of the properties), should obtain closer indexing keys so that they will be placed closer in the index to speed up the subsequent service selection. To achieve that, we leverage one of the space-filling curves, the Z-curve [89]. The reason to choose the Z-curve is manifold. First, it has been implemented

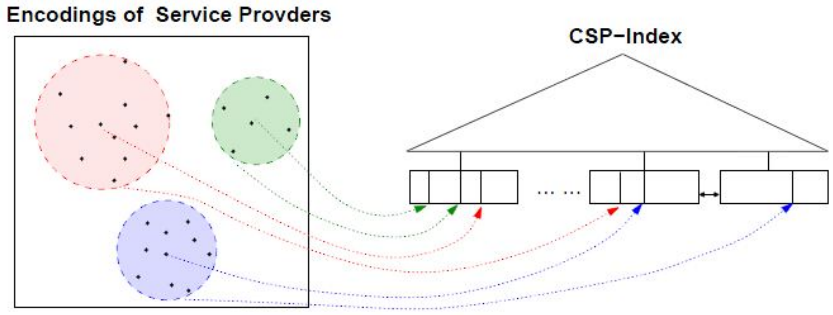

Figure 4.2. Structure of CSP-Index

in many commercial databases just like our base structure, the B^+-tree, which will help the deployment of our approach in practice. Second, the Z-curve has an important property that satisfies our need. Specifically, the Z-curve maps a data point in a multi-dimensional space to a single-dimensional value. The resulting one dimensional value has the proximity property that: for points which are closer in multi-dimensional space, they are very likely to receive closer one-dimensional value. For example, consider the data points (x,y), (x,y), which are near each other in the 2-dimensional space. Their Z-curve values are also closer to each other, which are xxx, yyy respectively. If we consider the coordinates of the data points as the properties of CSPs, the corresponding Z-curve values can then be used as the indexing keys. Notice that there are some cases where the Z-curve values cannot preserve the proximity properties. However, previous work [89] shows that Z-curve is quite effective (i.e., keeping the proximity property) for 10-dimensional data, which matches our case. In addition, Z-curve can be computed efficiently.

We now continue to elaborate how to employ the Z-curve to transform the encoded properties to the indexing key. There are three major steps.

- Given a service provider $CSP_1(p_1, ..., p_{10})$ and its property encodings denoted as $CSP_1(e_1, ..., e_{10})$, the broker converts each encoding into a binary representation: $e_i \rightarrow be_i$ $(1 \leq i \leq 10)$. The binary representation is normalized to be of the same length for all the properties. The normalization is done by taking the longest binary value as the standard length l of all binary values.

If an initial binary value does not have l bits, we fill up its beginning bit positions with 0.

- Then, we concatenate the binary encoding of the service type with the result of interleaving the binary values of the remaining 9 property's encodings. Let $[be_i]_j$ denote the j^{th} bit of be_i. The interleaving process is as follows, where the symbol "||" denote bit concatenation.

$$Z = [be_1]||[||_{j=2}^{l}(||_{i=1}^{10}[be_i]_j)] \tag{4.3}$$

- The last step is to convert Z into a decimal value, which would be the indexing key of the CSP_1: $Z \rightarrow \kappa$.

For better understanding, we step through the key generation processing using the following example.

Example 4. Consider the following small set of properties with divided value domains for example:

- Service type: 0001, 0002,...,1000
- Storage: [10G, ∞), [1G, 10G), [500M, 1G), [0, 500M)
- Pricing: [50cents/min, 1 Dollar/min], [10 cents/min, 50 cents/min), [0, 10cents/min)
- Quality of service: 3-High, 2-medium, 1-poor
- Security: 3-High, 2-medium, 1-poor

Suppose that a service provider CSP_1 provides service type '0001', 800M to 2G storage space to each end user at 10 cents/min with medium service quality and medium privacy protection. The corresponding encoding of each property is: '0110'(storage), '010'(pricing), '010'(service quality), '010'(privacy). After normalization, we have the following binary values for each property:

service type: '0001'

storage: '0110'

pricing: '0010'

privacy: '0010'

Next, we interleave the property encodings and obtain the Z-value '000001000 1111000'. Its corresponding decimal value is 1144, which will be used as the indexing key for this CSP.

Once the indexing key is generated, the insertion and deletion in the B^{cloud}-tree resembles that in the B^+-tree.

4.3 Service Selection

Indexing allows the broker to arrange the service providers in a fashion suitable for quick retrieval. The broker exploits this arrangement to retrieve the service providers that satisfy a user's query. In what follows, we first define the service selection query and then present the query algorithm.

4.3.1 The B^+-tree

The B^+-tree helps the broker arrange the service providers in a way that facilitates fast information retrieval, and provides the unique advantage that it is often used in many commercial databases. It thus forms the natural choice for indexing what can be considered as a database of CSPs.

4.3.1.1 Query Definition for using the B^+-tree

A user sends a service selection query to the broker which specifies what properties and values he/she expects from the service providers. A formal definition of the service selection query is given below.

Definition 3. A CSS query is in the form of $Q = \langle (QP_1 : D_1), (QP_2 : D_2), \ldots, (QP_k : D_k) \rangle$, where QP_i $(1 \leq i \leq k)$ is the property that the user requests the service provider to possess, and D_i describes the user expected values of property QP_i. $QP_1 \succeq QP_2 \succeq ... \succeq QP_k$ denotes the order of importance of the properties to the user. The result of the query will be the service provider that satisfies the most property requirements.

An example service selection query may look like: $Q = \langle (Service\_Type : 0001), (Cost : [50cents/min, 80cents/min]) \rangle$. At the user-end, a simplified GUI

with check boxes and drop down lists is provided, to facilitate property selection. The user needs not enter values for the entire list of properties, but only those that are relevant to him/her. The use of such GUI also ensures that the input for the Cloud broker is in a machine recognizable form so that the broker does not need to further clean the user's input before processing it.

4.3.1.2 CSS Query Algorithm for the B$^+$-tree

Querying the list of service providers consists of four phases: (1) query encoding; (2) k nearest neighbor search; (3) refinement; (4) consideration of special criteria. The query encoding converts the user query into the form of an index key of the CSP-index. Based on the query encoding, the k nearest neighbor search returns k candidate service providers whose index keys are similar to the query encoding, and hence may satisfy the query requirements. The last two phases further exam the properties of the candidate service providers and their relationship to find the best combination of service providers that addresses the user's needs. The pseudocode is reported in Algorithm 1. We detail the key steps of each phase as follows.

Step 1: Query Encoding.

Given a user query, the Cloud broker treats the property requirements in the query as properties of a new service provider, and encodes the properties in the same way as presented in Step 1 of the index construction in Section 4.2. The obtained property encoding is compared with the cluster centers stored in the auxiliary structure. The cluster center with the smallest hamming distance to the property encoding is selected. Then, we use Equation 4.2 to generate the index key value for this query.

Step 2: K-nearest neighbor search.

Based on the obtained index key value of the query (denoted as Key_q), we search the CSP-index to find the k candidate service providers whose property encodings are the k nearest neighbors of Key_q. The search starts from the root of the CSP-index. We follow the path that contains the entry with the smallest hamming distance to Key_q, until reach the leaf node of the CSP-index. Then, we examine the property encodings stored in the leaf nodes and find the k nearest values to Key_q. If the leaf node does not contain k entries, we expand the search

to its neighboring leaf nodes in both sides until k nearest neighbors are found.

Here, the chosen of the value of k, i.e., the number of neighbors to be considered, is critical to the overall performance. If too few neighbors are retrieved, we may not find the service provider which fully satisfy the query requirements. This is because the CSP-index stores service providers according to the similarity between all of their properties, in order to be versatile for different queries. A specific query usually focuses on a smaller set of properties, and hence the k nearest neighbors retrieved based on all properties may not contain the best solution regarding the querying properties. On the other hand, if k is too large, it will slow down the search process as well as the subsequent refinement phase. This value of k is therefore decided by a trial and error process. Based on extensive experiments, we set k to the 10^{th} of the total number of service providers.

Step 3: Refinement.

From the obtained candidate service providers, the refinement phase finds the service providers or the combinations of service providers that satisfy the query requirements.

The first step of the refinement is to further reduce the number of service providers that need to be fully examined. Specifically, we only consider top k_2 service providers in the k candidates obtained from the previous step, where $k_2 \ll k$. To find the top k_2 service providers, we create a new run-time index for k candidates based on only the properties listed in the query. The key of each candidate provider in the run-time index is computed based on new property encoding similar to Equation 4.1: $Key_{sp_i} = QP_1 \oplus QP_2 \oplus \ldots \oplus QP_n$, where QP_1, ..., QP_n are the properties listed in the query. For example, if the user lists only service type, instance size, and cost in the query, then only these three properties are used to form the key for the run-time index. The run-time index helps to quickly order the k candidate service providers according to their closeness to the query. Here the closeness is measured using the hamming distance between the index key and the query. Then, we execute a k_2-NN query to retrieve the top k_2 service providers.

The next step is to consider each querying property individually and sort the k_2 service providers in an ascending order of their hamming distances for that property. Suppose that there are n querying properties. We obtain n sorted lists of service providers corresponding to each property. Recall that the querying

properties are given in a decreasing order of importance in the query. Therefore, we start from the sorted list of the first (i.e., the most important) querying property, and apply a greedy algorithm to find the best combination of service providers that meets the user's service requirements.

In particular, we first select the service provider on top of the first sorted list. We remove the satisfied querying properties from subsequent process, and adjust partially satisfied querying properties. For example, if the user requests $20GB$ of storage space, while the selected service provider only has 5GB available, we adjust the querying property on storage space to 15GB (=20GB-5GB) and look for more service providers. As long as there are unsatisfied querying properties, we continue the selection of service providers by looking into the list of the next important unsatisfied property, and repeat the process. The selection process stops when all querying properties are satisfied. This set of service providers is sent to the next phase (discussed in Step 4) to verify possible collision and collusion among them caused by the shared subcontractors. If there exists any collision or collusion in current solution, Step 4 will return a ranking for this solution to indicate its collision and collusion degree. The higher the ranking, the less the collision or collusion. The service selection process will be repeated to find other possible combinations of service providers until a better solution cannot be found. The final output of the service selection algorithm may contain a ranked list of solutions to let end users to make the final decision.

Step 4: Consideration of Special Criteria

The possibility of a collision or collusion between service providers needs to be considered during their selection.

- *Collision* is the occurrence of a lack of a promised immutable resource due to the dependence of the selected service providers on the same contractor who promises the resource to all of them, not accounting for a simultaneous demand from all. For example, let us consider two service providers SP_1 and SP_2, who both lease 50 GB of storage space from a subcontractor $C1$. $C1$ can guarantee both $SP1$ and $SP2$ the 50GB provided there are no restrictions on the region in which the storage servers are located, being that it has a part of its servers in USA and the rest in China. When a user re-

quests a 100GB of storage space, $SP1$ and $SP2$ can fulfill this requirement together by depending on $C1$. However, if the user specifies that the servers be from a certain region, such as USA, then a collision occurs if the shared subcontractor is not taken into account.

- *Collusion* is the ability of service providers or subcontractors to derive more information or meta-information about the data stored on their resources, without the explicit permission or even the knowledge of the user. It occurs due to the providers or subcontractors cross-referencing two or more data sets. For example, if the selected service providers for a given user request, uses the same subcontractor that provides a record maintenance service, this subcontractor is then possible to identify that the data stored by these service providers are actually linked to the same user, and take advantage of such extra knowledge to infer user's information.

To detect collision and collusion, we verify the subcontractor encoding (p_{10}) of the service providers in the solution obtained from Step 3, to see if they have any subcontractor in common. The verification is conducted by computing $p_{10}^i \ AND \ p_{10}^j$ pairwise for each pair of service providers i, j in the solution. If any of the binary bit arrays that result from any of these ANDs have the first bit as 1, that means the service providers share a subcontractor. Otherwise, that means they do not have any subcontractor in common and the entire algorithm stops.

If there are shared subcontractors, we further quantify the degree of collision and collusion by assigning a ranking to the solution. Simply put, a solution that contains more shared subcontractors with more important properties will be assigned a lower rank. Specifically, we analyze the AND results of the subcontractor encodings (p_{10}) of each pair of service providers in the solution. If the AND result contains 1 in the second bit that represents storage space sharing, that means the corresponding pair of service providers may encounter a collision issue. If the AND result contains 1 in the third bit that represents security aspects, that means there may be a collusion issue. According to the order of the properties listed in the query, we know which property is more important, the storage space or the security aspect. The final ranking is then determined by the weighted sum of the number of collision and collusion, where larger weight is given to the more important property. The ranking is returned to Step 3 to check if there is a need to find

other solutions.

4.3.2 The B^{cloud}-tree

Despite being quick, the B$^+$-tree sacrifices some of the preciseness due to XOR-ing of the properties to form the index key. To overcome this issue we investigate another technique of encoding, which we refer to as the B^{cloud}-tree. The B^{cloud}-tree helps the broker arrange the service providers in a way that facilitates fast information retrieval. Recall that two common types of service selection are defined in Section 4.2.1: (i) the exact query; (ii) the interval query. Both types of queries can be answered using a three-phase query scheme: (1) generating query encoding; (2) searching the B^{cloud}-tree; (3) refine the search results and considering special criteria. The query encoding converts the user query into the form of intervals of indexing keys that cover CSPs who may satisfy the query. Based on the query encodings, we then search the B^{cloud}-tree to locate the candidate CSPs. The last step is to further examine the properties of the candidate CSPs and their relationship to find the best combination of service providers that addresses the user's needs. We detail the key steps of each phase in the following subsections.

4.3.2.1 Query Encoding for the B^{cloud}-tree

For both exact queries and interval queries, the first step is the query normalization which fills in the non-query properties using the property domains so that all the 10 properties are associated with a query domain. Since the exact query is a special case of an interval query, we define the query normalization on the interval query directly.

Definition 4. Query Normalization: Let Q=(QP_1:I_1, ..., QP_k:I_k) be an exact or an interval query. For each property $p_j \notin QP_i$ ($1 \leq i \leq k$), create a query interval $NP_j = (p_j : D_1)$ where D_j is the domain of the property p_j. Then, a normalized Q' will be in the form of Q=(QP_1:I_1, ..., QP_k:I_k, NP_1, ..., NP_n), where n is the total number of properties not listed in Q.

Observe that the number of combinations of property values in the query range could be huge. The computation of indexing keys for all the combinations will be

very time consuming. Instead of doing these one-by-one computation, it would be more efficient if we can find out the boundaries of the corresponding indexing keys, and execute interval queries rather than single point queries in the B^{cloud}-tree. After examining the properties of the Z-curve, we notice that the minimum and maximum indexing key values of a query are reached by taking all the lower and upper boundaries of the query properties respectively. The formal definition of the boundaries of the query indexing keys is given below.

Definition 5. *Let Q=(*QP_1*:*$[V_1^l, V_1^u]$*, ..,* QP_k*:*$[V_k^l, V_k^u]$*,* $NP_1 : [[V_{k+1}^l, V_{k+1}^u]$*, ...,* $NP_n : [V_{10}^l, V_{10}^u]$*) be a normalized interval query, where* V_j^l *and* V_j^u *denote the lower and upper bounds of the interval respectively. Let* κ^l *be the indexing key computed from (*QP_1*:*V_1^l*, ...,* QP_k*:*V_k^l*,* $NP_1 : V_{k+1}^l$*, ...,* $NP_n : V_{10}^l$*), and let* κ^u *be the indexing key computed from (*QP_1*:*V_1^u*, ...,* QP_k*:*V_k^u*,* $NP_1 : V_{k+1}^u$*, ...,* $NP_n : V_{10}^u$*). Then, for any CSP=(*p_1*,..,* p_{10}*) whose* $p_j \in [V_j^l, V_j^u]$ *(for all* $1 \leq j \leq 10$*), its indexing key* κ_{csp} *will be in the following range:* $\kappa^l \leq \kappa_{csp} \leq \kappa^u$

The range of the query indexing keys obtained from Definition 5 could still be too large due to false positives, where some indexing key values are within the boundaries but the corresponding points are not in the query range. As a tradeoff between query accuracy and efficiency, we take three values of indexing keys: the minimum, the median, and the maximum. We then conduct the k nearest neighbor search for the these three values as elaborated in the following subsection.

4.3.2.2 CSS Query in the B^{cloud}-tree

The output from the last step will contain three indexing key values: minimum boundary (denoted as κ_{min}), median (κ_{med}), and maximum (κ_{max}). For each of the query indexing key, we search from the root of the B^{cloud}-tree and locate the leaf node that contains the range of keys including this query indexing key. It may happen that the leaf node does not have the exact value of the querying key since there is not such a service provider who matches the query requirements. In this case, we will find the closest value to the querying indexing key and start the search. From the located starting value (either the exact querying indexing key or the closest one), we will find its k nearest neighbors. For κ_{min}, we look for k neighbors which have key values greater than it; for κ_{med}, we look for k

neighbors from both sides; for κ_{max}, we look for k neighbors that are smaller than it. During the search, if the leaf node containing the query indexing key does not have k entries, we expand the search to its neighboring leaf nodes along the search direction until k nearest neighbors are found.

Here, the chosen of the value of k, i.e., the number of neighbors to be considered, is critical to the overall performance. If too few neighbors are retrieved, we may not find the service provider which fully satisfy the query requirements. This is because the B^{cloud}-tree stores service providers according to the similarity between all of their properties, in order to be versatile for different queries. A specific query usually focuses on a smaller set of properties, and hence the k nearest neighbors retrieved based on all properties may not contain the best solution regarding the querying properties. On the other hand, if k is too large, it will slow down the search process as well as the subsequent refinement phase. This value of k is therefore decided by a trial and error process. We will present the tuning of the k in Chapter 7, where our experimental evaluation is provided.

Chapter 5

Policy-Based Node Selection in MapReduce

5.1 Overview

Once the service is selected by the users, the users often need to select worker nodes and storage nodes for their tasks. The need for node selection is especially pertinent when MapReduce tasks are involved. As mentioned in the introduction, the component of GABE which helps users with node selection is referred to as the PARiNgS framework. PARiNgS focuses specifically on MapReduce tasks, but can be adapted to node selection for any tasks in a Cloud. Its effectiveness is easily understood in a hybrid cloud, where the worker nodes may belong to both private customer-controlled clouds or public clouds wheree the customers do not manage or control the underlying cloud infrastructure, network, servers, operating systems, storage, or even individual application capabilities. The PARiNgS framework enables users to express security requirements that are applied to their data scattered across the cloud and processed by MapReduce functions. We assume that the Master node is always under the control of the cloud service provider, and therefore can be fully trusted. Although we do not trust the workers to the same degree as the Master since they maybe easily compromised, we assume that workers in the cloud are cooperative and willing to follow the PARiNgS protocols unless they are compromised.

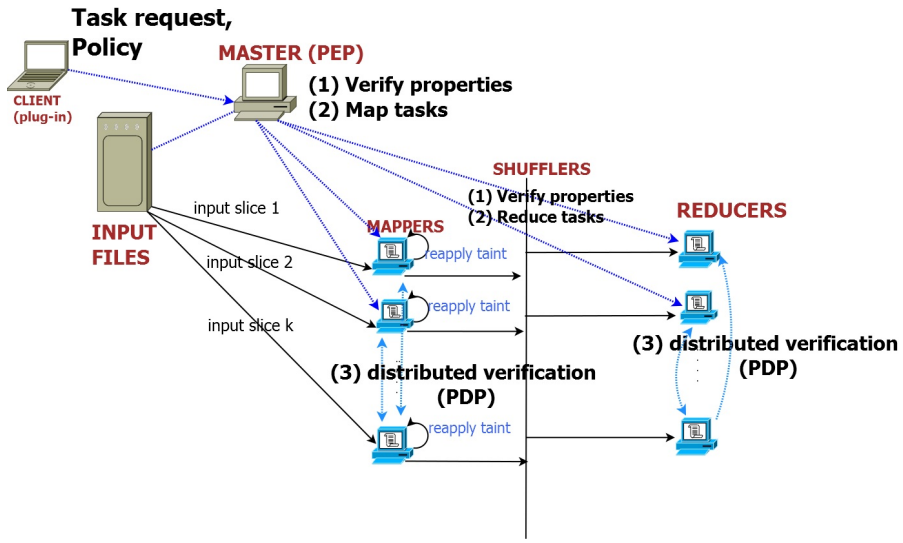

Figure 5.1. Overview of the main PARiNgS framework

Figure 5.1 illustrates the main components of the PARiNgS framework and their interactions. As shown, the framework includes the following components: (1) a client plugin for policy specification. Following the model proposed by [90] we assume that policy specification is supported by the availability of a service, the semantic based policy management service (SBPMS). The SBPMS enables the cloud users to specify, edit and manage their access policies. The access policies express conditions against properties of the workers managing the users' data. SBPMS also conducts a conflict resolution on the policies to resolve possible conflicts and export the policies into the Master node. Therefore, SBPMS acts as policy administration point (PAP) and a policy information point (PIP). (2) A policy enforcement point (PEP) module, installed at the Master node, which coordinates distributed evaluation of users' policies. The policy decision point includes a tainting module, that pre-processes and prepares the data for policy compliant computation, at the intermediate steps of the MapReduce processing. The evaluation of the policies is interleaved with the scheduling process of the MapReduce tasks. (3) One or more "target" worker nodes, whose properties are evaluated, so as to assess their eligibility to access portion of the input for MapReduce processing. (4) Multiple workers across the public and private clouds that, in synch with the Master node, act as verifiers and establish "target" worker's abilities to satisfy the policies, defining a distributed Policy Decision Point for the architecture (PDP).

The execution of PARiNgS-MapReduce programs is similar to MapReduce programs, the difference being that nodes processing MapReduce tasks on a given client input are selected according to clients' policies, while preserving the original execution flow. For instance, consider the classic word count example, wherein a user would request that only non-sensitive words can be processed by public clouds located in the USA. In this case, the Master node, upon receiving the policy, completes two preliminary steps: (1) it pre-processes the input data to partition them according to the user's policies (that is, it separates sensitive and non-sensitive words, and partitions the data in "policy-compliant" partitions), and (2) it triggers the property verification protocols, to identify nodes capable to carry out the required MapReduce tasks that also meet the users' policy requirements (e.g. it verifies which nodes are public and which ones are in the USA). The protocols for property verification are property-specific. With the exception of few selected properties that are fully under control and verified by the Master, the verification protocols of properties of unknown veracity[1], to ensure scalability and avoid dangerous bottlenecks. Upon verification of the policy's satisfiability, the data distributed by the Master becomes available to the eligible nodes, starting the process at the mappers. As the mappers complete their tasks, the control is back to the Master. The Master, upon reshuffling the data, in accordance with the application's logic and MapReduce routines, re-assigns the intermediate data to worker nodes for reduce tasks, after having re-applied the policies. To keep track of the input data and its related policies, as they go through intermediate stages, data may be tainted as part of the pre-processing stage.

5.2 Policy Model In MapReduce

We propose the following policy model for use with the PARiNgS framework. The model allows a user to specify confidentiality requirements on their data being processed by MapReduce. It is formally defined as follows.

Definition 6. *PARiNgS consists of the following components:*

[1]The Master may have knowledge of nodes' claims, e.g. a node's location, but not have verified them.

- *A set U of users, a set $Node$ of worker nodes each of which is described by a set of node properties $W = \{w_1, ..., w_n\}$, a set of datasets D. Each dataset is described by a set of data attributes $A = \{a_1, ..., a_m\}$, and a condition language $\mathcal{L}_c$.*

- *A PARiNgS policy of a user u consists of a set of rules and a rule combining algorithm: $P_u = \{\{R_1, R_2, \ldots, R_k\}, RCA\}$, where RCA is a rule combining algorithm.*

- *A PARiNgS rule consists of a target and a condition: $R = Target \wedge Cond$.*

  - *$Target$ is a function $F : A \rightarrow [true, false]$ on data attributes defined by a language $\mathcal{L}_c$.*
  - *$Cond$ is a function $F : W \rightarrow [true, false]$ on node properties defined by $\mathcal{L}_c$.*

In PARiNgS users are cloud clients, which assign MapReduce tasks to the computing environment. Users own portions of data D, which they protect through policies. Specifically, a policy is defined as a collection of rules. Each rule specifies that the $Target$ can be accessed under the condition $Cond$. The $Target$ uses a function of data attributes to describe a subset of user data, which is corresponding to a partition of the original input data. Data partitioning, as required for parallel MapReduce computation, is executed according to pre-processing algorithms, described in Section 5.3.1. The $Cond$ is expressed against properties associated with worker nodes. Examples of workers' properties that may instill confidence on the MapReduce process are support for cryptographic storage, support of file level access control, location, nodes' capabilities and proprietary nature of the node. The exact descriptives of the properties depend on the underlying MapReduce architecture being deployed. The properties are further elaborated in Appendix A.

Due to the consideration of overhead introduced by the policies, we cannot define arbitrary complex rules to be evaluated in MapReduce. For efficiency, we only consider the following forms of targets and conditions that are specified using the language $\mathcal{L}_c$.

Definition 7. *Let $att_1, \ldots, att_n$ denote a set of data or worker attributes, let $\theta \in \{<, \leq, =, \neq, >, \geq, \subset, \subseteq, \supseteq, \supset\}$, and let v denote a constant. Then, $att_i \theta v$ is an* **atomic condition** *of $\mathcal{L}_c$.*

Definition 8. *A function in $\mathcal{L}_c$ is defined as follows:*

- *An atomic function is a function of $\mathcal{L}_c$.*
- *Let c_i and c_j be functions of $\mathcal{L}_c$; then $c_i \wedge c_j$ or $c_i \vee c_j$ is a function of $\mathcal{L}_c$.*

Example 5. Consider a MapReduce task that processes files of records that contain multiple fields such as "age" and "country". Suppose that the file owner has the following privacy and security requirements: (1) All records belong to people younger than 14 must be processed by servers with cryptographic storage capability and located in USA ; (2) Records of other people who were born in USA must be processed by server in USA. The requirements can be specified using a PARiNgS policy as follows:

$P_1.R_1$: Target[age$\leq$ 14], Condition[location=USA $\bigwedge$ AES=enabled];
$P_1.R_2$: Target[age>14$\bigwedge$country=USA], Condition[location=USA].

We now introduce policy evaluation process. Given a user-defined policy, the policy decision point (PDP) takes a set of data items and a worker node's properties as input, and output a decision that has a value in {Permit, Deny, NotApplicable}. While the meaning of *Permit* and *Deny* are self-explanatory, in our case, a policy is *NotApplicable* if a certain rule is not applicable to the partition of data being processed. For instance, in the word count example, a rule applicable to proper nouns may not be applicable to rest of the words. The evaluation involves analysis of rules and combining of decisions from rules as elaborated in the following.

Definition 9. *Let d denote a data item requested by a worker node with a set of properties w_1, w_2, ..., w_k. A rule $R = Target \wedge Cond$ yields one of the following effects:*

- *Permit: If $d \in Target$ and $Cond(w_1, ..., w_k) = true$, then the worker node is allowed to access the data item d.*
- *Deny: If $d \in Target$ and $Cond(w_1, ..., w_k) = false$, then the worker node is denied the access to the data item d.*

- *NotApplicable: If $d \notin Target$, then the rule is not applicable to this request.*

Since one policy may contain multiple rules and each rule may return different effects regarding the same request, we adopt the permit-override rule combining algorithm to generate the final policy effect. This rule combining algorithm resolves all possible conflicts among policies in a simple and efficient manner.

Definition 10. *Let R_1, ..., R_k be a set of rules in a policy P_u. Let $E(P)$ denote the effect of the policy, and let $E(R_i)$ denote the effect of rule R_i. Then $E(P)$ is computed as follows:*

- *$E(P_u) = \vee_{R_i \in \mathcal{R}} E(R_i)$, where $\mathcal{R}$ is the set of applicable rules.*
- *$E(P) = NotApplicable$, if $\forall R_i$, $E(R_i) = NotApplicable$, where $1 \leq i \leq k$.*

The permit-override rule coming algorithm can speed up the policy evaluation process since the evaluation can stop as soon as one rule is evaluated "Permit".

5.3 Data Pre-processing for Policy Enforcement

To facilitate the enforcement of users' policies on their data, we propose two types of data pre-processing: data partitioning and tainting, which are tightly bound with the MapReduce tasks.

5.3.1 Data Partitioning

We aim to seamlessly integrate policy enforcement with the MapReduce workflow. The first task for policy enforcement is to partition data in accordance with the user's policy for proper processing. This task includes two main steps: (1) partitioning the data into buckets; (2) bucket merging.

A user submits a set of data items along with a policy to be enforced. Let $P = \{R_1, ..., R_n\}$ denote a policy, and let d_i denote a generic data item in the user input D. The Master node (specifically the partitioning module) partitions data according to the policy. For each rule, the Master first creates an empty bucket of the capacity that will be used to store assigned data items. The size of the bucket, is pre-defined according to the scheduling algorithm followed by the

partitioner. The basic idea is to distribute data items among all the buckets as evenly as possible while satisfying the policy constraints in order to balance the workload. The Master starts scanning the data items. Each data item will be evaluated against the rules in P. Not all the rules will need to be evaluated for all the data. That is, if R_j is the first rule that is applicable to d_i, i.e., d_i satisfies the Target component in R_j, the Master node will insert d_i to the bucket of rule R_j. Note that it is sufficient to identify one applicable rule for a data item because of the adoption of the permit-override rule combining algorithm. More specifically, Workers that satisfy any rule applicable to the data item are allowed to process the data item.

In case the bucket of rule R_j is full, the Master node will continue to evaluate d_i against other remaining rules until an applicable rule with sufficient bucket space is identified. If none of the applicable rules for d_i has sufficient space in the bucket, the Master node will add one more bucket to the last identified applicable rule and assign d_i to it. In addition, if none of the rules applicable to d_i, d_i will be inserted to a separate bucket marked as "FreeBucket". Data items in this FreeBucket can be assigned to any worker nodes. At the end, we will the following obtain $n + 1$ data partitions while each partition may be associated with multiple buckets.

- $D_1 = \{d|Target_{R_1}(d) = true, d \in D\}$: Data items in buckets of rule R_1:
- $D_2 = \{d|Target_{R_2}(d) = true, d \in D\} - D_1$: Data items in buckets of rule R_2. Note that for some data items that satisfy both R_1 and R_2, if they have already been included in the buckets of R_1, they will not be considered again in the buckets of R_2.
- $D_3 = \{d|Target_{R_2}(d) = true, d \in D\} - D_2 - D_1$: Data items satisfy rule in buckets of rule R_3.
- ... ...
- $D_n = \{d|Target_{R_n}(d) = true, d \in D\} - \cup_{i=1}^{n-1} D_i$: Data items in Buckets of rule R_n:
- $D_{n+1} = D - \cup_{i=1}^{n} D_i$: Data items in FreeBuckets.

According to the partitioning algorithm, the obtained results of data partitioning satisfy the following property, which ensures that the data partitions are non-overlapping:

Property 1. $\forall D_i,\ D_j,$ *where* $1 \leq i, j \leq n+1,\ D_i \cap D_j = \emptyset$

After data partitioning, some partitioning may contain very few data items. In this case, we will conduct bucket merging. For each pair of buckets that are less than half full, we compare the associated rule conditions. Buckets associated with same or non-conflicting rule conditions are directly merged. The following property presents the formal merging rules.

Property 2. *Let* B_i *denote a bucket of rule* R_i*, and* B_j *denote a bucket of rule* R_j*. Let* BZ *denote the bucket size.* B_i *and* B_j *can be merged if they satisfy the following conditions:*

- $|B_i| < \frac{1}{2}BZ$ *and* $|B_j| < \frac{1}{2}BZ$
- *There exists at least one worker node with properties* $w_1, ..., w_k$*, so that* $Cond_{R_i}(w_1, ..., w_k) = true$ *and* $Cond_{R_j}(w_1, ..., w_k) = true$*.*

Example 6. Consider policy P_1 in Example 5. If data partitions of rule R_1 and R_2 both have one bucket that is less than half full, we can merge the two buckets directly. This is because the conditions in R_1 and R_2 are not conflicting with each other.

5.3.2 Data Tainting

Observe that in many MapReduce applications, the output data no longer possesses the same set of attributes as the input, which causes difficulty in determining the proper access control policies for the intermediate results. An illustrative example is given below.

Example 7. In the example of calculating the area of facilities, the input to the mapper is the length, width and height of individual rooms. Suppose that the user has a policy P on the initial input file:

P: Target[$length > 10$], Condition[*(location="US WEST"*$\wedge$*crypto="3DES")*]

Protocol: Location_Verification(CharToEcho, Nonce)
Input: *CharToEcho* is a random character, or set of characters,
Nonce is pseudorandom number used to identify freshness of the echo, typically the time
1. **Worker**–>**Verifier**: Claim %Geo-Location of physical host
2. **Sub-Verifier**–>**Worker**: Request= CharToEcho, Nonce
3. **Worker**–>**Sub-Verifier**: Response= Echo(CharToEcho, Nonce)
4. **Sub-Verifier**: Output = Response.time()- Request.time()
5. **Sub-Verifier**–>**Verifier**: Output
5. **Verifier**: If (Output)==val && Geo-Location(Sub-Verifier)=x Then
6. Geo-Location(Worker)=$location(val)_i$
7. For Each Geo-Location(Sub-Verifier)==x
8. Geo-Location(Worker)=$\bigcap location(val)_i$ %triangulation of the possible locations
9. If Geo-Location(Worker)== Claim
10. **Verifier**: Worker(Verified)< −−True
11. If Worker(Verified)==True Then **Verifier**−− >**Worker**: $\{k_w, Sig(k_w)\}$ %send key share
12. Else **Verifier**−− >**Fail**

Figure 5.2. Location Verification

After the first round of computation, we obtain the area of rooms as the output which does not have the same type and unit compared to the input. Therefore, we cannot directly tell from the output area whether the policy target still applies to the data for the next round of processing.

To address the above issue, we adopt data tainting techniques to the data being protected by a given rule. In particular, we follow a guideline that output data items should be protected in the same way as the input data, i.e., under the protection of the same policy rule. In order to track the relationship between the input and output data, we let the Master apply the taint [91, 92] to the input data before assigning the mapping tasks. Tainting results in a modification of the input data type to add a new property to the data. In the above example, tainting consists of modifying the input length (usually defined as int or float) to an object. The object includes a data portion with the original integer or float, along with a Boolean portion called *tainted*, which shows whether the object is tainted or not and a string portion called *taint* which is used to set a particular taint value. After the map round, mappers may also apply or re-apply the taint in either of the following two cases: a) when the input to the mapper is tainted, or b) when the user inserts, deletes or revises existing policies.

5.4 Policy Evaluation

5.4.1 Distributed Policy Evaluation Protocol

Policy enforcement requires verification of worker nodes' properties against the users policies. The straightforward method is to ask the Master to complete the verification. However, given the large number of workers in the cloud, using the Master to verify all the workers for each requested task would negatively impact the effectiveness of the distributed nature of MapReduce. Also, the Master may not have collections of all worker nodes' properties up to date [10, 11]. To avoid the possible performance bottleneck caused by such centralized approach, we propose a distributed policy evaluation protocol. The key idea of the distributed policy evaluation protocol is to utilize ordinary worker nodes to conduct collaborative verification of other worker nodes' properties. A worker's properties will be verified by multiple peers randomly selected at each round. Each peer verifies a randomly selected property at a time, which not only speeds up the verification process but also reduces the collusion probability. In what follows, we elaborate the detailed protocol.

Given a policy $P = \{R_1, \ldots, R_k\}$ (see Section 5.2), the Master partitions the data into $D_i \in \{D_1, \ldots, D_k\}$ and associate each partition with a rule in the policy using the algorithm described in Section 5.3.1. For each partition D_i with rule R_i, the Master selects candidate worker nodes whose properties ($\{w_1, \ldots, w_n\}$) may satisfy rule R_i, i.e., $Cond_{R_i}(w_1, \ldots, w_n)$=true. Also the Master randomly selects verifier nodes to help verify partial properties of the candidate worker nodes. The worker nodes to be verified collect verification tokens returned from the verifiers. If a worker node received enough verification tokens, it would be able to access the corresponding data partition.

Our distributed verification protocol is developed based on the secret sharing scheme [93]. Secret sharing refers to methods for distributing a secret amongst a group of participants, each of which is allocated a share of the secret. We adopt the (k, n) threshold scheme by Shamir[93]. Such a scheme splits a secret S into n partial secrets so that k, with $k < n$, partial secrets are required to reconstruct S. In our scheme, the secret is a cryptographic key, and each verifier node returns a share of the secret (or "token") if and only if the condition is evaluated true. Only when

the worker node obtains the minimal number of shares required for verification, the worker node is permitted to access the data partition. Our protocol is given below:

1. Given a rule R_i, the Master extracts the properties $\{w_1, \ldots, w_z\}$ to be checked against the corresponding conditions $c_{i1}...c_{in}$ of rule R_i. If one of the properties in $w_1, \ldots, w_z$ is under the Master's control, such as the bucket sizes, then the Master pre-selects the workers who already meet this property.

2. The Master randomly selects n nodes as the verifier nodes (denoted as $v_1, \ldots, v_n$). n is such that for each of the z properties, there are at least t verifier nodes, with $zt \geq n$. To ensure a fair number of verifiers without overloading the system, the Master determines the number of verifications and verifiers according to a probabilistic scheme, as discussed in Subsection 5.4.2.

3. The Master computes $h(w_1|\ldots|w_z)$, and uses it as a secret for the Shamir's Secret Sharing Scheme, obtaining S, which is in turn used as a cryptographic key. It then breaks S into $k_1, \ldots, k_n$ the key shares.

4. Upon generating the shares, the Master assigns the verification tasks to nodes. Nodes are randomly selected, in order to limit the risk of collusion. In other words, the same node is not constantly used for verifying the same property of some other node. To further reduce the chance of collusion, a node cannot verify the same property twice, even for different workers, for a same job. This is easily dealt with by maintaining a hash table of workers at the Master node, and marking the hash table for each property being verified.

   Each verification task includes verifying a particular property against the corresponding condition. The verifier is given the name of the task, such as location verification, file access security verification and the condition(s) c_j to be verified[2].

[2]Notice that verifier nodes are capable of carrying out the verification protocols, by means of methods that are either locally installed at the nodes or that can be dynamically invoked.

5. The Master sends the worker the message $\{\{IDList, task_{verlist}\}, r, E_S(f), Sig(h(ID|r|f)\}$ where $task_{verlist}$ denotes the verification tasks (e.g. location verification, security property verification) assigned to the corresponding verifiers in the IDList, $E_S(f)$ denotes the encrypted data file (or the address where the encrypted data is located), r a random number. $Sig(h(ID|r|f))$ denotes a signed hash of all the message content, computed using a collision resistant hash function. The signed hash is to ensure authentication and integrity of the request.

6. The worker sends a claim for each property being verified, e.g. w_i, to the corresponding verifiers tasked for property w_i's verification. The message includes a nonce non, and a hash of the claim, to be verified by v_i, $h(c_k|non|w_s)$. For example, if three verifiers are in charge of location verification, the worker sends a claim $h(c_k|non|location)$ to all the three verifiers.

7. Upon verification of integrity of the message, v_i and the worker engage in a property-specific verification protocol. Then, as a protocol is successfully completed, a key share k_i is released to the worker. An example of verification protocol to check a node's reported location is outlined in Figure 2. In this scheme, the verifier analyzes the round trip time (RTT) of a message sent by the verifier to estimate its source, using multiple sub-verifiers. The protocol adopts an approach similar to the multi-lateration scheme used for distance verification in mobile ad-hoc networks. The multilateration scheme uses distance-bounding protocols [94] which are in turn dependent on estimating the location based of a node on response time from the node. Other security protocols, such as support for AES or CBC mode are verified using NIST verification programs [95]. Lack of space prevents us from further discussing these protocols. A detailed discussion with pointers to the case of the Azure deployment is reported in Appendix B.

8. Upon completing zt successful verifications (steps 6 and 7), the worker obtains enough shares $k_1, \ldots, k_{zt-1}$ to reconstruct the key S and access the file by decrypting $E_s(f)$.

5.4.2 Number of Nodes Needed for Verification

Since some nodes may be corrupted and may not send back the requested secret share or may send the share without properly verifying the worker's claim, we aim to figure out the minimum number of nodes needed to obtain at least t successful verifications at probability greater than a given threshold ρ. Further, to ensure that each individual property is verified a sufficient amount of times, we request that each property is checked at least $\frac{t}{n_p}$ times, by $\frac{t}{n_p}$ different verifiers.

Let n denote the total number of nodes with secret shares, and n_p is the number of properties of the worker node to be verified. Assume that same number of secret shares are generated for each property of the worker. Let p_i denote the probability of a node N_i being corrupted. The probability P_s of receiving secret shares from t non-corrupted nodes is computed as follows.

$$P_s = \left(\begin{array}{c} \frac{n}{n_p} \\ \frac{t}{n_p} \end{array} \right)^{n_p} p_i^{n-t}(1-p_i)^t \tag{5.1}$$

Equation 5.1 can be understood as follows. There are $\frac{n}{n_p}$ nodes secret shares for any given property of the worker node. Since each property needs to be verified $\frac{t}{n_p}$ times, there are $C(\frac{n}{n_p}, \frac{t}{n_p})$ different ways to choose $\frac{t}{n_p}$ from $\frac{n}{n_p}$ nodes. Then $C(\frac{n}{n_p}, \frac{t}{n_p})^{n_p}$ is the total number of ways to choose t nodes from n nodes while guarantee that there are $\frac{t}{n_p}$ nodes per property. The number of combinations is then multiplied with the probability for t nodes not being corrupted, i.e., $p_i^{n-t}(1-p_i)^t$.

The corruption probability p_i could be obtained from statistic data. Given known values of t and p_i, we can compute the minimum value of n by resolving the following inequality which contains only n as an unknown value.

$$P_s \geq \rho \tag{5.2}$$

5.5 Discussion and Conclusion

PaRingS is designed to provide users with stronger controls on the nodes managing users' potentially sensitive data. By restricting access to nodes with desirable

properties without burdening users with complex configuration tasks, clients can gain confidence on the trustworthiness of the computation. Needless to say, our solution tackles only a small problem in the complex space of secure and customized computation in cloud settings, and has some limitations itself. For instance, even if successfully verified, there is no guarantee that if some functional properties are tested (like cryptographic support) the worker will actually behave as expected. For instance, the worker may demonstrate support of robust data encryption, and yet later apply a weaker algorithm, if at all. In order to solve this problem, depending upon the sensitivity of the data, the verification process for the appropriate property can be carried out at random intervals on the data processed by the workers. Further, since properties are verified by at least t nodes, the nodes can cheat the verification process if enough of them collude with each other. To mitigate this risk, we can strengthen our current approach by ensuring the verifiers selected do not consist of any loops [96]. Or, we can employ incentivized supervision schemes commonly used in peer-to-peer networks [97, 98]. In addition, the generic problem of correctness of the actual computation tasked to the site is also not addressed. Correctness of computation is complementary to the problem dealt with in this paper. Correctness of computation can be checked using various approaches, e.g. [99, 80].

Chapter 6

Cloud Information Accountability

6.1 Introduction

Once the service and the nodes are selected, GABE enables users to protect their data. While cloud computing enables easy data processing, details of the services provided for it are abstracted from the users who no longer need to be experts of technology infrastructure. Moreover, users may not know the machines which actually process and host their data. While enjoying the convenience brought by this new technology, users also start worrying about losing control of their own data. The data processed on clouds is often outsourced, leading to a number of issues related to accountability, including the handling of personally identifiable information. Such fears are becoming a significant barrier to the wide adoption of cloud services [42].

To allay users' concerns, it is essential to provide an effective mechanism for users to monitor the usage of their data in the cloud. For example, users need to be able to ensure that their data is handled according to the service level agreements made at the time they sign on for services in the cloud. Conventional access control approaches developed for closed domains such as databases and operating systems, or approaches using a centralized server in distributed environments, are not suitable, due to the following features characterizing cloud environments. First, data handling can be outsourced by the direct cloud service provider (CSP) to other entities in the cloud and theses entities can also delegate the tasks to others, and so on. Second, entities are allowed to join and leave the cloud in a flexible manner.

As a result, data handling in the cloud goes through a complex and dynamic hierarchical service chain which does not exist in conventional environments.

To overcome the above problems, we propose a novel approach, namely Cloud Information Accountability (CIA) framework, based on the notion of *information accountability* [36]. Unlike privacy protection technologies which are built on the hide-it-or-lose-it perspective, information accountability focuses on keeping the data usage transparent and trackable. Our proposed CIA framework provides end-to-end accountability in a highly distributed fashion. One of the main innovative features of the CIA framework lies in its ability of maintaining light-weight and powerful accountability that combines aspects of access control, usage control and authentication. By means of the CIA, data owners can track not only whether or not the service level agreements are being honored, but also enforce access and usage control rules as needed. Associated with the accountability feature, we also develop two distinct modes for auditing: *push mode* and *pull mode*. The push mode refers to logs being periodically sent to the data owner or stakeholder while the pull mode refers to an alternative approach whereby the user (or another authorized party) can retrieve the logs as needed.

The design of the CIA framework presents substantial challenges, including uniquely identifying CSPs, ensuring the reliability of the log, adapting to a highly decentralized infrastructure, etc. Our basic approach toward addressing these issues is to leverage and extend the programmable capability of JAR (Java ARchives) files to automatically log the usage of the users' data by any entity in the cloud. Users will send their data along with any policies such as access control policies and logging policies that they want to enforce, enclosed in JAR files, to cloud service providers. Any access to the data will trigger an automated and authenticated logging mechanism local to the JARs. We refer to this type of enforcement as "strong binding" since the policies and the logging mechanism travel with the data. This strong binding exists even when copies of the JARs are created, thus the user will have control over his data at any location. Such decentralized logging mechanism meets the dynamic nature of the cloud but also imposes challenges on ensuring the integrity of the logging. To cope with this issue, we provide the JARs with a central point of contact which forms a link between them and the user. It records the error correction information sent by the JARs, which allows it to monitor the

loss of any logs from any of the JARs. Moreover, if a JAR is not able to contact its central point, any access to its enclosed data will be denied.

Currently, we focus on image files since images represent a very common content type for end-users and organizations (as is proven by the popularity of Flickr [100, 101]), and are increasingly hosted in the cloud as part of the storage services offered by the utility computing paradigm featured by cloud computing. Further, images often reveal social and personal habits of users, or are used for archiving important files from organizations. In addition, our approach can handle personal identifiable information provided they are stored as image files (they contain an image of any textual content, for example the SSN stored as a .jpg file).

We tested our CIA framework in a cloud testbed, the Emulab testbed [102], with Eucalyptus as middleware [103]. Our experiments demonstrate the efficiency, scalability and granularity of our approach. In addition, we also provide a detailed security analysis and discuss the reliability and strength of our architecture in the face of various non-trivial attacks, launched by malicious users or due to compromised Java Running Environment (JRE).

In summary, our main contributions are as follows:

- We propose a novel automatic and enforceable logging mechanism in the cloud. To our knowledge, this is the first time a systematic approach to data accountability through the novel usage of JAR files is proposed.

- Our proposed architecture is platform-independent and highly decentralized, in that it does not require any dedicated authentication or storage system in place.

- We go beyond traditional access control in that we provide a certain degree of usage control for the protected data after it is delivered to the receiver.

- We conduct experiments on a real cloud testbed. The results demonstrate the efficiency, scalability and granularity of our approach. We also provide a detailed security analysis and discuss the reliability and strength of our architecture.

In this section, we present an overview of the Cloud Information Accountability

(CIA) framework and discuss how the CIA framework meets the design requirements discussed in the previous section.

The Cloud Information Accountability (CIA) framework proposed in this work conducts automated logging and distributed auditing of relevant access performed by any entity, carried out at any point of time at any cloud service provider. It has two major components: *logger* and *log harmonizer*.

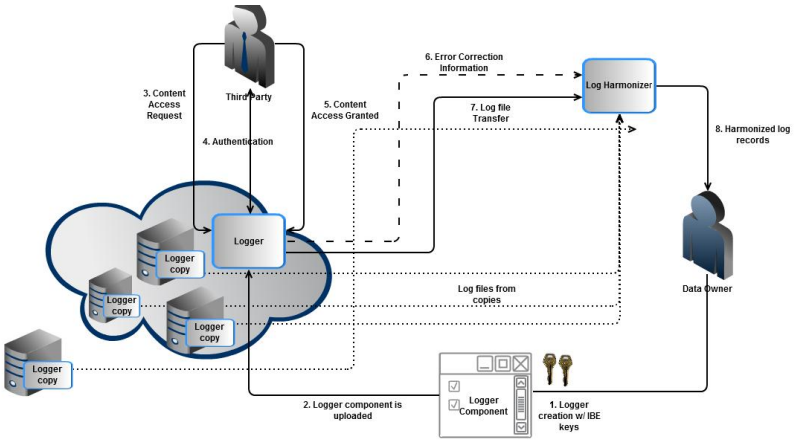

Figure 6.1. Overview of Cloud Information Accountability Framework

6.1.1 Major Components

There are two major components of the CIA, the first being the logger, and the second being the log harmonizer. The logger is the component which is strongly coupled with the user's data, so that it is downloaded when the data is accessed, and is copied whenever the data is copied. It handles a particular instance or copy of the user's data and is responsible for logging access to that instance or copy. The log harmonizer forms the central component which allows the user access to the log files. It resides on the Cloud broker, along with the rest of the CSS framework.

The logger is strongly coupled with user's data (either single or multiple data items). Its main tasks include automatically logging access to data items that it contains, encrypting the log record using the public key of the content owner, and periodically sending them to the log harmonizer. It may also be configured to

ensure that access and usage control policies associated with the data are honored. For example, a data owner can specify that user X is only allowed to view but not to modify the data. The logger will control the data access even after it is downloaded by user X.

The logger requires only minimal support from the server (e.g., a valid Java virtual machine installed) in order to be deployed. The tight coupling between data and logger, results in a highly distributed logging system, therefore meeting our first design requirement. Furthermore, since the logger does not need to be installed on any system or require any special support from the server, it is not very intrusive in its actions, thus satisfying our fifth requirement. Finally, the logger is also responsible for generating the error correction information for each log record and send the same to the log harmonizer. The error correction information combined with the encryption and authentication mechanism, provides a robust and reliable recovery mechanism, therefore meeting the third requirement.

The log harmonizer is responsible for auditing. It supports two auditing strategies: *push* and *pull*. Under the push strategy, the log file is pushed back to the data owner periodically in an automated fashion. The pull mode is an on-demand approach, whereby the log file is obtained by the data owner as often as requested. These two modes allow us to satisfy the aforementioned fourth design requirement. In case there exist multiple loggers for the same set of data items, the log harmonizer will merge log records from them before sending back to the data owner. The log harmonizer is also responsible for handling log file corruption. In addition, the log harmonizer can itself carry out logging in addition to auditing. Separating the logging and auditing functions improves the performance. The logger and the log harmonizer are both implemented as lightweight and portable JAR files. The JAR file implementation provides automatic logging functions, which meets the second design requirement.

6.1.2 Data Flow

The overall CIA framework, combining data, users, logger and harmonizer is sketched in Figure 6.1. At the beginning, each user creates a pair of public and private keys based on Identity-Based Encryption (IBE) [104] (step 1 in Figure 6.1).

This IBE scheme is a Weil-pairing based IBE scheme, which protects us against one of the most prevalent attacks to our architecture as described in Section 6.4. Using the generated key, the user will create a logger component which is a JAR file, to store its data items. The JAR file includes a set of simple access control rules specifying whether and how the cloud servers, and possibly other data stakeholders (users, companies) are authorized to access the content itself. Then, he sends the JAR file to the cloud service provider (CSP) that he subscribes to. To authenticate the CSP to the JAR (step 3-5 in Figure 1), we use OpenSSL based certificates, wherein a trusted certificate authority certifies the CSP. In the event that the access is requested by a user, we employ SAML-based authentication [105], wherein a trusted identity provider issues certificates verifying the user's identity based on his username.

Once the authentication succeeds, the service provider (or the user) will be allowed to access the data enclosed in the JAR. Depending on the configuration settings defined at the time of creation, the JAR will provide usage control associated with logging, or will provide only logging functionality. As for the logging, each time there is an access to the data, the JAR will automatically generate a log record, encrypt it using the public key distributed by the data owner, and store it along with the data (step 6 in Figure 6.1). The encryption of the log file prevents unauthorized changes to the file by attackers. The data owner could opt to reuse the same key pair for all JARs or create different key pairs for separate JARs. Using separate keys can enhance the security (detailed discussion is in Section 6.4) without introducing any overhead except in the initialization phase. In addition, some error correction information will be sent to the log harmonizer to handle possible log file corruption (step 7 in Figure 6.1). To ensure trustworthiness of the logs, each record is signed by the entity accessing the content. Further, individual records are hashed together to create a chain structure, able to quickly detect possible errors or missing records. The encrypted log files can later be decrypted and their integrity verified. They can be accessed by the data owner or other authorized stakeholders at any time for auditing purposes with the aid of the log harmonizer (step 8 in Figure 6.1).

As discussed in Section 6.4, our proposed framework prevents various attacks such as detecting illegal copies of users' data. Note that our work is different from

traditional logging methods which use encryption to protect log files. With only encryption, their logging mechanisms are neither automatic nor distributed. They require the data to stay within the boundaries of the centralized system for the logging to be possible, which is however not suitable in the cloud.

Example 8. Considering Example 1, Alice can enclose her photographs and access control policies in a JAR file and send the JAR file to the cloud service provider. With the aid of control associated logging (called *AccessLog* in Section 6.2.2), Alice will be able to enforce the first four requirements and record the actual data access. On a regular basis, the push-mode auditing mechanism will inform Alice about the activity on each of her photographs as this allows her to keep track of her clients' demographics and the usage of her data by the cloud service provider. In the event of some dispute with her clients, Alice can rely on the pull-mode auditing mechanism to obtain log records.

6.2 Automated Logging Mechanism

In this section, we first elaborate on the automated logging mechanism and then present techniques to guarantee dependability.

6.2.1 The Logger structure

We leverage the programmable capability of JARs to conduct automated logging. A logger component is a nested Java JAR file which stores a user's data items and corresponding log files. As shown in Figure 6.2, our proposed JAR file consists of one outer JAR enclosing one or more inner JARs.

The main responsibility of the outer JAR is to handle authentication of entities which want to access the data stored in the JAR file. In our context, the data owners may not know the exact CSPs that are going to handle the data. Hence, authentication is specified according to the servers' functionality (which we assume to be known through a lookup service), rather than the server's URL or identity. For example a policy may state that Server X is allowed to download the data if it is a storage server. As discussed below, the outer JAR may also have the access control functionality to enforce the data owner's requirements, specified as Java

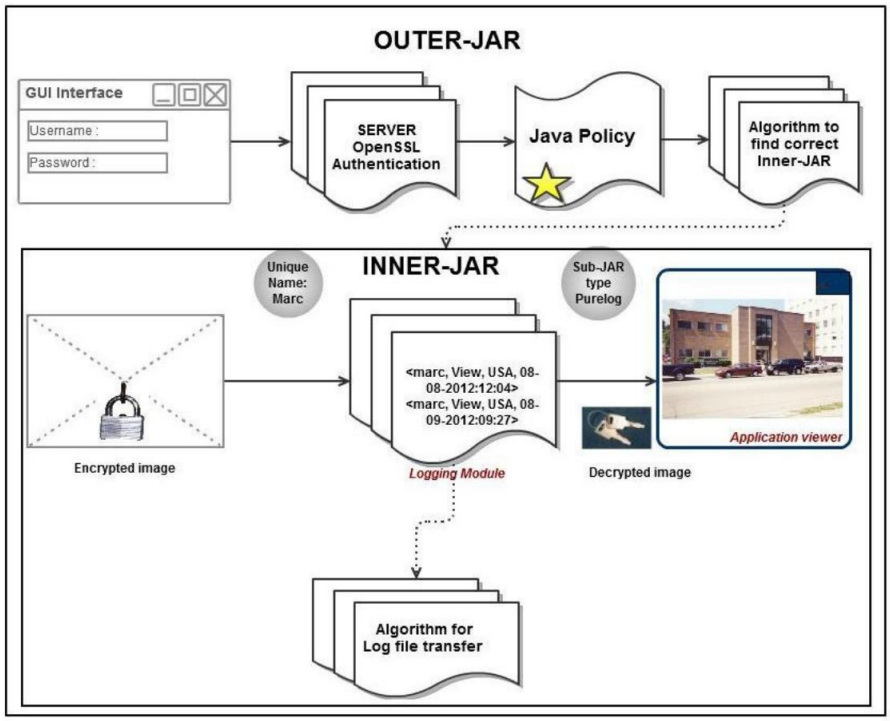

Figure 6.2. The Structure of the JAR File

policies, on the usage of the data. A Java policy specifies which permissions are available for a particular piece of code in a Java application environment. The permissions expressed in the Java policy are in terms of File System Permissions. However, the data owner can specify the permissions in user-centric terms as opposed to the usual code-centric security offered by Java, using Java Authentication and Authorization Services. Moreover, the outer JAR is also in charge of selecting correct inner-JAR according to the identity of the entity who requests the data.

Example 9. Consider Example 1. Suppose that Alice's photographs are classified into three categories according to the locations where the photos were taken. The three groups of photos are stored in three inner JAR J_1, J_2 and J_3, respectively, associated with different access control policies. If some entities are allowed to access only one group of the photos, say J_1, the outer JAR will just render the corresponding inner JAR to the entity based on the policy evaluation result.

Each inner JAR contains the encrypted data, class files to facilitate retrieval of log files and display enclosed data in a suitable format, and a log file for each

encrypted item. We support two options.

- *PureLog*: Its main task is to record every access to the data. The log files are used for pure auditing purpose.
- *AccessLog*: It has two functions: logging actions and enforcing access control. In case an access request is denied, the JAR will record the time when the request is made. If the access request is granted, the JAR will additionally record the access information along with the duration for which the access is allowed.

The two kinds of logging modules allow the data owner to enforce certain access conditions either pro-actively (in case of AccessLogs) or reactively (in case of PureLogs). For example, services like billing may just need to use PureLogs. AccessLogs will be necessary for services which need to enforce service level agreements such as limiting the visibility to some sensitive content at a given location.

To carry out these functions the inner JAR contains a class file for writing the log records, another class file which corresponds with the log harmonizer, the encrypted data, a third class file for displaying or downloading the data (based on whether we have a PureLog, or an AccessLog), and the public key of the IBE key pair that is necessary for encrypting the log records. The outer JAR may contain one or more inner JARs, a class file for authenticating the servers or the users, another class file finding the correct inner JAR, a third class file which checks the JVM's validity using oblivious hashing, a fourth class file for managing the GUI for user authentication and finally the Java Policy

6.2.2 Log Record Generation

Log records are generated by the logger component. Logging occurs at any access to the data in the JAR, and new logs are appended sequentially, in order of creation LR=$\langle r_1, \ldots, r_k \rangle$. Each record r_i is encrypted individually and appended to the log file. In particular, a log record takes the following form:

$$r_i = \langle ID, Act, T, Loc, h((ID, Act, T, Loc)|r_{i-1}| \ldots |r_1), sig \rangle$$

Here, r_i indicates that an entity identified by ID has performed an action Act on the user's data at time T at location Loc. The component $h((ID, Act, T, Loc)|r_{i-1}|\ldots|r_1)$ corresponds to the checksum of the records preceding the newly inserted one, concatenated with the main content of the record itself (we use I to denote concatenation). The checksum is computed using a collision-free hash function [106]. The component sig denotes the signature of the record created using the server's public key. If more than one file is handled by the same logger, an additional $ObjID$ field is added to each record. An example of log record for a single file is shown below.

Example 10. Suppose that a cloud service provider with ID Kronos, located in USA, read the image in a JAR file (but did not download it) at 4:52 pm on May 20, 2011. The corresponding log record is:

⟨`Kronos, View, 2011-05-29 16:52:30, USA, 45rftT024g, r94gm30130ff`⟩.

The location is converted from the IP address for improved readability.

To ensure the correctness of the log records, we verify the access time, locations as well as actions. In particular, the time of access is determined using the Network Time Protocol [107] to avoid suppression of the correct time by a malicious entity. The location of the cloud service provider (CSP) can be determined using IP address. The JAR can perform an IP lookup and use the range of the IP address to find the most probable location of the CSP. More advanced techniques for determining location can also be used [108]. Similarly, if a trusted timestamp management infrastructure can be set up or leveraged, it can be used to record the timestamp in the accountability log [109]. The most critical part is to log the actions on the users' data. In the current system, we support four types of actions, i.e., *Act* has one of the following four values: *view*, *download*, *timed_access*, and *Location-based_access*. For each action, we propose a specific method to correctly record or enforce it depending on the type of the logging module, which are elaborated as follows.

- ***View***: The entity (e.g., the cloud service provider) can only read the data but is not allowed to save a raw copy of it anywhere permanently. For this

type of action, the PureLog will simply write a log record about the access, while the AccessLogs will enforce the action through the enclosed access control module. Recall that the data is encrypted and stored in the inner JAR. When there is a view-only access request, the inner JAR will decrypt the data on the fly and create a temporary decrypted file. The decrypted file will then be displayed to the entity using the Java application viewer in case the file is displayed to a human user. Presenting the data in the Java application viewer disables the copying functions using right click or other hot keys such as PrintScreen. Further, to prevent the use of some screen capture software, the data will be hidden whenever the application viewer screen is out of focus. The content is displayed using the headless mode in Java on the command line when it is presented to a CSP.

- ***Download***: The entity is allowed to save a raw copy of the data and the entity will have no control over this copy neither log records regarding access to the copy.

  If PureLog is adopted, the user's data will be directly downloadable in a pure form using a link. When an entity clicks this download link, the JAR file associated with the data will decrypt the data and give it to the entity in raw form. In case of AccessLogs, the entire JAR file will be given to the entity. If the entity is a human user, he/she just needs to double click the JAR file to obtain the data. If the entity is a CSP, it can run a simple script to execute the JAR.

- ***Timed_access***: This action is combined with the view only access, and it indicates that the data is made available only for a certain period of time.

  The Purelog will just record the access starting time and its duration, while the AccessLog will enforce that the access is allowed only within the specified period of time. The duration for which the access is allowed is calculated using the Network Time Protocol (NTP). To enforce the limit on the duration, the AccessLog records the start time using the NTP, and then uses a timer to stop the access. Naturally, this type of access can be enforced only when it is combined with the *View* access right and not when it is combined with the *Download*.

- ***Location-based_access***: In this case, the PureLog will record the location of the entities. The AccessLog will verify the location for each of such access. The access is granted and the data is made available only to entities located at locations specified by the data owner.

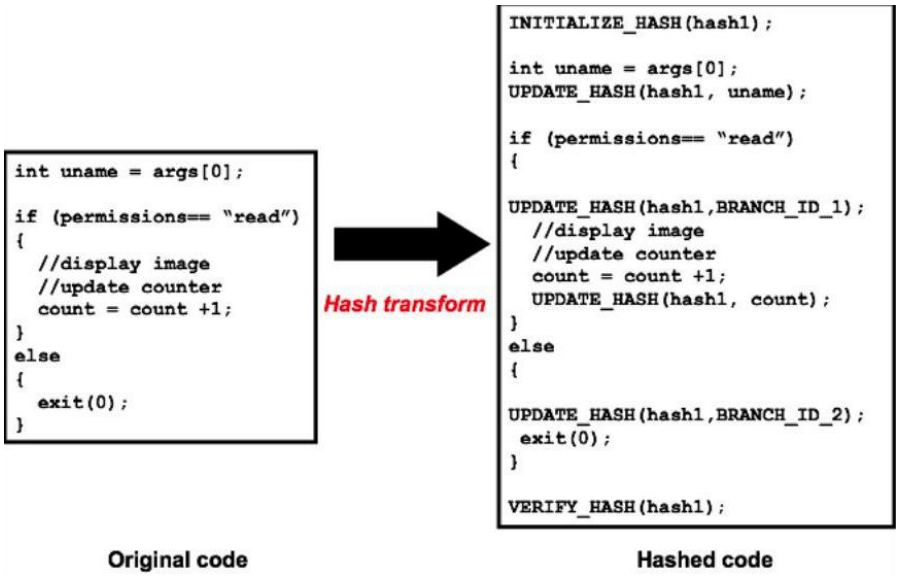

Figure 6.3. Oblivious Hashing applied to the logger

6.2.3 Dependability of Logs

In this section, we discuss how we ensure the dependability of logs. In particular, we aim to prevent the following two types of attacks. First, an attacker may try to evade the auditing mechanism by storing the JARs remotely, corrupting the JAR or trying to prevent them from communicating with the user. Second, the attacker may try to compromise the JRE used to run the JAR files.

JARs Availability

To protect against attacks perpetrated on offline JARs, the CIA includes a log harmonizer which has two main responsibilities: to deal with copies of JARs and to recover corrupted logs.

Each log harmonizer is in charge of copies of logger components containing the same set of data items. The harmonizer is implemented as a JAR file. It

does not contain the user's data items being audited, but consists of class files for both a server and a client processes to allow it to communicate with its logger components. The harmonizer stores error correction information sent from its logger components, as well as the user's IBE decryption key, to decrypt the log records and handle any duplicate records. Duplicate records result from copies of the user's data JARs. Since user's data is strongly coupled with the logger component in a data JAR file, the logger will be copied together with the user's data. Consequently, the new copy of the logger contains the old log records with respect to the usage of data in the original data JAR file. Such old log records are redundant and irrelevant to the new copy of the data. To present the data owner an integrated view, the harmonizer will merge log records from all copies of the data JARs by eliminating redundancy.

For recovering purposes, logger components are required to send error correction information to the harmonizer after writing each log record. Therefore, logger components always ping the harmonizer before they grant any access right. If the harmonizer is not reachable, the logger components will deny all access. In this way, the harmonizer helps prevent attacks which attempt to keep the data JARs offline for unnoticed usage. If the attacker took the data JAR offline after the harmonizer was pinged, the harmonizer still has the error correction information about this access and will quickly notice the missing record.

In case of corruption of JAR files, the harmonizer will recover the logs with the aid of Reed-Solomon error correction code [110]. Specifically, each individual logging JAR, when created, contains a Reed-Solomon based encoder. For every n symbols in the log file, n redundancy symbols are added to the log harmonizer in the form of bits. This creates an error correcting code of size $2n$ and allows the error-correction to detect and correct n errors. We choose the Reed-Solomon code as it achieves the equality in the Singleton Bound [111], making it a maximum distance separable code and hence leads to an optimal error correction.

The log harmonizer is located at a known IP address. Typically, the harmonizer resides at the user's end as part of his local machine, or alternatively, it can either be stored in a user's desktop or in a proxy server.

Log Correctness

For the logs to be correctly recorded, it is essential that the JRE of the system on which the logger components are running remain unmodified. To verify the integrity of the logger component, we rely on a two-step process: 1) we repair the JRE before the logger is launched and any kind of access is given, so as to provide guarantees of integrity of the JRE. 2) We insert hash codes, which calculate the hash values of the program traces of the modules being executed by the logger component. This helps us detect modifications of the JRE once the logger component has been launched, and are useful to verify if the original code flow of execution is altered.

These tasks are carried out by the log harmonizer and the logger components in tandem with each other. The log harmonizer is solely responsible for checking the integrity of the JRE on the systems on which the logger components exist before the execution of the logger components is started. Trusting this task to the log harmonizer allows us to remotely validate the system on which our infrastructure is working.

The repair step is itself a two-step process where the harmonizer first recognizes the Operating System being used by the cloud machine and then tries to reinstall the JRE. The OS is identified using nmap commands. The JRE is reinstalled using commands such as `sudo apt install` for Linux-based systems or `$ <jre>.exe [lang=] [s] [IEXPLORER=1] [MOZILLA=1] [INSTALLDIR=:] [STATIC=1]` for Windows-based systems.

The logger and the log harmonizer work in tandem to carry out the integrity checks during run time. These integrity checks are carried out using Oblivious Hashing (OH) [112]. OH works by adding additional hash codes into the programs being executed. The hash function is initialized at the beginning of the program, the hash value of the result variable is cleared and the hash value is updated every time there is a variable assignment, branching or looping. An example of how the hashing transforms the code is shown in figure 6.3.

As shown, the hash code captures the computation results of each instruction and computes the oblivious-hash value as the computation proceeds. These hash codes are added to the logger components when they are created. They are present in both the inner and outer JARs. The log harmonizer stores the values for the

hash computations. The values computed during execution are sent to it by the logger components. The log harmonizer proceeds to match these values against each other to verify if the JRE has been tampered with. If the JRE is tampered, the execution values will not match.

Adding OH to the logger components also adds an additional layer of security to them in that any tampering of the logger components will also result in the OH values being corrupted.

6.3 End-to-End Auditing Mechanism

In this section we describe our distributed auditing mechanism including the algorithms for data owners to query the logs regarding their data.

6.3.1 Push and Pull Mode

To allow users to be timely and accurately informed about their data usage, our distributed logging mechanism is complemented by an innovative auditing mechanism. We support two complementary auditing modes: (i) push mode; (ii) pull mode.

Push Mode

In this mode, the logs are periodically pushed to the data owner (or auditor) by the harmonizer. The push action will be triggered by either type of the following two events: one is that the time elapses for a certain period according to the temporal timer inserted as part of the JAR file; the other is that the JAR file exceeds the size stipulated by the content owner at the time of creation. After the logs are sent to the data owner, the log files will be dumped, so as to free the space for future access logs. Along with the log files, the error correcting information for those logs is also dumped.

This push mode is the basic mode which can be adopted by both the PureLog and the AccessLog, regardless of whether there is a request from the data owner for the log files. This mode serves two essential functions in the logging architecture:

1) It ensures that the size of the log files does not explode and 2) It enables timely detection and correction of any loss or damage to the log files.

Concerning the latter function, we notice that the auditor, upon receiving the log file, will verify its cryptographic guarantees, by checking the records' integrity and authenticity. By construction of the records, the auditor, will be able to quickly detect forgery of entries, using the checksum added to each and every record.

Pull Mode

This mode allows auditors to retrieve the logs anytime when they want to check the recent access to their own data. The pull message consists simply of an FTP pull command, which can be issues from the command line. For naive users, a wizard comprising a batch file can be easily built. The request will be sent to the harmonizer, and the user will be informed of the data's locations and obtain an integrated copy of the authentic and sealed log file.

6.3.2 Algorithms

Pushing or pulling strategies have interesting trade-offs. The pushing strategy is beneficial when there are a large number of accesses to the data within a short period of time. In this case, if the data is not pushed out frequently enough, the log file may become very large, which may increase cost of operations like copying data (see Section 8). The pushing mode may be preferred by data owners who are organizations and need to keep track of the data usage consistently over time. For such data owners, receiving the logs automatically can lighten the load of the data analyzers. The maximum size at which logs are pushed out is a parameter which can be easily configured while creating the logger component. The pull strategy is most needed when the data owner suspects some misuse of his data; the pull mode allows him to monitor the usage of his content immediately. A hybrid strategy can actually be implemented to benefit of the consistent information offered by pushing mode and the convenience of the pull mode. Further, as discussed in Section 7, supporting both pushing and pulling modes helps protecting from some non-trivial attacks.

The log retrieval algorithm for the Push and Pull modes is outlined in Algorithm

6.4.

The algorithm presents logging and synchronization steps with the harmonizer in case of PureLog. First, the algorithm checks whether the size of the JAR has exceeded a stipulated size or the normal time between two consecutive dumps has elapsed. The size and time threshold for a dump are specified by the data owner at the time of creation of the JAR. The algorithm also checks whether the data owner has requested a dump of the log files. If none of these events has occurred, it proceeds to encrypt the record and write the error-correction information to the harmonizer.

The communication with the harmonizer begins with a simple handshake. If no response is received, the log file records an error. The data owner is then alerted through emails, if the JAR is configured to send error notifications.Once the handshake is completed, the communication with the harmonizer proceeds, using a TCP/IP protocol. If any of the aforementioned events (i.e., there is request of the log file, or the size or time exceeds the threshold) has occurred, the JAR simply dumps the log files and resets all the variables, to make space for new records.

In case of AccessLog, the above algorithm is modified by adding an additional check after step 6. Precisely, the AccessLog checks whether the CSP accessing the log satisfies all the conditions specified in the policies pertaining to it. If the conditions are satisfied, access is granted; otherwise, access is denied. Irrespective of the access control outcome, the attempted access to the data in the JAR file will be logged.

Our auditing mechanism has two main advantages. First, it guarantees a high level of availability of the logs. Second, the use of the harmonizer minimizes the amount of workload for human users in going through long log files sent by different copies of JAR files. For a better understanding of the auditing mechanism, we present the following example.

Example 11. With reference to Example 1, Alice can specify that she wants to receive the log files once every week, as it will allow her to monitor the accesses to her photographs. Under this setting, once every week the JAR files will communicate with the harmonizer by pinging it. Once the ping is successful, the file transfer begins. On receiving the files, the harmonizer merges the logs and sends

them to Alice. Besides receiving log information once every week, Alice can also request the log file anytime when needed. In this case, she just need to send her pull request to the harmonizer which will then ping all the other JARs with the "pull" variable to 1. Once the message from the harmonizer is received, the JARs start transferring the log files back to the harmonizer.

6.4 Security Discussion

We now analyze possible attacks to our framework. Our analysis is based on a semi-honest adversary model by assuming that a user does not release his master keys to unauthorized parties, while the attacker may try to learn extra information from the log files. We assume that attackers may have sufficient Java programming skills to disassemble a JAR file and prior knowledge of our CIA architecture. We first assume that the JVM is not corrupted, followed by a discussion on how to ensure that this assumption holds true.

6.4.1 Copying Attack.

The most intuitive attack is that the attacker copies entire JAR files. The attacker may assume that doing so allows accessing the data in the JAR file without being noticed by the data owner. However, such attack will be detected by our auditing mechanism. Recall that every JAR file is required to send log records to the harmonizer. In particular, with the push mode, the harmonizer will send the logs to data owners periodically. That is, even if the data owner is not aware of the existence of the additional copies of its JAR files, he will still be able to receive log files from all existing copies. If attackers move copies of JARs to places where the harmonizer cannot connect, the copies of JARs will soon become inaccessible. This is because each JAR is required to write redundancy information to the harmonizer periodically. If the JAR cannot contact the harmonizer, the access to the content in the JAR will be disabled. Thus the logger component provides more transparency than conventional log files encryption; it allows the data owner to detect when an attacker has created copies of a JAR, and it makes offline files unaccessible.

6.4.2 Disassembling Attack.

Another possible attack is to disassemble the JAR file of the logger and then attempt to extract useful information out of it or spoil the log records in it. Given the ease of disassembling JAR files, this attack poses one of the most serious threats to our architecture. Since we cannot prevent an attacker in possession of the JARs, we rely on the strength of the cryptographic schemes applied to preserve the integrity and confidentiality of the logs.

Once the JAR files are disassembled, the attacker is in possession of the public IBE key used for encrypting the log files, the encrypted log file itself, and the *.class files. Therefore, the attacker has to rely on learning the private key or subverting the encryption to read the log records.

To compromise the confidentiality of the log files, the attacker may try to identify which encrypted log records correspond to his actions by mounting a chosen plaintext attack to obtain some pairs of encrypted log records and plain texts. However, the adoption of the Weil Pairing algorithm ensures that the CIA framework has both chosen ciphertext security and chosen plaintext security in the random oracle model [104]. Therefore, the attacker will not be able to decrypt any data or log files in the disassembled JAR file. Even if the attacker is an authorized user, he can only access the actual content file but he is not able to decrypt any other data including the log files which are viewable only to the data owner.[1] From the disassembled JAR files, the attackers are not able to directly view the access control policies either, since the original source code is not included in the JAR files. If the attacker wants to infer access control policies, the only possible way is through analyzing the log file. This is, however, very hard to accomplish since, as mentioned earlier, log records are encrypted and breaking the encryption is computationally hard.

Also, the attacker cannot modify the log files extracted from a disassembled JAR. Would the attacker erase or tamper a record, the integrity checks added to each record of the log will not match at the time of verification (see Section 6.2.2

[1]Notice that we do not consider the attack on the log harmonizer component, since it is stored separately in either a secure proxy or at the user-end and the attacker typically cannot access it. As a result, we assume that the attacker cannot extract the decryption keys from the log harmonizer.

for the record structure and hash chain), revealing the error. Similarly, attackers will not be able to write fake records to log files without going undetected, since they will need to sign with a valid key and the chain of hashes will not match. The Reed-Solomon encoding used to create the redundancy for the log files, the log harmonizer can easily detect a corrupted record or log file.

Finally, the attacker may try to modify the Java classloader in the JARs in order to subvert the class files when they are being loaded. This attack is prevented by the sealing techniques offered by Java. Sealing ensures that all packages within the JAR file come from the same source code [113]. Sealing is one of the Java properties, which allows creating a signature that does not allow the code inside the JAR file to be changed. More importantly, this attack is stopped as the JARs check the classloader each time before granting any access right. If the classloader is found to be a custom classloader, the JARs will throw an exception and halt. Further, JAR files are signed for integrity at the time of creation, to avoid that an attacker writes to the JAR. Even if an attacker can read from it by disassembling it - he cannot "reassemble" it with modified packages. In case the attacker guesses or learns the data owner's key from somewhere, all the JAR files using the same key will be compromised. Thus, using different IBE key pairs for different JAR files will be more secure and prevent such attack.

6.4.3 Man-in-the-Middle Attack.

An attacker may intercept messages during the authentication of a service provider with the certificate authority, and reply the messages in order to masquerade as a legitimate service provider. There are two points in time that the attacker can replay the messages. One is after the actual service provider has completely disconnected and ended a session with the certificate authority. The other is when the actual service provider is disconnected but the session is not over, so the attacker may try to renegotiate the connection. The first type of attack will not succeed since the certificate typically has a timestamp which will become obsolete at the time point of reuse. The second type of attack will also fail since renegotiation is banned in the latest version of OpenSSL and cryptographic checks have been added.

6.4.4 Compromised JVM Attack.

An attacker may try to compromise the JVM.

To quickly detect and correct these issues we discussed in Section 5.3 how to integrate Oblivious Hashing (OH) to guarantee the correctness of the JRE [112] and how to correct the JRE prior to execution, in case any error is detected. OH adds hash code to capture the computation results of each instruction and computes the oblivious-hash value as the computation proceeds. These two techniques allow for a first quick detection of errors due to malicious JVM, therefore mitigating the risk of running subverted JARs. To further strengthen our solution, one can extend OH usage to guarantee the correctness of the class files loaded by the JVM.

Figure 6.4. Push and Pull PureLog Mode

Require: *size*: maximum size of the log file specified by the data owner, *time*: maximum time allowed to elapse before the log file is dumped, *tbeg*: timestamp at which the last dump occurred, *log*: current log file, *pull*: indicates whether a command from the data owner is received.

1: Let TS(NTP) be the network time protocol timestamp
2: $pull = 0$
3: $rec := \langle UID, OID, AccessType, Result, Time, Loc \rangle$
4: $curtime := TS(NTP)$
5: $lsize := sizeof(log)$ //current size of the log
6: **if** $((cutime - tbeg) < time) \&\& (lsize < size) \&\& (pull == 0)$ **then**
7: $log := log + ENCRYPT(rec)$ // ENCRYPT is the encryption function used to encrypt the record
8: $PING$ to CJAR //send a PING to the harmonizer to check if it is alive
9: **if** $PING$-CJAR **then**
10: PUSH $RS(rec)$ // write the error correcting bits
11: **else**
12: EXIT(1) // error if no PING is received
13: **end if**
14: **end if**
15: **if** $((cutime - tbeg) > time) || (lsize >= size) || (pull \neq 0)$ **then**
16: // Check if PING is received
17: **if** $PING$-CJAR **then**
18: PUSH log //write the log file to the harmonizer
19: $RS(log) := NULL$ // reset the error correction records
20: $tbeg := TS(NTP)$ // reset the $tbeg$ variable
21: $pull := 0$
22: **else**
23: EXIT(1) // error if no PING is received
24: **end if**
25: **end if**

Chapter 7

Performance Study

We have prototyped GABE's service selection framework (introduced in Chapter 4) as a trusted system separate from the Cloud on a Sony Vaio F series Laptop, with a 8GB (4GBx2) DDR3-SDRAM-1333, 640GB Hardrive and a Intel Core i7-2820QM quad-core processor (2.30GHz) with Turbo Boost up to 3.40GHz. All the components are implemented as C programs. We implemented GABE's node selection framework (introduced in Chapter 5) on top of Microsoft's Azure framework. We used the Daytona runtime developed by Microsoft Research for Azure. Daytona is an iterative MapReduce runtime, designed to support wide class of data analytics and machine learning algorithms [10]. An affinity group is created to co-host the hosted service, which is comprised of the actual MapReduce code, in the same geographical location. For our deployment, we selected the West US affinity group. A node can be either a"Master" or a"Slave". The instance count of the Slave role is updated as per the anticipated load and the number of cores allocated to the Azure project. In our testbed, we allocated a varying number of VMs per core, starting with 1 and scaling up to 20. We deploy 5 of these projects, to utilize a total of about 5 cores. The sample application of k-means is updated with our modules using Visual Studio 2012. We extend some of Daytona's modules to integrate the functionality offered by the proposed PARiNgS framework. The core modules of PARiNgS are Policy Enforcement Point (PEP) and Policy Decision Points (PDP), which are in charge of enforcement and evaluation of privacy policies respectively. The PEP includes data pre-processing module, tainting and evaluation, and is deployed exclusively at the Master, with the exclusion of the

tainting module. Tainting is also deployed at the worker nodes, that may have to re-apply taint upon completion of partial processes. We have prototyped the Cloud information accountability framework on a small Cloud-like environment on Emulab [102] using Eucalyptus [103] as the underlying Cloud middleware.

7.1 Materials and Methods

In order to identify what is the actual information that a broker should account for when performing the service selection, and providing accountability and distributed policy enforcement, we studied the profile of the top ten Cloud service providers [12]. Our analysis included providers offering storage services or the Platform as a Service (Rackspace, Salesforce, Cloud Foundry from VMWare), enterprise Cloud platforms (CloudSwitch from Verizon, IBM Cloud), and service providers who offer multiple types of services (Microsoft Azure, Amazon EC2, and Google Cloud).

We begin with a study of the services provided by major CSPs, and then focus on generating datasets for service selection based on the information gathered.

7.2 A study of Cloud Services

Amazon[21], Google[114], Microsoft[20], Saleforce.com [115] and Sun are considered among the key players in the cloud computing market, but they represent only a small portion of the providers in this space. Other emerging cloud providers are Proofpoint [116], RightScale [117], and Workday [118]. We now summarize the features of some of the most well known providers, highlighting the major differences among them.

Amazon (AWS) offers a number of infrastructure-related web services, including the Elastic Computing Cloud (EC2), Simple Storage Service (S3), CloudFront, SimpleDB and Simple Query Service (SQS). EC2 provides resizable computing capacity in the cloud. It allows scalable deployment of applications by providing web services interfaces through which customers can create virtual machines. S3 is used to store and retrieve unlimited amounts of data at any time from the web. CloudFront is a content delivery network. SimpleDB provides core database functions of data indexing and querying. SQS is a distributed queue messaging service

that supports the programmatic sending of messages via web services applications. AWS can be used for several purposes, in that it supports various operating systems and programming languages. For example, organizations can leverage the AWS worldwide network of edge servers to minimize degradation of delivery and services for content delivery by using cloudfront and S3. Organizations can also leverage AWS as an option for managing internal backup as an alternative to on-site storage infrastructure.

While AWS offers an infrastructure, Google App Engine and Microsoft Azure Services offer platforms as a service for building and hosting web applications on the web infrastructure. They can be used for multiple purposes, such as messaging, securing email systems, collaboration and application development. For example, customers now can develop their applications by using the cloud services without having their own infrastructure installed locally.

Google App Engine offers the platform as a service for building and hosting web applications on the web infrastructure. It can be used for multiple purposes, such as messaging, securing email systems, collaboration and application development. Currently. it supports Python and Java programming languages. Customers can develop their own applications in these two languages by using Google App Engine. Microsoft Azure Services is also a platform, which includes a number of services: .net, SQL and Live services, to be used primarily for application development. Microsoft provides an operating system called Azure, that serves as a runtime for the applications and provides a set of services that allows development, management and hosting of managed applications at Microsoft data centers.

Recently, Sun is promoting an open cloud philosophy, targeting primarily the developer's community. Their Open Cloud Platform plans to offer an infrastructure related to servers, storage and databases. Developers will access the Sun public cloud services from a web browser to provision resources on their platform of choice. With respect to competitors, it will stand out for supporting the various operating systems/programming languages and virtual data center capabilities.

Overall, the cloud is already a viable and sensible solution for many small and medium size business. This is especially the focus area of Amazon, Google, and RightScale or ProofPoint, while Microsoft Azure, Sun and Salesforce are geared toward larger scale enterprises.

7.3 Generation of Testing Datasets

To extract functional and non-functional properties of each provider, we first analyzed the providers' available manifests. These manifests included documents related to security practices, privacy policies, the Cloud documentations on getting started and other user guides, FAQs, white papers, Terms of use, and Service Level Agreements (SLAs).

For example, **Amazon EC2**'s documentation includes: Overview of security Processes (Amazon Virtual Private Cloud Documentation, Amazon Virtual Private Cloud Getting Started Guide, etc.), EC2 SLA, FAQs, Privacy Policy. **Rackspace**'s documentation includes: Privacy Statement, Terms of use, Developer Guides, and Pricing.

The data and the level of detail provided by each Cloud service provider varies widely. For instance, the Amazon EC2 Cloud breaks down its pricing based on various factors including the region to which the service is provided, while the Windows Azure Cloud, despite having an equally detailed documentation uses a different break down for its pricing [20]. Further, the information provided by Salesforce and Rackspace is not easily comparable with the information provided by Amazon or Microsoft. These differences come from the inherently diverse purposes and customer bases for which these services are intended - thus making the service type of the service providers one of the most important differentiating variables among them.

However, as mentioned before, a customer often does not have a clear understanding of the services being offered, and he may simply have a basic idea of the service type he seeks along with the price he is willing to pay. Therefore, the role of middle-man able to synthesize the services features and the relative price units is necessary, so as to lead the customer toward an optimal choice. Precisely, in the case of the provider we analyzed, we identify normalized variables and domain as reported in Table 7.1. We then identified and extracted a set of common properties based on common business recommendations for service selection [19]. A complete set of these properties qualifies a specific service provider and its service offering. Using the information from the manifests, we identified the acceptable values for each of the properties (using ranges), based on the maximum and minimum service

levels offered for a given property by any of the service providers. This gave us our starting set of ten data points and shaped the representation of service providers. With the starting data points, we generated 10K data points representing synthetic providers. Each synthetic providers was generated using random combinations for each of the properties describing it. Specifically, we use a pseudo random number generator to generate a subset of the total possible 10^{10} combinations, and filter out the outliers wherein all the properties have either very low or very high values. The properties were derived also according to common business recommendations for service selection [119]. Table 7.1 provides an excerpt of our data collection analysis. As introduced in Section 4.2, the collected properties include Service Types offered, Security and privacy levels, QoS, Service Measurement Units, Pricing Unit times, Instance Sizes supported by the Cloud, Operating System types supported, the price range, and the regions in which the servers are located. In the table, `Service Type` of type 1 refers to service on-demand, 2 refers to reserved instances, while 3 refers to specialized services such as custom IPs in case of Amazon, or caching in case of Windows Azure. Sec stands for security. `Msrmt` stands for measurement units, where 1 refers to measurement in terms of memory used, 2 refers to measurement in terms of number of transactions, 3 stands for number of connections or data transfers done and 4 for the data transfer time. `Prcg unts` stands for Pricing Units, where 1 stands for per month, 2 per year, 3 per 3 years, and 4 per hour. `IS` denotes Instance sizes where 1 refers to Small and anything below small such as Micro in case of Amazon EC2, 2 for Medium, 3 for Large, while 4 for extra large and above such as Quadruple extra Large provided by Amazon. `OS` is the operating system. A value 1 corresponds to Linux, while 2 is Windows. `Prc` stands for pricing and is normalized to per hour for each SP. `Reg` stands for regions where `Yes` denotes that pricing varies by region, while `No` denotes there is no differential pricing across various regions. With this information, we form a synthetic data set of Cloud service providers, each of which is described by some varying combination of the set of properties described in Table 7.1

Table 7.1. Cloud Service Provider Attributes and Variable Ranges

CSP Name	Variable Names								
	Service Type	Sec	QoS	Msrmt	Prcg unts	IS	OS	Prc	Reg
Amazon EC2	1, 2, 3	High	High	1, 2, 3, 4	1, 2, 3, 4	1, 2, 3, 4	1, 2	0.000 - 2.60	Yes
Windows Azure	1, 2, 3	High	High	1, 2, 3, 4	1, 4	1, 2, 3, 4	2	0.04- 0.96	No
Rackspace	1, 2, 3	Low	Medium	1, 3, 4	1, 4	2	1, 2	0.015 - 1.08	No
Salesforce	1, 2, 3	Low	Medium	4	1	N/D	N/D	2 - 260	No
Joynet	1, 2, 3	Low	Medium	4	1, 4	3, 4	1, 2	0.085 -2.80	No
Google Clouds	1, 2, 3	Medium	High	1, 2	1, 4		N/D	0.0057 - 0.0068	No

7.4 Experimental Results related to Cloud Service Selection

We compare our cloud service selection (CSS) algorithm introduced in Chapter 4 with a baseline approach which uses an exhaustive search to check all possible combinations of all service providers for a given query and find the service providers that match the query properties best. The performance is evaluated in terms of both efficiency and accuracy. Efficiency is measured using the processing time. Accuracy is measured as the number of different properties in the service providers returned by our solution and that by the baseline solution. We first present the results of the B$^+$-tree, followed by the results of the tests for the B^{cloud}-tree.

7.4.1 Effect of Number of Service Providers on the B$^+$-tree

In the first set of experiments, we compare the performance of our CSS algorithm with the baseline approach when the total number of service providers is increased from 1000 to 10000. For each set of service providers, we execute 100 service selection queries that contain 9 number of desired properties. Figure 7.1 shows the average service selection time. Observe that our CSS algorithm significantly outperforms the baseline approach, and the performance gap between the two approaches is enlarged quickly with the increase of the number of service providers. Specifically, our CSS algorithm is about 100 times faster than the baseline approach when there are 10000 service providers. This demonstrates the pruning power of the CSP-index used by our approach. The CSP-index arranges service providers according to the similarity among their properties. Given a service request, the CSP-index helps to quickly direct the search to the group of service providers that may satisfy the querying properties. The baseline approach is very time consuming since it needs to check the property of each service provider and verify all possible

combinations of service providers.

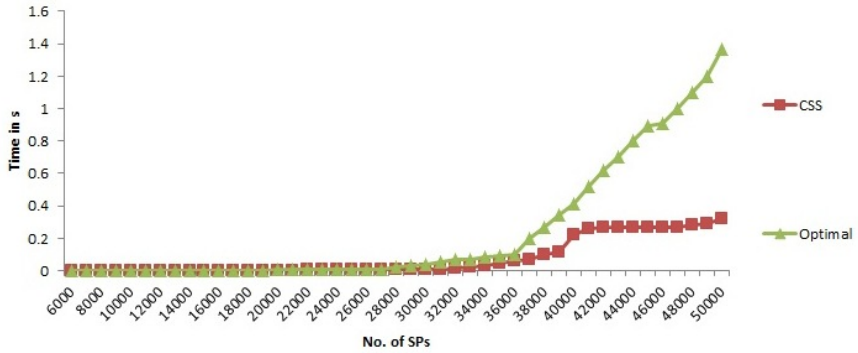

Figure 7.1. Variation of processing time with number of SPs

Next, we compare the results returned from our approach with that from the baseline approach to evaluate our query accuracy. Figure 7.2 reports the number of different properties in the service providers obtained from our approach and the baseline approach. It also shows how a purely greedy algorithm fares with respect to the baseline. As shown, our CSS algorithm usually has just one or two properties that are different from the baseline approach. This minor difference is due to the greedy algorithm adopted by our approach after the initial selection of CSPs. Considering the overall performance in terms of efficiency and accuracy, our approach is considerably better than the baseline approach.

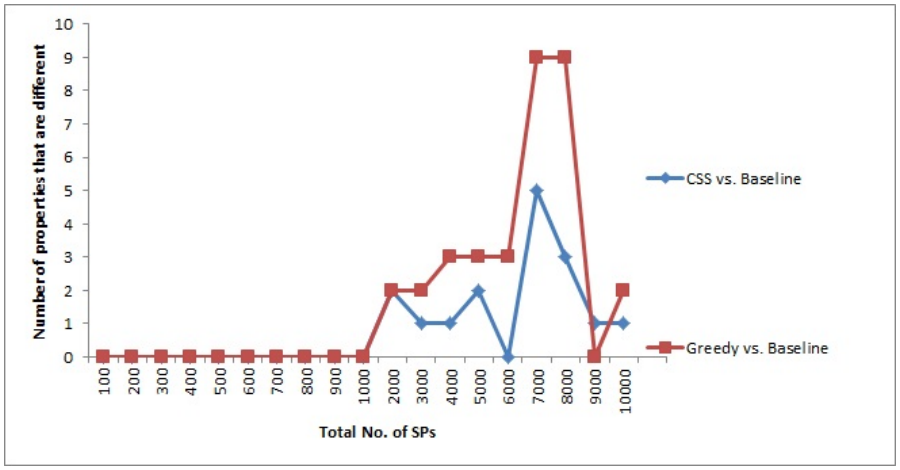

Figure 7.2. Accuracy of the CSS algorithm

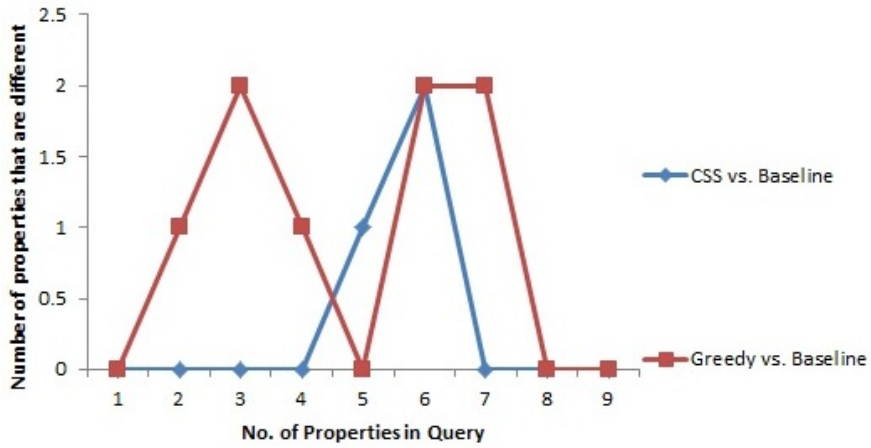

Figure 7.3. Effect of the number of querying properties for queries returning a single SP

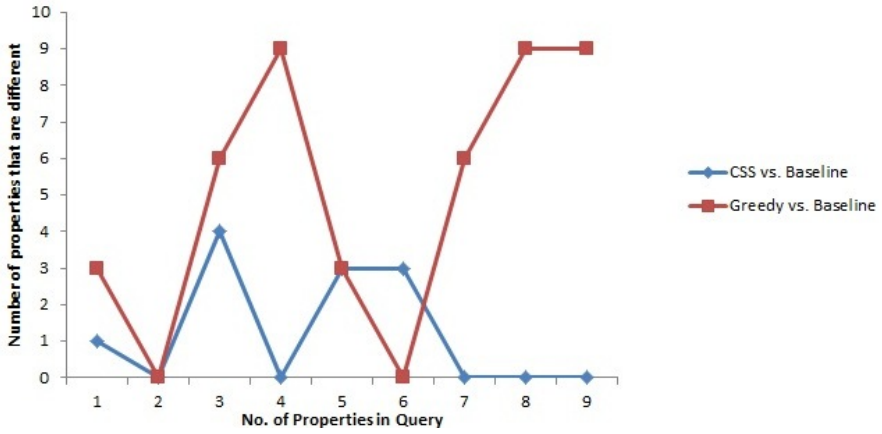

Figure 7.4. Effect of the number of querying properties for queries returning multiple SPs

7.4.2 Effect of Number of Properties Required in the Service Selection on the B$^+$-tree

To evaluate the effect of the properties, we vary the number of querying properties per request from 1 to 9 and test them in the dataset containing 5000 service providers. Two types queries are considered. One denoted as in Figure 7.3 refers to queries that return more than one service provider for a single request. The other denoted as "single" refers to queries that can be satisfied by a single service provider and these results are shown in Figure 7.4. The first observation is that

the results returned by our CSS algorithm have only minor differences from that of the baseline approach in most cases. As can be seen in both figures, it clearly outperforms a simple greedy algorithm. Second, we notice that the CSS algorithm yields better accuracy, i.e., fewer number of different properties, when the query result contains just single service provider. This is because it is easy to verify whether a single service provider matches the query requirement. There are more possibilities when selecting the combination of service providers when the service request needs to be fulfilled by multiple service providers.

7.4.3 Experiments on the B^{cloud}-tree

We calculate the time taken by the B^{cloud}-tree to find a comparable match to the query both when the query is expressed in terms of fixed values, and in terms of a range of desired values. To check the performance of the B^{cloud}-tree in terms of time taken, we increase the number of CSPs being checked from 100 to 20000 and measure the system time in milliseconds. This is compared with the optimal approach for time, which is the greedy approach.

We then measure the accuracy of the B^{cloud}-tree for both fixed value queries and interval queries by comparing the number of properties in which the CSP returned by the B^{cloud}-tree differed from an exhaustive match of the queries. We find that the accuracy of the B^{cloud}-tree is very high for the interval queries.

7.4.3.1 Performance of Exact Queries on the B^{cloud}-tree

First, we evaluate the performance of exact queries in terms of varying number of CSPs and the number of querying properties.

• **Effect of Number of CSPs on the B^{cloud}-tree**:

In this round of experiments, we generate 100 exact queries, each of which includes 5 randomly chosen query properties and values. We execute the 100 queries against different number of CSPs ranging from 1000 to 10000, and record the average processing time and accuracy.

Figure 7.5 shows the query processing time of the two approaches: the B^{cloud}-tree, and the B^{+}-tree.

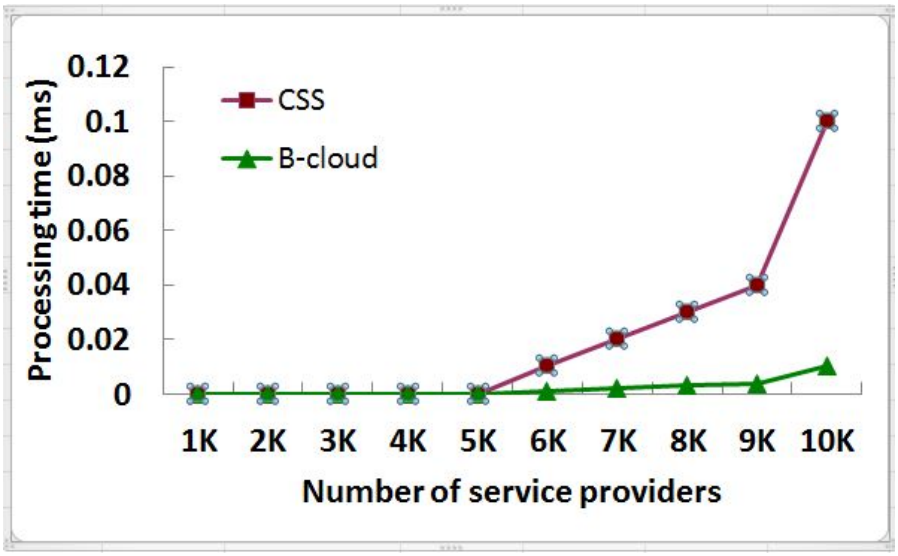

Figure 7.5. Performance of the B^{cloud}-tree, and the B^{+}-tree

Figure 7.6 shows the accuracy of the query results obtained from the two approaches as the number of service providers are increased

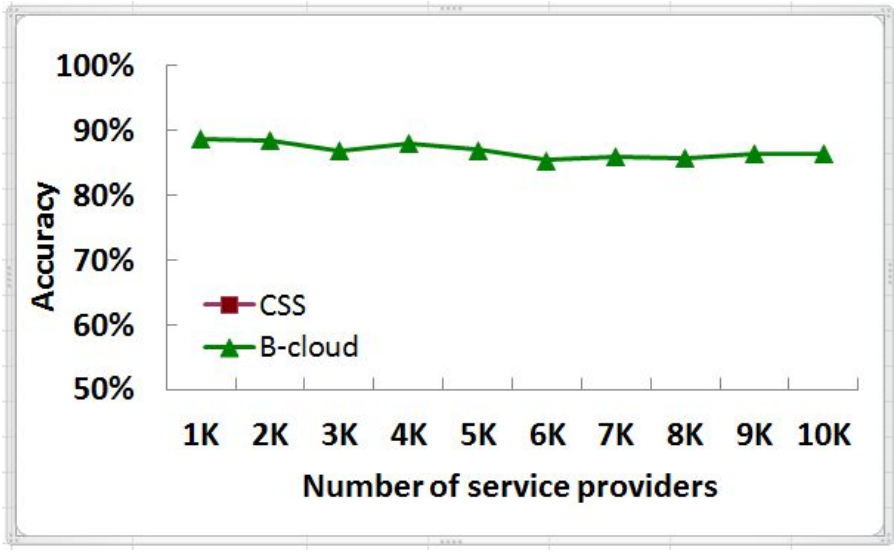

Figure 7.6. Accuracy Comparison of the B^{cloud}-tree, and the B^{+}-tree

• **Effect of Number of Querying Properties on the B^{cloud}-tree**:

In this set of experiments, we vary the number of querying properties per request from 1 to 10 and test them in the dataset containing 5000 service providers. We report the average cost of 100 queries with the same number of querying

properties for each setting.

Figure 7.7 shows the number of properties in the query results that do not satisfy query requirements.

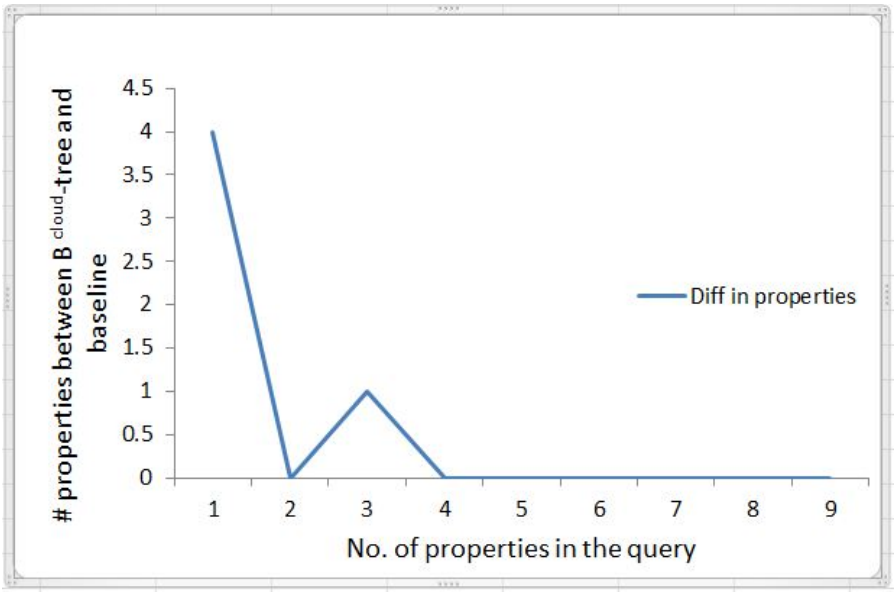

Figure 7.7. Number of properties in the fixed query results that do not match results of the baseline in the B^{cloud}-tree

Figure 7.8 compares the average query processing time as the number of service providers are increased up to 20000.

7.4.3.2 Performance of Interval Queries

We now evaluate the performance of the interval queries which allow the users to specify a range of desired values for querying properties. We also examine the effect regarding the varying number of CSPs and the number of querying properties. Since the B^+-tree does not support interval queries, in what follows, we present the comparison results between the B^{cloud}-tree and the baseline approach.

• **Effect of the Number of CSPs**:

For this round of experiments, we generate 100 interval queries with the following characteristics: (1) each query contains 5 randomly selected properties; (2) each query property is associated with a randomly generated interval with the medium size. The number of CSPs is increased from 1000 to 20000 and the time taken for the results to be returned is recorded. The results are shown in Figure7.9

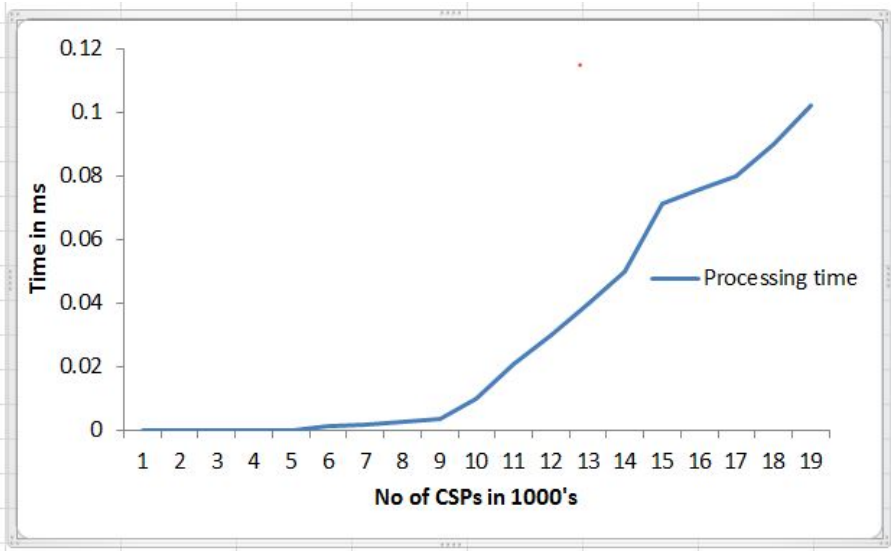

Figure 7.8. Average processing time for fixed queries in the B^{cloud}-tree

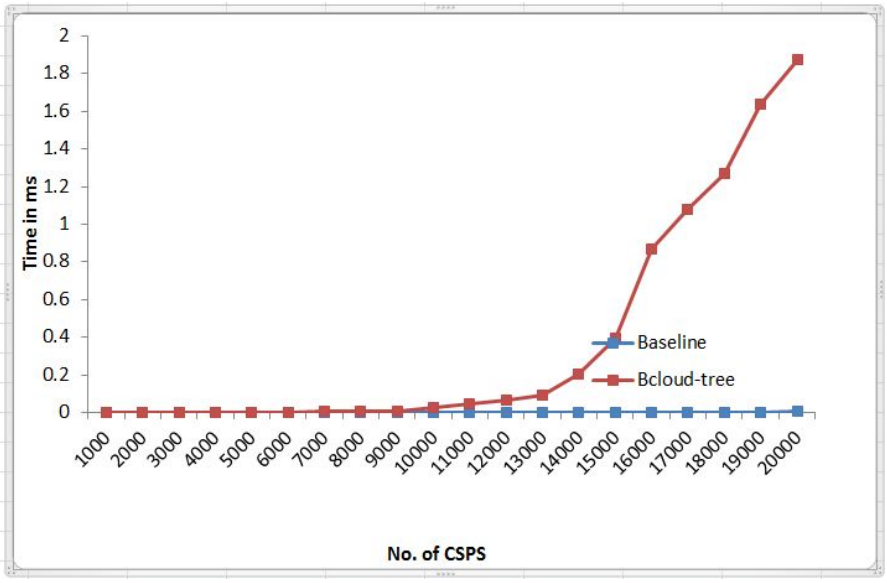

Figure 7.9. Average processing time for interval queries in the B^{cloud}-tree

• **Effect of the Number of Querying Properties and Their Ranges**:

We generate interval queries with the number of query properties varying from 1 to 10. For each query property, we randomly generate a medium size query interval. We generate 10 sets of 100 queries. For each set of queries, they have the same number of query properties while the specific properties in each query is randomly chosen from the 10 properties. We then measure the number of differences in the

results returned by the B^{cloud}-tree as compared to the baseline. The results are shown in Figure 7.10

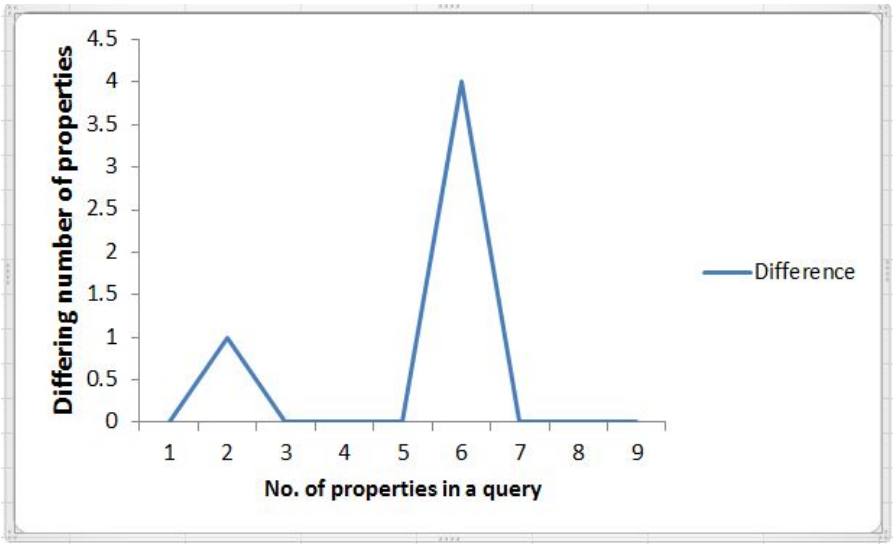

Figure 7.10. Number of properties in the interval query results that do not match results of the baseline in the B^{cloud}-tree

7.5 Experimental Evaluation of GABE's Node Selection Framework

We evaluate the scalability and overheard introduced by GABE's Node Selection framework, with respect to the data partitioning, data tainting, distributed policy evaluation tasks, as well as the overall time for completing a job. All experiments were conducted in public settings, using the k-means algorithm.

The first set of experiments aims to measure the time taken for data partitioning based on user's policies. We used a policy with two rules. The first rule has a *Target* component specified on the indexes of the data (since the k-means data is tabular in form). The second rule has the *Target* consisting of some conditions specified on columns of the table. In the experiment we repeated multiple tests, varying the file size and number of conditions. The results are reported in Table 7.2. In the first case, the pre-processing takes place in constant time, because the partitioner just needs to partition the data at the indexes specified (assuming

the bucket partitioning algorithm does not shuffle around content). It averages at 0.03765 seconds for 30k file size. In the second case, wherein the $Target$ of the rule is expressed as a boolean function against the data attributes, the pre-processing on conditions takes linear time. The input file size also affects pre-processing time as the number of buckets the data is divided into depends on file size.

File Size	1 cond	2 cond	3 cond	4 cond	5 cond	6 cond	7 cond	8 cond	9 cond	10 cond
10k	0.1754	0.35	0.52618	0.7001	0.8787	0.99976	1.2278	1.4082	1.5768	1.754
20k	0.1761	0.35187	0.527976	0.70397	0.8799	1.0543	1.2327	1.40876	1.58478	1.7598
30k	0.1768	0.35358	0.529991	0.70735	0.8814	1.0608	1.2376	1.4099	1.6001	1.7652
40k	0.1773	0.3546	0.5331	0.70987	0.886	1.0637	1.2411	1.4208	1.6014	1.7698
50k	0.1779	0.356	0.53365	0.7116	0.8888	1.0676	1.2435	1.4256	1.621	1.7891

Table 7.2. Time taken for pre-processing in seconds

The second set of experiments evaluate the overhead introduced by tainting. Tainting is executed by the Master before the map phase, and, after the map phase is completed when the mappers have to apply the taint again. Since the Master and workers have similar configurations, the measurement for both tasks can be done at any node. We simply chose a worker node at random for these measurements. As shown in Figure 7.11, the time taken for tainting increases linearly with the number of data partitions, assuming that each partition represents a single target and therefore has one applicable rule. We increased the number of partitions from 5 to 50 in increments of 5. The minimum time taken for tainting is 0.7 seconds, when we have 5 partitions, while at 50 partitions, it takes about 1.7 seconds.

Next, we evaluate the efficiency of the distributed policy evaluation protocol introduced in Chapter 5. The CPU time is measured for all the operations starting with property extraction and ending with the key formation for data access. Up to 7 properties were verified for the workers. We carefully selected the ratio of workers versus verifiers, per our discussion in Section 5.4.2. Up to 36 nodes act as verifiers, for 99 workers[1]; when a lower number of workers are used for a job, the number of verifiers is accordingly reduced to maintain a ratio of workers to verifiers as about 2.75:1 in order to ensure that the probability of receiving a key share from an uncorrupted verifier is higher than 0.5. The results of this experiment, for up to 100 workers, are shown in Figures 7.11, and 7.12. As can be seen, the time taken for the keys to be obtained by all the workers shows a nearly linear time increase with the increase in the number of nodes. Finally, our last set of experiments

[1]A verifier for a given job may itself be being verified for one or all of the properties.

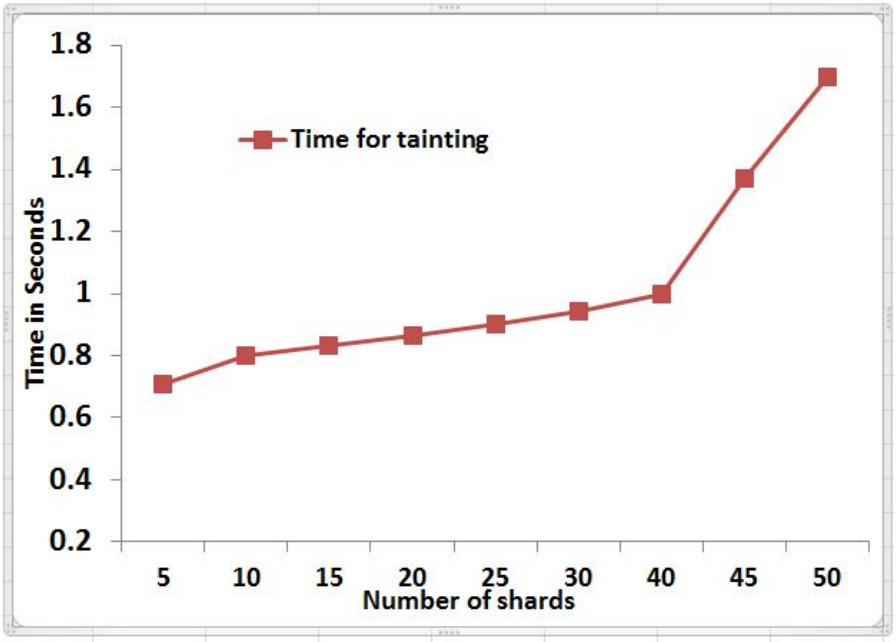

Figure 7.11. Processing time for tainting varying number of shards

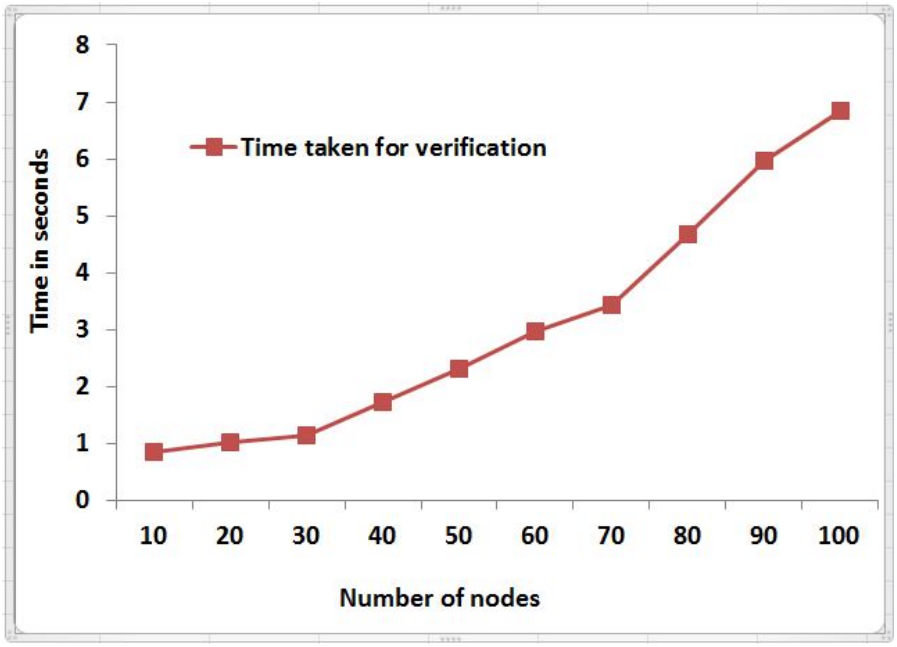

Figure 7.12. Processing time for distributed verification

measures the overall performance of our PARiNgS framework which includes all steps, from rule writing to job completion. We measure the time taken to complete running the k-means algorithm for a data set with 1000 data points. We compare our framework with a centralized baseline approach. In the centralized evaluation approach, only the Master is in charge of verifying the workers' properties. As baseline, the runtime for the k-means algorithm averages at around 40 seconds for one iteration, and 650 seconds for 10 iterations. As for our approach, we used 100 VMs with one of them being the Master. We assume the probability of a node being corrupted is 0.1. For a trustworthy verification, P_S, the probability of getting shares from non-corrupted nodes, needs to be higher than 0.5. The number of verifiers were calculated so that $P_S \geq 0.6$. The experiment is run for both a mono-iterative and multi-iterative k-means. We find that the runtime using our distributed policy evaluation protocol shows a linear increase. The overhead in case of distributed evaluation is reported in Table 7.3. As reported, simple policies on non iterative k-mean processes incur a very small overhead increase (less than 5%). In contrast, the execution time in case of centralized processing is extremely high, and of orders of magnitude higher than the distributed case. We report the results of the comparison between iterative and centralized execution in Figure 7.13. As shown, for computationally expensive tasks (such as iterative k-means) the time explodes when multiple rules are being verified.

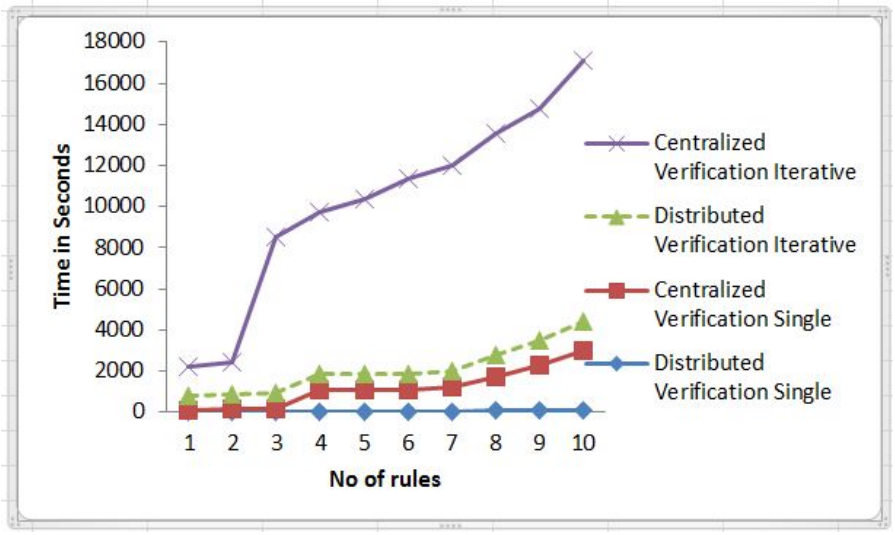

Figure 7.13. Processing time for complete task execution on two separate settings

No. of rules	Time(sec)1 round	Time(sec) 10 rounds	No. of cond	No. of Verifiers
1	45	705	1	18
2	50	750	1	18
3	51	760	2	36
4	52	770	2	36
5	53	780	2	36
6	53.5	785	3	40
7	56	790	3	42
8	75.86	1008.6	4	45
9	95.764	1207.6	5	45
10	115.584	1411.1	7	45

Table 7.3. Runtime for Single and Multiple Iterations of K-means

By taking a further look of the above experiment, for both approaches, the properties being verified include: nodes' capabilities, location, files access, support for cryptographic protocols (i.e AES, DES, 3DES). Verifying some of such properties, as discussed in Appendix, may involve several steps and is therefore time consuming. For instance, the base-time for location verification by a sub-verifier is approximately 65 ms for echoing 32 bytes of data, when the sub-verifier is on the same continent, but a different zone from the worker (e.g. the sub-verifier is in US East, while worker is in US West). It is about a max. of 251 ms when the sub-verifier is in a different continent (the sub-verifier was in US, while the server being pinged was in India)[2]. The average verification time by the verifier is 0.0651 seconds. For nodes capabilities, an average of 0.00435 seconds is taken when the sample job being submitted is k-means. For each of the security properties, the average verification time is 0.06324 seconds.

7.6 Experimental Results related to Cloud Information Accountability

In the experiments conducted on the CIA framework presented in Chapter 6, we first examine the time taken to create a log file and then measure the overhead in

[2]This is almost never a realistic setup, and was only tested as an extreme outlier case. Since the affinity group of all our workers is the same, we had to ping a different server existing in India.

the system. With respect to time, the overhead can occur at three points: during the authentication, during encryption of a log record, and during the merging of the logs. Also, with respect to storage overhead, we notice that our architecture is very lightweight, in that the only data to be stored is given by the actual files and the associated logs. Further, JAR act as a compressor of the files that it handles. In particular, as introduced in Section III, multiple files can be handled by the same logger component. To this extent, we investigate whether a single logger component, used to handle more than one file, results in storage overhead.

7.6.1 Log Creation Time

In the first round of experiments, we are interested in finding out the time taken to create a log file when there are entities continuously accessing the data, causing continuous logging. Results are shown in Figure 7.14. It is not surprising to see that the time to create a log file increases linearly with the size of the log file. Specifically, the time to create a 100Kb file is about 114.5ms while the time to create a 1 MB file averages at 731ms. With this experiment as the baseline, one can decide the amount of time to be specified between dumps, keeping other variables like space constraints or network traffic in mind.

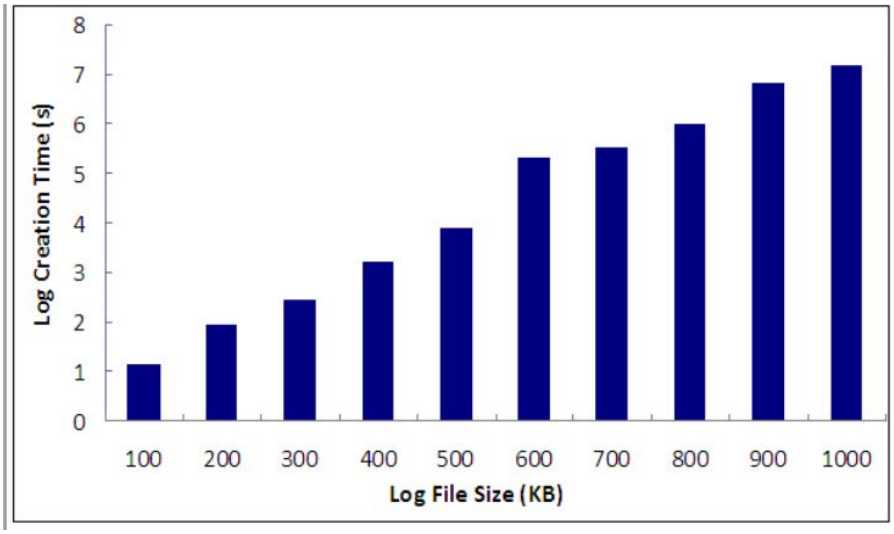

Figure 7.14. Time to Create Log Files of Different Sizes

7.6.2 Authentication Time

The next point that the overhead can occur is during the authentication of a CSP. If the time taken for this authentication is too long, it may become a bottleneck for accessing the enclosed data. To evaluate this, the head node issued OpenSSL certificates for the computing nodes and we measured the total time for the OpenSSL authentication to be completed and the certificate revocation to be checked. Considering one access at the time, we find that the authentication time averages around 920 ms which proves that not too much overhead is added during this phase. As of present, the authentication takes place each time the CSP needs to access the data. The performance can be further improved by caching the certificates.

The time for authenticating an end user is about the same when we consider only the actions required by the JAR, viz. obtaining a SAML certificate and then evaluating it. This is because both the OpenSSL and the SAML certificates are handled in a similar fashion by the JAR. When we consider the user actions, such as submitting his username to the JAR, it averages at 1.2 minutes.

7.6.3 Time Taken to Perform Logging

This set of experiments studies the effect of log file size on the logging performance. We measure the average time taken to grant an access plus the time to write the corresponding log record. The time for granting any access to the data items in a JAR file includes the time to evaluate and enforce the applicable policies and to locate the requested data items.

In the experiment, we let multiple servers continuously access the same data JAR file for a minute and recorded the number of log records generated. Each access is just a view request and hence the time for executing the action is negligible. As a result, the average time to log an action is about 10 seconds, which includes the time taken by a user to double click the JAR or by a server to run the script to open the JAR. We also measured the log encryption time which is about 300ms (per record) and is relatively constant when the log file size increases.

7.6.4 Log Merging Time

To check if the log harmonizer can be a bottleneck, we measure the amount of time required to merge log files. In this experiment, we ensured that each of the log files had 10% to 25% of the records in common with one other. The exact number of records in common was random for each repetition of the experiment. The time was averaged over 10 repetitions. We tested the time to merge up to 70 log files of 100KB, 300KB, 500KB,700KB, 900KB, and 1 MB each. The results are shown in Figure 7.15. We can observe that the time increases almost linearly to the number of files and size of files, with the least time being taken for merging two 100KB log files at 59 ms, while the time to merge 70 1 MB files was 2.35 minutes.

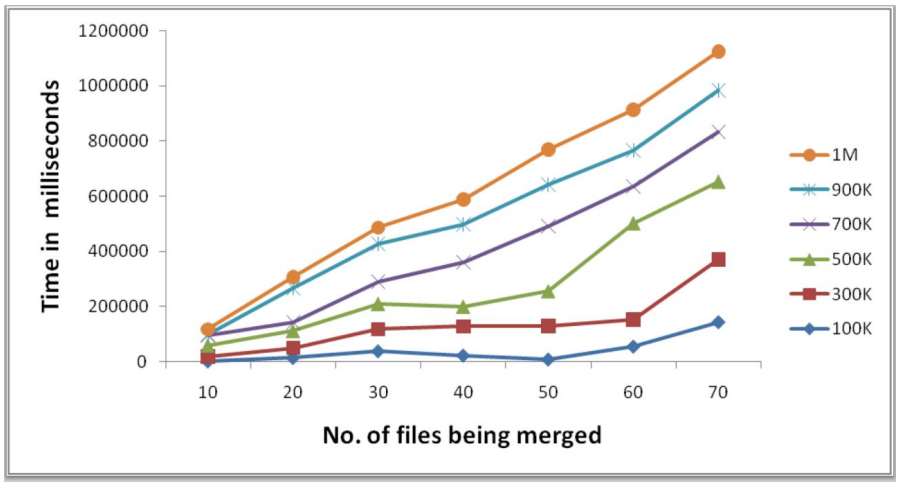

Figure 7.15. Time to Merge Log Files
in a Cloud Brokerage System

7.6.5 Effect of Size of the Data on Size of the JAR Files

Finally, we investigate whether a single logger, used to handle more than one file, results in storage overhead. The results of this experiment are reported in Figure 7.16. We measure the size of the loggers (JARs) by varying the number and size of data items held by them. We tested the increase in size of the logger containing 10 content files (i.e. images) of the same size as the file size increases. Intuitively, in case of larger size of data items held by a logger, the overall logger also increases in size. The size of logger grows from 3500KB to 4035KB when the size of content

items changes from 200KB to 1MB. Overall, due to the compression provided by JAR files, the size of the logger is dictated by the size of the largest files it contains. Notice that we purposely did not include large log files (less than 5KB), so as to focus on the overhead added by having multiple content files in a single JAR.

7.6.6 Overhead added by JVM Integrity Checking

We investigate the overhead added by both the JRE installation/repair process, and by the time take for computation of hash codes.

The time taken for JRE installation/repair averages around 6500 ms. This time was measured by taking the system time stamp at the beginning and end of the installation/repair.

To calculate the time overhead added by the hash codes, we simply measure the time taken for each hash calculation. This time is found to average around 9 ms. The number of hash commands varies based on the size of the code in the code does not change with the content, the number of hash commands remain constant.

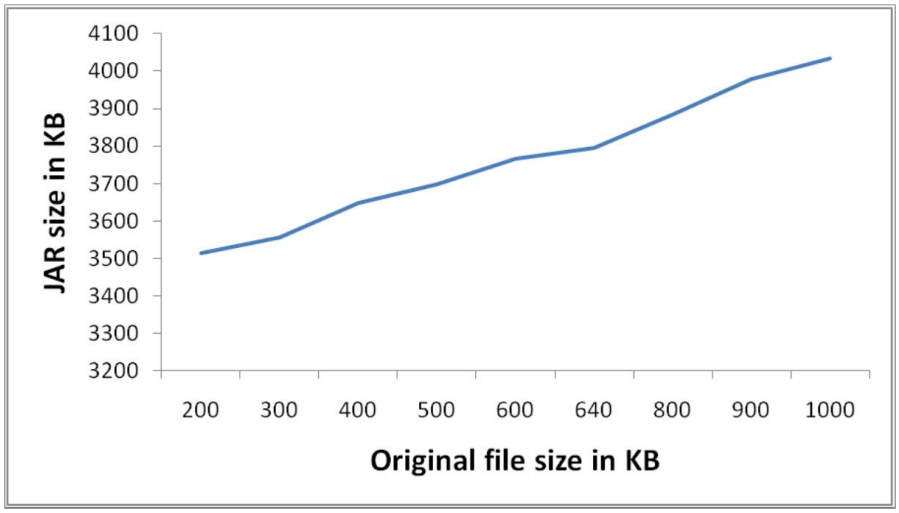

Figure 7.16. Size of the Logger Component
in a Cloud Brokerage System

Chapter 8

Concluding Remarks and Future Work

8.1 Conclusions

Cloud computing is beginning to play a large role in small and medium businesses. Cloud computing systems have many parallels to distributed systems, but they present a new business model, with which comes a new threat model. Due to the similarities between Cloud computing and other distributed systems, the solutions developed for Cloud computing environments pan well for other distributed environments. For instance, like web services Clouds also present users with different services hosted on a server. They allow users to host applications on them too. However, the service models developed for other distributed environments, such as web services do not necessarily pan out for the Clouds. For instance, web services have a widely accepted standard for representing their SLAs, while not all Cloud service providers follow the same standard for their SLAs. Further, in other distributed systems, the users are faced with semi-trusted service providers; however, in Clouds, the service providers are fully untrusted as they can be compromised. As presented in this dissertation, the Cloud brokers form a first step towards bridging the gaps in the trust as they help end users select and rank Cloud service providers.

We designed and presented a detailed framework for the Cloud broker architec-

ture. We also developed detailed algorithms to realize the proposed architecture. In particular, we presented two novel indexing structures, namely the B^+-tree and the B^{cloud}-tree, to facilitate the arrangement and retrieval of the information about service providers. We also proposed innovative approaches for automatically logging any access to the data in the cloud together with an auditing mechanism. Our approach allows the data owner to not only audit his content but also enforce strong back-end protection if needed. Moreover, one of the main features of our work is that it enables the data owner to audit even those copies of its data that were made without his knowledge. Further, we demonstrated that the back-end protection used by the CIA framework, can be easily extended for email attachments. In fact, the back-end framework presented forms a light-weight, portable and flexible architecture for content protection.

8.2 Limitations and Ongoing Work

Given that GABE forms a central trusted piece of each of the tasks being carried out, verifying GABE's actions are carried out correctly is necessary. The meaning of the term correctness varies with respect to the function being carried out by GABE. We are currently taking steps to relax some of the assumptions on the trust placed on GABE.

8.2.1 JRE Security Limitations

An inherent limitation of our brokerage architecture lies in the reliance of GABE's accountability framework on Java. The integrity of GABE's accountability framework is directly dependent on the integrity of the JRE of the remote systems the data resides on. This is a non-trivial problem faced by any system that is located on a remote untrusted machine. Accordingly, we plan to refine our approach to verify the integrity of the JRE. For example, we will investigate whether it is possible to leverage the notion of a secure JVM [120] being developed by IBM. This research is aimed at providing software tamper resistance to Java applications. The JAR files will check the JVM before enforcing the policies as a part of our effort to achieve software tamper resistance.

8.2.2 Ongoing Work on the CSS framework

As presented in Chapter 4, GABE's service selection architecture fulfills the basic requirements of helping a user select a CSP. It ranks CSPs, and helps users identify possible collisions and collusions between them. However, in doing so we assume that the broker indeed searches the entire list of CSPs stored in either tree structure. In order to relax this assumption and offer proof to the client on the correctness of the results returned, we are currently working on extending the B^+-tree using a Merkle B^+-tree *MBT*. Specifically, the is used as an authenticated structure for indexing the re-defined numeric property Pricing. Specifically, in the *MBT*, the leaf nodes store the actual prices of CSPs and the values of internal nodes are computed from the concatenation of the hash tags of their child nodes via an appropriate hash function. This type of structure allows clients to present verifiable range query over the Pricing property. Note that the reason for indexing Pricing in this scheme is twofold: on the one side, price is one of the most common and important criterion used by clients to specify clients query requirements; on the other side, it has many possible values in domain, leading to a better dispersion performance in B+ tree and facilitating the search over this property. We note that this verifiable query over single dimensional space is not enough for CSP selection in that CSPs are typically associated with multiple properties and a clients service selection query usually involves at least a subset of them. Accordingly, the *MBT* will be extended to support indexing over multiple properties to better support multi-dimensional queries.

8.2.3 Ongoing Work on Ensuring Reachability of Virtual Machines

Another non-trivial problem faced in the Cloud computing architecture is the reachability and integrity of virtual machines (VMs). Virtualization allows Cloud providers to maximize the use of their hardware resources by allowing them to multiplex many customer VMs across a single physical machine. Data leakage problems can be introduced due to co-resident VMs as shown in the paper [13]. Accordingly, we are currently investigating the detection of data leakage via covert side channels by relying on exceptions and events occurring within each VM. we

focus on load-based measurement attacks, which are covert side-channel attacks born specifically as a result of the virtualization in Cloud systems [13]. In general, a covert channel attack is an attack which takes place when two entities or processes communicate with each other via channels that are hidden and therefore not subject to the general access control techniques. These channels can be formed by relying on time-based operations [121], such as opening and closing a file at a certain time, or can rely on techniques such as port knocking [122]. our solution consists of few main steps: (1) collection of system calls occurring at the physical host, and the exceptions which may be specific to the attacks, to (a) identify the VM causing the exception, and the process that spawned the exception, and (b) identify whether conditions sufficient for the attack exist, and (2) processing of these exceptions to detect the load-variation attacks. Each of the steps is associated with a logic module, which we refer to as **Observer** and **Defender**, in what follows. The **Observer** and **Defender** are implemented as part of a trusted VM. he Observer component is designed to dynamically collect metrics indicative of suspicious load variations. We specifically focus on tracking network processes, their CPU loads and spawned system calls. Hence, the Observer has three main tasks: (1) extract systems calls and interrupts of monitored processes, (2) map the identified system calls and interrupts to specific programs from the specific VMs which generated them, and (3) determine whether the conditions necessary for an attack to be carried out co-exist. In order to extract and map the system calls, the Observer spawns multiple tracing threads, instantiated by means of debugging tools, such as Linux strace and WinDBg in Windows. To quantify the necessary conditions for an attack, the Observer uses some baseline system metrics on CPU utilization by processes observed in absence of attacks, as well as the expected number of system calls experienced by any given process, as a part of a training process. Upon gathering sufficient training data on all possible network processes triggered by the VMs, the Observer labels as suspicious each process PID if (a) the CPU activity is above a given threshold τ, and (b) PID is a network process. Specifically, with respect to (b), upon crossing the τ CPU threshold activity, the Observer checks the event logs which are downloaded from the monitored VM to the trusted VM hosting the Observer and Defender, to identify if a particular external host or a group of hosts has been trying to ping or otherwise activate

the process. If the increase in activity is indeed caused by external systems, the Observer alerts the Defender to check for possible attack patterns. The Observer determines the minimum conditions for the system to be in an anomalous condition. However, it does not provide any indications on the actual methodology taken to affect load variations. Therefore, once it receives the IDs of the VMs, the corresponding suspected processes and the exceptions from the Observer, the Defender starts searching for attack-specific patterns. It specifically starts monitoring for patterns if the network processes reach a high load due to network events, per the information obtained from the Observer. Each pattern consists of a particular sequence of exceptions, wherein both the type of exceptions observed, their order, as well as the frequency of particular system calls within the sequence matter. Of course, system calls may be suppressed by a sophisticated attacker at the originating VM.

However this work is currently limited only to one attack and also suffers from false positives and negatives. It will be further refined to deal with false positives and negatives arising from the rate of exceptions occurring, and then extended to address other attacks.

8.3 Future Work

Apart from the completion of the ongoing work, we foresee many additional extensions and uses of the GABE framework to enable it to reach its full potential.

GABE currently has the ability to process a user's query, but the query being presented to GABE has to be well formulated for GABE to make a correct choice for the user. This in turn assumes that a user has a complete understanding of the exact services he requires from the Cloud. However, this is often not the case. Users often only have the knowledge of the task they are looking to outsource to the Cloud. Accordingly, an interesting direction for GABE to take would be interpreting the user's needs from a raw query and formulating the actual query that would be beneficial to the user. Since GABE is responsible for sending the user's data to the CSPs, GABE can also set up the actual Cloud service required by selecting the number of VMs required, the firewall groups and any access control policies as required. Further, the logger components of GABE can be extended to

log the verification actions done during the node selection and setup to prove to the user that his data is indeed not being misused. The logger components can also be integrated with various business intelligence and statistical tools to provide business intelligence reports and incident detection reports to the appropriate personnel. These are just some of the possible extensions of GABE. We foresee GABE becoming a powerful, integrated tool for use in multiple distributed systems once its full potential is realized.

Appendix A

Application Scenario for the Cloud Information Architecture beyond Cloud Computing

The CIA framework is designed for Cloud computing platforms, but it is applicable to other application domains, one of them being emails. In this Section, we discuss the basis of the policy enforcement architecture, referred to as JUiCE, and present how it binds with the data and how the policies are enforced. We then present how the distributed policy enforcement system can be adapted to protect email attachments. We begin with the formulation of a problem statement for distributed policy enforcement, and then present our detailed framework

A.1 Problem Statement

The focus of this work is to provide an extended form of content protection for images, and more in general, multimedia files. A document is an abstraction of a data item to be protected, such as a text file, a database tuple or an image. We aim at providing a way for content owners to achieve extended control, according to which selected recipients can access, and use, the content they are entitled for, within a certain time interval. Our primary challenge is to provide a platform-independent and lightweight approach that does not require users to deploy dedicated software or to be constantly connected online to obtain access. To this end, we aim to

develop techniques which satisfy the following requirements.
1. The content should be protected according to the sender's specified protection requirements. Content protection should go beyond the traditional access control mechanisms that have no control over the file after the access rights are granted to the recipients.
2. Content owners should have a certain degree of control over the actions that the recipients are allowed on the data. For example, if a user does not wish to the recipients to be able to share or otherwise distribute his image, the proposed technique should prevent the recipient from sharing it with other unauthorized recipients.
3. Content owners may have different levels of privacy concerns over their content. For example, the user may just want to obtain the highest level of protection of her content for less-trusted users. Hence, the proposed technique should be able to accommodate to a variety of data protection needs.
4. The content protection mechanism should not intrusively monitor the recipient's or the sender's system, nor should it introduce too much communication and computation overhead.

The primary threat we aim to guard against is **content misuse or distribution**, which occurs when malicious users forge content control protection mechanisms to handle the data differently from the way originally intended by the owner. Our threat model assumes that the actors involved in the content exchange run uncompr-omised Java Running Environment (JRE), and that content recipients do not share their authentication credentials. We assume that the sender does not purposely send malicious content.

A.2 Content Protection Policies

Each CPP is a collection of rules $R_1, \ldots, R_n$, authored by one content owner and applies to a specific *protection object*. While the CPPs infrastructure can be applied to any object type, in this paper we consider their application to all formats of media files (i.e., images or videos). Therefore, the object can be any of these file types and O is simply the ID of the object, such as the name of the file. Each rule has a simple structure, that we represent by means of a tuple $R_i = (S, P, I)$

where $i \in [1, n]$. S denotes the set of users the rule refers to, P is the privilege granted, and I denotes the temporal constraint. I is an optional component, and in case it's omitted the rule has no temporal limit. We support *Full*, *Partial* (or Upload-Only), and *Minimal* privileges. As far as time privileges are concerned, I includes the time interval and the action, denoted as *(δ, Act)*. Three actions are possible upon expiration of the interval: the content can be released completely, made unaccessible, or it can be released with the privileges specified in P. Further, temporal constraints can apply starting from the time where 1) the policy is created and attached to the document, or 2) the file is opened and the policy enforced. δ is a possibly endless time interval, limited by two temporal boundaries, t_b, which indicates the time when the temporal constraint becomes effective and t_e, which indicates the time when the temporal constraint stops being effective. t_b can be instantiated at the time of policy creation - for temporal constraints that apply from the moment the policy is created, or it can be instantiated at the time the policy is first executed. *Act* can take on three possible values: *open*, *revoke* or *release*. In case of *revoke*, at t_e the file along with the policy expires, and the content is no longer accessible. On the contrary, *open* corresponds to the disclosure of the protected content, upon expiration of the time interval according to the rule's privilege. *Release* applies in case of a *Partial* or *Minimal* privilege. At t_e, in case of *release* the content is fully disclosed.

A.2.1 Content Binding Mechanism

To closely bind the user's content with the corresponding CPP for the purpose of strong enforcement we use two types of nested JARs, a super-JAR, and some sub-JARs. The super-JAR is in charge of managing the recipient authentication. The CPP is stored at the super-JAR, and is encoded as a java.policy file. The CPP's rules' subjects are expressed using randomized identifiers. Precisely, upon creating a CPP, a seemingly randomized id is created per recipient by computing $rid_i = h(id_i|s_i)$ where h denotes a cryptographic hash function, id_i is the user's original identifier (e.g., username) concatenated with a random salt s_i selected by the sender ($i \in [1, n]$, where n denotes the number of actual recipients). Therefore,

the recipient's original username, say alice12 is actually never used in the JAR. If alice12 username is randomized to xyz689, the CPP will have a rule saying that a user with userid xyz689 has the right to access sub-JAR10. Each username is combined with a different salt value, for obvious security reasons.
Each rule in the CPP contains a reference to a JAR file, which is nested in the super-JAR. The super-JAR in fact encases m ($m > n$) JAR files (sub-JARs), a subset (n) of which contains *actual* data, while the remaining m-n contain spurious data, and are referred to as *spurious sub-JARs*. Each sub-JAR $SubJ_i$ translates the access control rights for a given user and contains the actual content being distributed in an encrypted form. In the case of spurious sub-JARs, the usernames are fictitious, and enforce access rights to empty files. As explained later in the section, spurious sub-JARs are added for confidentiality and security reasons. When an actual sub-JAR is executed, it produces either a raw copy, or some type of protected copy as an output, based on the access right it enforces.

The sub-JAR stores the content file in an encrypted form along with the class files required to decrypt it. The key used for encryption is obtained using a Password-Based Key Derivation Function [123]. The password is not exchanged between the owner and the recipient. Rather, it is generated as a cryptographic hash of a subset of the randomized usernames appearing in the policy, with an additional salt value S. That is, $pwd = h(rid_1||\ldots||rid_m||S)$ where $rid_1, \ldots, rid_m$ are the user ids corresponding to sub-JARs $SubJ_1, \ldots, SubJ_m$ which either share the same access right (and time constraint, if any) with the recipient or are spurious sub-JARs, and S denotes a salt value -different from the ones used for randomizing usernames- which is passed along to the JAR using the same mechanism discussed for the username randomization. Spurious *rids* (and corresponding fake sub-JARs) are indistinguishable from randomized ids of actual usernames, and are added to increase the number of possible *rids* forming the alphabet of our password. Each such password, which has at least 80 bits of entropy, generates a unique 256 bit long key. An observer cannot identify which JARs share the same access rights: he cannot observe the execution of the JARs, and even if he accesses the java.policy file at the super-JAR, he only learns that a given *rid* has access to a JAR, which however does not lead to knowing what the JAR enforces.
JUiCE can be configured to deliver the salt values required by the recipient to

reconstruct the password in two different ways. Notice that per each individual recipient id_i two salt values are to be obtained: s_i, required to obtain the anonymized id rid_i and the salt S which is required to complete the password. These salt values can be either (1) stored with the class files, or (2) kept remotely as a part of the sender's profile on the authentication system. In the case of option (1), the salt is handled as a local variable, which is alive only within the function calling it after which its value changes and therefore is not available without the actual source files to a debugger or an external decompiler [124]. In the case of option (2) the salt is passed to the JAR, when the recipient authenticates himself to the authentication provider, and after the authentication provider verifies that the recipient is indeed on the list of recipients intended by the sender. This list of intended recipients is available to the authentication provider, for example from the email's To, CC or BCC fields.

A.2.2 Policy Enforcement

CPPs' enforcement has two major aspects to it: recipients' authentication and actual enforcement of the access rights. We discuss them in what follows.

Authentication. In order to obtain access, the recipient needs to provide proof of authentication, by passing along to the JAR, a valid authentication token. Such a token is obtained by a trusted authentication system which the JAR connects to. For example, the authentication token can be obtained from an email provider, or any trusted authentication provider. If the content is exchanged through a network of providers, SAML user assertions can be employed [105]. The token is used as a proof of identity, and to extract the username and the salt to decrypt the content according to the sender's policy. The main steps of the authentication are as follows:

1. The authentication token is verified by the super-JAR, which (a) checks the token's freshness and validity, and (b) extracts the corresponding user id, id_i.

2. Upon extracting the user id (task (b)), the super-JAR maps it into a rid, which corresponds to a JAAS [125] subject for whom the authorization rules are specified in the CPP. JAAS is employed so as to manage subjects' identities while allowing for a lightweight policy specification. JAAS in fact allows access rights specifica-

tion in a form which is handled by the Java Virtual Machine without any need for serialization. Next, the rule having rid_i as a target in the Java policy is found.
3. The super-JAR signs the token and passes it to the sub-JAR indicated by the policy.
4. The sub-JAR starts executing by first comparing the token it received from the super-JAR with the one it received initially on authentication, and checks its freshness.
5. If the token is successfully validated, the sub-JAR forms the decryption key. To obtain the decryption key, the password needs to be reconstructed. By construction, the $rid_1, \ldots, rid_m$ which are concatenated and hashed together to form the password are associated with JAR files that execute the same access right with the user who is attempting to open the content. Hence, the sub-JAR forming the decryption key first computes the *rid* assigned to it from the token passed by the super-JAR. This ensures that an attacker cannot bypass the authentication checks by authenticating with his own username to the authentication provider and sending as input a different *rid* to the sub-JAR. Next, to identify which subset of the randomized user ids are to be used, therefore, the sub-JARs are run sequentially to identify which sub-JARs correspond to a same access right. That is, the sub-JAR forming the decryption key calls all the other sub-JARs while acting as a superuser. The sub-JAR forming the decryption key notes the output of the sub-JAR being called. If the output is the same as its own, the sub-JAR being called has the same access right as itself. All of the randomized ids associated with the same access rights are hashed together to form the decryption key. The recipient cannot view the usernames corresponding to the selected *rids*, nor does he need to know which of the rids are of real users and which ones are not. The salt S is retrieved either by the authentication token or stored locally. Finally, the pwd is hashed one more time with the recipient's *rid* and used as input of Password-Based Key Derivation Function.
Relying on an external authentication party, does not imply that we require a trusted-JUiCE compliant authentication system. Rather, our approach supports any claim-based authentication system able to provide a valid authentication token.
Access and Usage Rights Enforcement. As anticipated, the privileges sup-

ported by the CPP are mapped into Java Policies. A Java policy specifies which permissions are available for a particular piece of code in a Java application environment. The permissions expressed in the Java policy are in terms of File System Permissions, i.e., in terms of read, write and execute. However, the sender can specify the permissions in user-centric terms as opposed to the usual code-centric security offered by Java, due to the JAAS. Each JAR can take one of the following forms.

Full privileges. If the recipient is given full privileges, the content must be available to him in a raw form. These privileges are guaranteed to him by a *JAR-Auxiliary* file, that decrypts the encrypted content and saves it on the local machine of the user. Simply double clicking the JAR file will result in the unencrypted, raw content being saved on the user's desktop.

Minimum Privileges. To provide minimum privileges, we embed the encrypted file in another JAR file, referred to as JAR-Core, which decrypts the content on the fly into temporary files, which are then used to display the content to the user using the Java application viewer. When the file is decrypted on the fly, the decrypted part is never available outside the JAR file, and is destroyed within less than a millisecond. Therefore an attacker never has the ability to decompile and retrieve the decrypted parts. The recipient cannot simply upload the entire JAR file to a content sharing sites, as these sites restrict the file types that can be uploaded to exclude JAR files. The strength of Java encryption and decryption scheme can be further improved using chaotic map lattices, which are secure against chosen plaintext and cipher text attacks [126].

Partial privileges. In our context, partial rights imply the ability of the subject to open the file and possibly distribute it, while however not being able to modify it. For example, subjects may upload the file to another content sharing website. The content is handled by a sub-JAR, namely the *JAR-Upload*, that renders it an applet, whenever the user tries to access it on the local machine. The JAR-Upload carries a specially watermarked version of the content, which indicates to a JAR-enabled handler that a user can upload the content stored in the JAR. When the content is uploaded, a JAR-enabled handler renders the content into its raw form. In the case of minimum privileges, an upload attempt would fail, due to the lack of the marking.

For the case of minimum and partial privileges, the sub-JARs are also responsible for displaying the content using a Java application viewer window or some other Java SWT equivalent [127]. The content rendered from these type of windows cannot be modified or saved on the users' Desktop in its raw form. Besides watermarking it, the sub-JARs also programmatically disables hotkeys of the keyboard so long as the viewer window is open, preventing the malicious user from copying the content being displayed.

The temporal constraints can be combined with any of the above privileges. As introduced, these constraints can enforce *open*, *release* or *revoke* actions. The constraints can be set to start either as soon as the JAR is created or as soon as the actual file is accessed. The JAR maintains the time stamp indicating the start time and the stop time, as well as a timer to keep track of the passing of time. The timer obtains the time from a NTP server. If the connection to the server is not available, the JAR revokes the access granted. If the temporal constraint takes effect upon creation of the JAR, the timer is immediately activated. If it is instead based on the time the JAR is opened, the JAR keeps track of the number of accesses. Both inner and outer JARs are signed and sealed to prevent the recipients from modifying the code or trying to access the class files from outside the package [127].

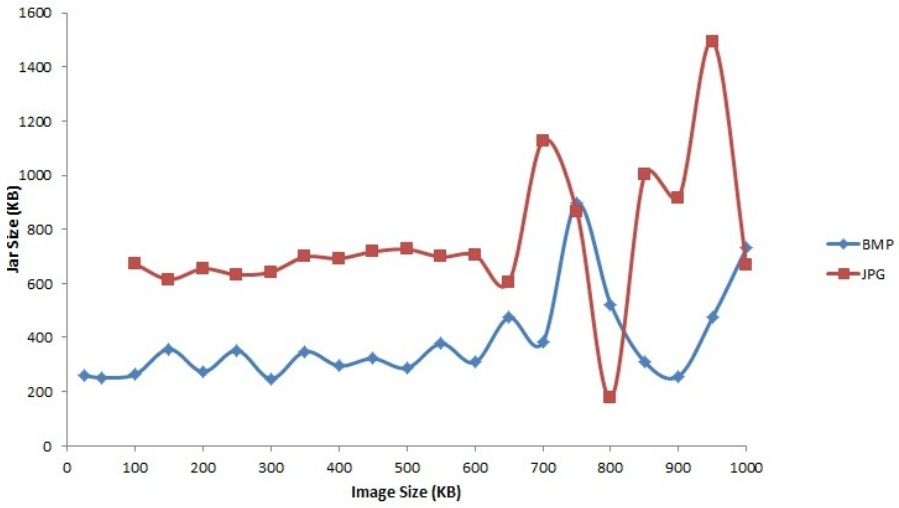

Figure A.1. Overhead due to encapsulation

A.3 Performance Evaluation

In order to evaluate the overhead added by our approach, we performed two separate sets of tests on a set of images which range in size from a 100KB to a 1000 KB. We tested both jpg and bitmap (BMP) files. Our tests were conducted using a Dell Latitude D630 Laptop, with 2G Ram and a Intel(R) Core(TM)2 Duo CPU T7500 @ 2.20GHz processor. The first set of tests is aimed at measuring the overhead in terms of increase in overall file size when the size or resolution of the image being protected increases. We reported a limited overall overhead due to the JARs. As shown in Figure A.1 , the increase in overall file size does not depend on the size of the image being protected. Rather, it is influenced by the resolution of the image, i.e. the number of pixels in the image. In the case of JPG images, the worst case occurs when a large image of 950KB results in a file which is approximately 1495KB. The best case for JPG images occurs when an image of 0.23 MegaPixels and a file size of 800KB is actually *compressed* into a file size of 277 KB. BMPs are overall more efficient, where the overall JAR size is only 895KB for a file of 700Kb. We believe that the fluctuations in the size of the JAR are due to the lossy compression of jpg files. Next, we tested the overhead added by evaluating the policies. CPPs take only 10^{-9}s, as compared to about 0.04ms for XML policies.

The second set of tests measured the overhead in terms of time taken to encrypt and decrypt the images. According to our approach, files will be encrypted and decrypted based on the access control rights specified in CPPs, and CPPs can be combined with any encryption and decryption algorithms supported by Java (i.e., AES, 3DES, PBECipher, RSA, RC2). The RC2 and RSA algorithms added the least amount of overhead (up to 300 ms) whereas the PBECipher with MD5 and Hash added the maximum overhead. The maximum time taken in case of the PBECipher with MD5 and Hash is 0.3 seconds. Based on such results, we used an RSA based implementation of Password-Based Key derivation for content encryption [123]. Again, the test results prove that there is no significant time overhead added due to the addition of CPPs. Finally, we also tested how much improvement in the file size increase can be obtained when protecting a dense image (18mp). We added a CPP of 27 rules to an image of 950KB, using the Triple

DES algorithm for encryption, since it fared well in our previous experiment. The increase in file size was minimal, resulting to only about 965KB, thus confirming that our architecture is very lightweight.

In this appendix, we discuss example properties, and their verification protocols while referring to the Azure Cloud implementation[20, 10]. Specifically, we examine what type of claims maybe submitted by a worker with respect to a given property. We then detail classes or methods can be used to describe the property. Finally, we describe the tasks that need to be completed before the property is deemed as satisfactorily verified. A parallel analysis can be carried out for Hadoop and other MapReduce frameworks.

A. Properties

Capabilities The capabilities of a node are important to verify policy satisfiability and identify candidate workers able to handle sensitive data (per the policy). In general, depending upon the task to be completed and related workload, a node can play several *roles*, or have specific *capabilities* in the Map Reduce framework. Here, we use the term role to reflect the activities and capabilities of a given worker. Roles can be of three main types: mapper, reducer or shuffler. For given applications, the mappers or reducers can further split the input, and therefore, additional roles may exist.

For instance, calculating the word count requires the mappers to generate a key-value pair of the type "word", "", while a mapper used for k-means clustering generates the distance between a point in a cluster and its centroid. Based on their tasks, the mappers can be differentiated in "kmeansmapper", "wordcountmapper" etc.

Similarly, a policy may express constraints against specific sub roles of the reducers. Referring to the above examples, a worker used in a word count application, sums up all the key-value pairs where the key words are similar. For the k-means clustering, the worker produces the sum of squares for a given cluster. It then finds the updated centroids for each cluster by dividing the sum of squares by the number of clusters. Since the tasks performed by each reducer are different, the reducers also have application-specific sub roles such as "kmeansreducer" or "wordcountreducer".

Location. Users may be interested in three locations. First is the hosted service location. Next are the storage locations, i.e. locations of of input and output, respectively.
The location of a physical host is expressed as a combination of the cloud service provider (CSP), and the region the node belongs to. The hosted service location is the location of the directory containing the client's application's required assemblies (i.e. the location of the worker forms a part of the hosted service location). The CSP specifies it in the package creation for a hosted service, and during the submission of an application. The input and output locations refer to the location of the directories where the inputs and outputs are placed, in form of an URI. The input and output locations form a part of the storage locations.

Proprietary, or Public/Private This binary property indicates whether a node is public or private. A "public" node denotes a node that belongs to the portion of the cloud to which the access is not curtailed, nor can the nodes be administrated by dividing them into subnets. On the contrary, a "private" node belongs to an access-restricted part of the cloud, which can be only accessed using a VPN connection. These nodes can be divided into subnets and administrated accordingly by the client. By identifying the proprietary nature of a node, one may identify whether there is any conflict based on the clouds the workers belong to. For example, in a public cloud, the workers can belong to any client, and therefore there is no clear separation of the workers' interests (i.e., a worker may be used for data processing by another client with conflicting interests to the original client), thus causing fear of information leakage. Further, a client may request that certain input is only processed by a proprietary cloud.

Security Security restrictions may be expressed to identify whether a node is capable of providing basic security functions. One such example is the support file level access control, or support of encryption/decryption. Additional properties refer to database (DB) access control (i.e. in case the input or output are stored in a DB, the access to this DB is modulated), private calculations (i.e., mandating that sensitive calculations be done only by nodes placed in a trusted or private cloud), storage security, and transport security etc. By means of policies constraining one

Protocol: Cap_Ver(worker.workertype, ipdataformat, opdataformat, ExpectOutA)

Input: *worker.workertype, ipdataformat, opdataformat,inputA* are the worker type (i.e. mapper or reducer), format of input and output data, and a sample input respectively
Begin
1. **Worker** →**Verifier**: Claim(context.taskno, context.cloudclient)
2. **Verifier** →**Worker**: Request to compute k-means((ipdataformat)inputA)
3. **Worker** →**Verifier**: (opdataformat)OutputA= k-means(inputA)
4. **Worker** →**Verifier**: OutputA
5. **Verifier**: If ((opdataformat)(OutputA)== (opdataformat)(context.taskno))
   AND (OutputA==ExpectOutA)
6. **Verifier**: Verified<–True
7. If Verified==True
8. **Verifier**→**Worker**: $\{k_w, Sig(k_w)\}$ %send key share
9. Else
10. **Verifier**→**Fail**

End

Figure A.2. Role/Capability Verification

or more of these properties, a user may gain some guarantees on the confidentiality of the MapReduce computation. Notice that while the above properties are a part of specifications of the workers, security configuration is typically not specified in any methods or constructors used for creating the worker. Instead, a worker is endowed with these properties during the submission of the job, by the Master, which uses the controller to either set up a certain protocol or algorithm, or transfer credentials such as TLS keys.

B. Properties' Verification protocols

We now present examples of properties' verification protocols. We briefly discuss a property verified by the Master, followed by an example of the methodology adopted to check nodes' location.

Proprietary The Master node checks whether a node is in a private or public cluster. The Master uses the *controller* module for this verification. The controller is the core algorithm that gets invoked and executed to manage job submissions

on behalf of its corresponding application. Since the *controller* is responsible for orchestrating the flow of the application, it has a complete mapping of public vs. private nodes. This is part of the definition of the cloud storage account created by the client at the time of instantiating his nodes. The *controller* specifically checks in the cloud storage account whether the node is assigned to a group deemed as private. In Azure, this information is available to the *controller* when it checks the *TaskNo* and *CloudClient* properties. The *TaskNo* is a number identifying the task, while *CloudClient* provides access to Azure's storage services. Therefore, the *controller* checks whether the *TaskNo* and the *CloudClient* of the *context* of the worker, matches that of a client with a proprietary network.

Capabilities verification. Verifying a worker's role means that the worker must be able to process input parameters of a given type to produce corresponding outputs; with the input and output parameters specified during job creation. This is a three step process.

1. The worker submits its claim to the verifier in charge of verifying the role. The claim consists of the pre-configured properties: that is, the type of input and output parameters the job is specified for. In Azure, this information is stored in the *context* class. The *context* class provides the configuration of the nodes, including the task no., cloud client, job parameters, task parameters and iteration count. The *context* class of mapper and reducer additionally contain the records enumerator property, which is an enumerator used to iterate over all the key value pairs.

2. The verifier then requests the worker for a demo-run on certain small "sample" jobs, on specified input (which output is also known). Notice that the verifier knows the sample jobs to assign to the worker based on the input provided by the Master (Step 3 of the verification protocol of Sect. 5.4.1). In case of Azure, this is specified using the *inputdataformat* and *outputdataformat* properties of the *controller*. Additionally, the *MapperType* and *ReducerType* are also specified in the properties of *JobConfiguration* along with the job parameters. The *inputdataformat* and *outputdataformat*, along with the *MapperType* and *ReducerType* in particular, are what help the verifier select its sample jobs.

3. When the output of the sample jobs are submitted back to the verifier, and the verifier checks whether the obtained output is correct and consistent with the claim made by the worker. In Azure, for a worker to be verified, the output of the worker should match the expected output type for the task no, and cloud client. A step-by-step presentation of the protocol for the *K-means* example is reported in Figure A.2.

Notice that, for both cases, the node may purposely pass the verification protocol and then fail to apply the security mechanisms in place. For instance, assume a worker is required to have a particular encryption property. It could pass the verification for a property, but then refrain from applying the security algorithm to the data being processed. To avoid this issue, the verifier periodically checks the output of the processing for the encryption being requested.

Location verification involves 1) Checking the worker nodes to ascertain that the input, output locations specified match the actual input and output locations, and the location of the hosts of the VMs performing the computation are the same as the location of the computation specified by the user. 2) Checking whether the directories specified for the input and output, and the computation assemblies indeed exist. The latter location verification protocol is treated by our system as a security verification task, and is similar to file access security verification (see Appendix A, Security Verification), i.e. the verifier tries to either store or access a document from the specified directory.

To estimate a node's location with reasonable accuracy, the verifier analyzes the round trip time (RTT) of a message sent by the verifier to estimate its source, following an approach similar to the mulitlateration scheme used for distance verification in mobile ad-hoc networks [128]. The multilateration scheme uses distance-bounding protocols [94] which are in turn dependent on estimating the location based of a node on response time from the node.

In our scheme, reported in Figure 5, multiple sub-verifiers, coordinated by the verifier node, are involved. Each sub-verifier is selected from a different location. The verifier knows the maximum number of hops, minimum number of hops and the average number of hops from each geographical location wherein the sub-verifiers are located. These values are stored by the verifier in a hash table upon execution of the protocol. Every sub-verifier requests the worker to echo a message within a

given number of hops or within a specified time time interval. The number of hops and the time constraint requested by each verifier vary, so that the worker does not know the location being requested beforehand. That is, the purpose of varying the requested time/ number of hops is that the worker being verified cannot tell one request apart from another, and therefore even if it uses proxies, or otherwise delays the reply, it will not be able to selectively satisfy the requests. The verifier uses minimum and maximum time/number of hops collected by the sub-verifiers to approximate the worker's location. That is, upon receiving the echoes from the worker, the verifier performs a lookup int the hash table to see if the location claimed by the worker, is consistent with the responses obtained by the workers (and matched against the local table). If the maximum and the minimum differ widely enough to show two different geographical locations, then the worker is lying either by using a proxy or by otherwise delaying the message.

Security Capabilities The properties listed in the policy specified by the client could be numerous, and the corresponding verification protocol may change accordingly. We briefly discuss the verification of two security properties: that is support of certain data encryption standard, and file level access control.
Cryptographic support (as a generic capability) is verified using the cryptographic algorithm verification program provided by the NIST [95]. The cryptographic module verification program maintains a list of implementations of various algorithms such as the AES, DED, Triple-DES. For each of the algorithms, the module has a set of tests built to verify different modes of operation of the algorithm such as ECB, CBC, with different key sizes if necessary/applicable. The module requests configuration information, and then provides the worker with some test data (i.e. the key, some plaintext, and an initialization vector if applicable), which is to be processed, and returned to the module. This processed test data is then validated to identify whether the worker's implementation of the algorithm is indeed standard-compliant. For instance, if the worker is verified for AES in CBC mode, using the Monte-Carlo Test, the worker is provided with a key, initialization vector and some plaintext. The response of the worker consists of the key, initialization vector, the plaintext and the resulting cipher text. If the cipher text is evaluated to be correct, the worker is said to have a standard-compliant algorithm.

Protocol: Location_Verification(CharToEcho, Nonce)
Input: *CharToEcho* is a random character, or set of characters,
Nonce is pseudorandom number used to identify freshness of the echo, typically the time
1. **Worker**−>**Verifier**: Claim %Geo-Location of physical host
2. **Sub-Verifier**−>**Worker**: Request= CharToEcho, Nonce
3. **Worker**−>**Sub-Verifier**: Response= Echo(CharToEcho, Nonce)
4. **Sub-Verifier**: Output = Response.time()- Request.time()
5. **Sub-Verifier**−>**Verifier**: Output
5. **Verifier**: If (Output)==val && Geo-Location(Sub-Verifier)=x Then
6. Geo-Location(Worker)=$location(val)_i$
7. For Each Geo-Location(Sub-Verifier)==x
8. Geo-Location(Worker)=$\bigcap location(val)_i$ %triangulation of the possible locations
9. If Geo-Location(Worker)== Claim
10. **Verifier**: Worker(Verified)< −−True
11. If Worker(Verified)==True Then
Verifier−− >**Worker**: $\{k_w, Sig(k_w)\}$ %send key share
12. Else **Verifier**−− >**Fail**

Figure A.3. Location Verification

File level access control is also checked using a multi-party protocol. In brief, multiple nodes (or sub-verifiers) execute repeated attempts to access a given protected file stored on a URI pointing to a directory location on the worker being verified. The verifier randomly selects few nodes, which are either public or private (if any, belonging to the same cluster of the worker), as well as the Master node. It then requests the nodes to attempt access to the specified directory, without informing the worker on the types and number of attempts being executed. If the file is accessible by all nodes, then the verifier concludes that no file level permissions are applied. On the other hand, if only the Master and private nodes can access the file, then the file level access control is presumed in place. Based on whether or not the worker grants access to the file (i.e. the worker's response to the file access request), the worker's response is judged as correct or incorrect. Specifically, when the verifier requests permission to a private file, if the verifier is private and it gets access, then the worker's response is correct. Otherwise it is incorrect.

Bibliography

[1] Mell, P. and T. Grance (2009), "NIST Definition of cloud computing," .

[2] Cusumano, M. (2010) "Cloud computing and SaaS as new computing platforms," *Commun. ACM*, **53**(4), pp. 27–29.
URL http://doi.acm.org/10.1145/1721654.1721667

[3] Armbrust, M., A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia (2010) "A view of cloud computing," *Commun. ACM*, **53**(4), pp. 50–58.
URL http://doi.acm.org/10.1145/1721654.1721672

[4] La, H. J. and S. D. Kim (2009) "A Systematic Process for Developing High Quality SaaS Cloud Services," in *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, Springer-Verlag, Berlin, Heidelberg, pp. 278–289.

[5] Bernstein, D., N. Vidovic, and S. Modi (2010) "A Cloud PAAS for High Scale, Function, and Velocity Mobile Applications - With Reference Application as the Fully Connected Car," in *Proceedings of the 2010 Fifth International Conference on Systems and Networks Communications*, ICSNC '10, IEEE Computer Society, Washington, DC, USA, pp. 117–123.
URL http://dx.doi.org/10.1109/ICSNC.2010.24

[6] Chohan, N., C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski (2009), "AppScale Design and Implementation," .

[7] Hay, B., K. Nance, and M. Bishop (2011) "Storm Clouds Rising: Security Challenges for IaaS Cloud Computing," *Hawaii International Conference on System Sciences*, **0**, pp. 1–7.

[8] Lenk, A., M. Klems, J. Nimis, S. Tai, and T. Sandholm (2009) "What's inside the Cloud? An architectural map of the Cloud landscape," in *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of*

Cloud Computing, CLOUD '09, IEEE Computer Society, Washington, DC, USA, pp. 23–31.
URL http://dx.doi.org/10.1109/CLOUD.2009.5071529

[9] Amazon (2009), "Amazon EMR with the MapR Distribution for Hadoop," Http://aws.amazon.com/elasticmapreduce/mapr/.

[10] Barga, R. (2011), "Project Daytona: Iterative MapReduce on Windows Azure," .

[11] Dean, J. and S. Ghemawat (2008) "MapReduce: simplified data processing on large clusters," *Commun. ACM*, **51**(1), pp. 107–113.
URL http://doi.acm.org/10.1145/1327452.1327492

[12] Cloud Computing Advices (2013), "Top 10 Cloud Computing Companies in 2013," .
URL http://cloudcomputingadvices.com/top-cloud-computing-companies-2013/

[13] Ristenpart, T., E. Tromer, H. Shacham, and S. Savage (2009) "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, ACM, New York, NY, USA, pp. 199–212.
URL http://doi.acm.org/10.1145/1653662.1653687

[14] Ingthorsson, O. (2011), "5 Cloud Computing Statistics You May Find Surprising," .
URL http://cloudcomputingtopics.com/2011/11/5-cloud-computing-statistics-you-may-find-surprising/

[15] Nichols, R. (2010), "Cloud computing by the numbers: What do all the statistics mean?" .
URL http://blogs.computerworld.com/16863/

[16] Cochrane, N. (2010), "Security experts ponder the cost of cloud computing," .
URL http://blogs.computerworld.com/16863/

[17] Cisco (2010), "Cisco Cloud Security: Choosing the Right Email Security Deployment," .
URL http://www.cisco.com/en/US/prod/collateral/vpndevc/ps10128/ps10339/

[18] VanTil, S. (2011), "Study on Cloud Computing Security: Managing Firewall Risks," .

URL http://resource.onlinetech.com/study-on-cloud-computing-security-managing-firewall-risks/

[19] Cloud Security Alliance, "Security Guidance for Critical Areas of Focus in Cloud Computing," .
URL https://cloudsecurityalliance.org/csaguide.pdf

[20] Microsoft (2010), "Windows Azure," .
URL http://www.windowsazure.com/en-us/

[21] "Amazon Web Services," .
URL http://aws.amazon.com/

[22] Lordan, F., E. Tejedor, J. Ejarque, R. Rafanell, J. lvarez, F. Marozzo, D. Lezzi, R. Sirvent, D. Talia, and R. Badia (2013) "ServiceSs: An Interoperable Programming Framework for the Cloud," *Journal of Grid Computing*, pp. 1–25.
URL http://dx.doi.org/10.1007/s10723-013-9272-5

[23] Vizard, M., "Hybrid Cloud Computing Faces Multiple Challenges," .
URL http://www.cioinsight.com/it-strategy/cloud-virtualization/hybrid-cloud-computing-faces-multiple-challenges.html/

[24] Taylor, S., A. Young, and J. Macaulay, "Small Businesses Ride the Cloud:SMB Cloud WatchU.S. Survey Result," .
URL http://www.cisco.com/web/about/ac79/

[25] Translattice, "TransLattice Unveils First Cross-Cloud Database," .
URL http://www.translattice.com/pr/

[26] Naehrig, M., K. Lauter, and V. Vaikuntanathan (2011) "Can homomorphic encryption be practical?" in *Proc. of the 3rd ACM workshop on Cloud computing security workshop*, ACM, pp. 113–124.
URL http://doi.acm.org/10.1145/2046660.2046682

[27] Brenner, M., J. Wiebelitz, G. von Voigt, and M. Smith (31 2011-June 3) "Secret program execution in the cloud applying homomorphic encryption," in *Proc. of the 5th IEEE International Conference on Digital Ecosystems and Technologies Conference (DEST)*, pp. 114–119.

[28] Blanton, M., J. M. Atallah, B. K. Frikken, and Q. M. Malluhi (2012) "Secure and Efficient Outsourcing of Sequence Comparisons," in *ESORICS*, Springer, pp. 505–522.

[29] Chen, X., J. Li, J. Ma, Q. Tang, and W. Lou (2012) "New Algorithms for Secure Outsourcing of Modular Exponentiations," in *ESORICS*, Springer, pp. 541–556.

[30] Burt, J., "Gartner Predicts Rise of Cloud Service Brokerages," .
URL http://www.eweek.com/c/a/Cloud-Computing/Gartner-Predict-Rise-of-Cloud-Service-Brokerages-759833/

[31] Gartner, "Cloud Services Brokerages: The Dawn of the Next Intermediation Age," .
URL http://www.gartner.com/technology/research/cloud-computing/cloud-services-brokerage.jsp

[32] Stickeleather, J., "Cloud service brokerage," .
URL http://en.community.dell.com/dell-blogs/enterprise/b/it-executive/archive/2011/02/22/cloud-service-brokerage.aspx

[33] Ivan, "Cloud Business Trend: Cloud Brokerage," .
URL http://www.cloudbusinessreview.com/2011/04/26/cloud-business-trend-cloud-brokerage.html

[34] SearchCloudApplications, "CloudSwitch," .
URL http://searchcloudapplications.techtarget.com/definition/CloudSwitch

[35] "RightScale," .
URL http://www.rightscale.com/

[36] Weitzner, D. J., H. Abelson, T. Berners-Lee, J. Feigenbaum, J. Hendler, and G. J. Sussman (2008) "Information accountability," *Commun. ACM*, **51**(6), pp. 82–87.

[37] Cavoukian, A. (2008) "Privacy in the clouds," *Identity in the Information Society*, **1**, pp. 89–108.

[38] Jaeger, P. T., J. Lin, and J. M. Grimes (2009) "Cloud Computing and Information Policy: Computing in a Policy Cloud?" *Journal of Information Technology and politics*, **5**(3).

[39] Kandukuri, B. R., R. V. Paturi, and A. Rakshit (2009) "Cloud Security Issues," in *IEEE International Conference on Services Computing*, pp. 517–520.

[40] Kaufman, L. (2009) "Data Security in the World of Cloud Computing," *Security Privacy, IEEE*, **7**(4), pp. 61–64.

[41] Mather, T., S. Kumaraswamy, and S. Latif (2009) *Cloud Security and Privacy: An Enterprise Perspective on Risks and Compliance (Theory in Practice)*, first ed., O' Reilly.

[42] Pearson, S. and A. Charlesworth (2009) "Accountability as a Way Forward for Privacy Protection in the Cloud," *Hewlett-Packard Development Company (HPL-2009-178)*.

[43] Lillibridge, M., S. Elnikety, A. Birrell, M. Burrows, and M. Isard (2003) "A Cooperative Internet Backup Scheme," in *USENIX Annual Technical Conference*, pp. 29–41.

[44] Schwarz, T. J. E. and E. L. Miller (2006) "Store, Forget, and Check: Using Algebraic Signatures to Check Remotely Administered Storage," in *IEEE International Conference on Distributed Systems*, p. 12.

[45] Ateniese, G., R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song (2007) "Provable data possession at untrusted stores," in *Proc, of ACM conference on Computer and communications security*, pp. 598–609.

[46] Wang, Q., C. Wang, J. Li, K. Ren, and W. Lou (2009) "Enabling Public Verifiability and Data Dynamics for Storage Security in Cloud Computing," in *ESORICS*, pp. 355–370.

[47] SearchCloudComputing, "Top 10 cloud computing providers of 2011," .
URL `http://searchcloudcomputing.techtarget.com/feature/Top-10-cloud-computing-providers-of-2011`

[48] Eggebrecht, M., "Is Cloud Brokerage the Next Big Thing?" .
URL `http://www.ciozone.com/index.php/Cloud-Computing/Is-Cloud-Brokerage-the-Next-Big-Thingu.html`

[49] Miller, R., "Cloud Brokers: The Next Big Opportunity?" .
URL `http://www.datacenterknowledge.com/archives/2009/07/27/cloud-brokers-the-next-big-opportunity`

[50] Buyya, R., C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic (2009) "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation computer systems*, **25**(6), pp. 599–616.

[51] Xin, L. and A. Datta (2010) "On trust guided collaboration among cloud service providers," in *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2010 6th International Conference on*, IEEE, pp. 1–8.

[52] Rao, J. and J. Su (2005) "A survey of automated web service composition methods," *Semantic Web Services and Web Process Composition*, pp. 43–54.

[53] Kalepu, S., S. Krishnaswamy, and S. W. Loke (2003) "Verity: a QoS metric for selecting Web services and providers," in *Web Information Systems Engineering Workshops, 2003. Proceedings. Fourth International Conference on*, IEEE, pp. 131–139.

[54] Zeng, L., B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang (2004) "QoS-aware middleware for web services composition," *Software Engineering, IEEE Transactions on*, **30**(5), pp. 311–327.

[55] Paolucci, M., T. Kawamura, T. Payne, and K. Sycara (2002) "Semantic matching of web services capabilities," *The Semantic WebISWC 2002*, pp. 333–347.

[56] Benatallah, B., M. Dumas, Q. Sheng, and A. H. Ngu (2002) "Declarative composition and peer-to-peer provisioning of dynamic web services," in *Proceedings on 18th International Conference on Data Engineering*, IEEE, pp. 297–308.

[57] Yu, C., B. C. Ooi, K.-L. Tan, and H. V. Jagadish (2001) "Indexing the distance: an efficient method to KNN processing," in *Proceedings of the international conference on Very large data bases (VLDB)*, pp. 421–430.

[58] Ciaccia, P., M. Patella, and P. Zezula (1997) "M-tree: An Efficient Access Method for Similarity Search in Metric Spaces," in *Proceedings of the International Conference on Very large data bases (VLDB)*, pp. 426–435.

[59] Despain, A. M. and D. A. Patterson (1978) "X-Tree: A tree structured multi-processor computer architecture," in *Proceedings of the annual symposium on Computer architecture*, pp. 144–151.

[60] Jagadish, H. V., B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang (2005) "iDistance: An adaptive B+-tree based indexing method for nearest neighbor search," *ACM Trans. Database Syst.*, **30**, pp. 364–397.

[61] Katayama, N. and S. Satoh (1997) "The SR-tree: an index structure for high-dimensional nearest neighbor queries," in *Proceedings of the ACM SIGMOD international conference on Management of data*, pp. 369–380.

[62] Lin, K.-l., H. V. Jagadish, and C. Faloutsos (1994) "The TV-tree: an index structure for high-dimensional data," *The VLDB Journal*, **3**, pp. 517–542.

[63] White, D. A. and R. Jain (1996) "Similarity Indexing with the SS-tree," in *Proceedings of the International Conference on Data Engineering (ICDE)*, pp. 516–523.

[64] Pearson, S., y. Shen, and M. Mowbray (2009) "A Privacy Manager for Cloud Computing," in *CloudCom*, pp. 90–106.

[65] Chun, B. and A. Bavier (2004) "Decentralized trust management and accountability in federated systems," in *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*, pp. 9 pp.–.

[66] Kailar, R. (1996) "Accountability in Electronic Commerce Protocols," *IEEE Trans. Software Eng.*, **22**(5), pp. 313–328.

[67] Crispo, B. and G. Ruffo (2001) "Reasoning about Accountability within Delegation," in *ICICS*, pp. 251–260.

[68] Corin, R., S. Etalle, J. Hartog, G. Lenzini, and I. Staicu (2005) "A Logic for Auditing Accountability in Decentralized Systems," in *Formal Aspects in Security and Trust* (T. Dimitrakos and F. Martinelli, eds.), vol. 173 of *IFIP International Federation for Information Processing*, Springer US, pp. 187–201.

[69] Jagadeesan, R., A. Jeffrey, C. Pitcher, and J. Riely (2009) "Towards a Theory of Accountability and Audit," in *ESORICS*, pp. 152–167.

[70] Lee, W., A. C. Squicciarini, and E. Bertino (2009) "The Design and Evaluation of Accountable Grid Computing System," in *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009)*, pp. 145–154.

[71] Holford, J. W., W. J. Caelli, and A. W. Rhodes (2004) "Using self-defending objects to develop security aware applications in Java," in *Proceedings of the 27th Australasian conference on Computer science - Volume 26*, pp. 341–349.

[72] Squicciarini, A., S. Sundareswaran, and D. Lin (2010) "Preventing Information Leakage from Indexing in the Cloud," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pp. 188–195.

[73] Feng, X., Z. Ni, Z. Shao, and Y. Guo (2007) "An open framework for foundational proof-carrying code," in *Proc. of the ACM SIGPLAN international workshop on Types in languages design and implementation*, pp. 67–78.

[74] Hasan, R., R. Sion, and M. Winslett (2009) "The case of the fake Picasso: preventing history forgery with secure provenance," in *Proccedings of the 7th conference on File and storage technologies*, USENIX Association, Berkeley, CA, USA, pp. 1–14.

[75] Bose, R. and J. Frew (2005) "Lineage retrieval for scientific data processing: a survey," *ACM Comput. Surv.*, **37**, pp. 1–28.

[76] Buneman, P., A. Chapman, and J. Cheney (2006) "Provenance management in curated databases," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, ACM, New York, NY, USA, pp. 539–550.

[77] Kagal, L., T. Finin, and A. Joshi (2001) "Moving from security to distributed trust in ubiquitous computing environments," *IEEE Computer*, **34**(12), pp. 154–157.

[78] Zhang, K., X. Zhou, Y. Chen, X. Wang, and Y. Ruan (2011) "Sedic: privacy-aware data intensive computing on hybrid clouds," in *Proc. of the 18th ACM conference on Computer and communications security*, CCS '11, ACM, pp. 515–526.

[79] Roy, I., S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel (2010) "Airavat: security and privacy for MapReduce," in *Proc. of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, USENIX Association, Berkeley, CA, USA, pp. 20–20.
URL `http://dl.acm.org/citation.cfm?id=1855711.1855731`

[80] Moca, M., G. Silaghi, and G. Fedak (2011) "Distributed Results Checking for MapReduce in Volunteer Computing," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pp. 1847–1854.

[81] Wei, W., J. Du, T. Yu, and X. Gu (2009) "SecureMR: A Service Integrity Assurance Framework for MapReduce," in *Proc. of Computer Security Applications Conference, ACSAC*, pp. 73–82.

[82] Huang, C., S. Zhu, and D. Wu (2012) "Towards Trusted Services: Result Verification Schemes for MapReduce," in *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pp. 41–48.

[83] Vu, V., S. Setty, A. Blumberg, and M. Walfish (2013) "A Hybrid Architecture for Interactive Verifiable Computation," in *Security and Privacy (SP), 2013 IEEE Symposium on*, pp. 223–237.

[84] Nair, K., Srijith, P. Pawar, A. Sajjad, M. Kiran, and M. Jiang (2011), "Requirements and Architecture of a Cloud Broker," .
URL `http://www.optimis-project.eu/sites/default/files/content-files/document/`

[85] Sriharee, N., T. Senivongse, K. Verma, and A. Sheth (2004) "On using ws-policy, ontology, and rule reasoning to discover web services," *Intelligence in Communication Systems*, pp. 246–255.

[86] Mondal, A., K. Yadav, and S. Madria (2010) "EcoBroker: An economic incentive-based brokerage model for efficiently handling multiple-item queries to improve data availability via replication in mobile-p2p networks," *Databases in Networked Information Systems*, pp. 274–283.

[87] Hartigan, J. A. and M. A. Wong (1979) "Algorithm AS 136: A k-means clustering algorithm," *Applied Statistics*, **28**, pp. 100–108.

[88] Jagadish, H. V., B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang (2005) "iDistance: An adaptive B+-tree based indexing method for nearest neighbor search," *ACM Trans. Database Syst.*, **30**, pp. 364–397.

[89] Connor, M. and P. Kumar (2010) "Fast Construction of k-Nearest Neighbor Graphs for Point Clouds," *Visualization and Computer Graphics, IEEE Transactions on*, **16**(4), pp. 599–608.

[90] Takabi, H. and J. B. Joshi (2012) "Semantic–based policy management for cloud computing environments," *International Journal of Cloud Computing*, **1**(2), pp. 119–144.

[91] Myers, A. C. (1999) "JFlow: Practical mostly-static information flow control," in *Proc. of the 26th SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, pp. 228–241.

[92] Dalton, M., H. Kannan, and C. Kozyrakis (2007) "Raksha: a flexible information flow architecture for software security," in *ACM SIGARCH Computer Architecture News*, vol. 35, ACM, pp. 482–493.

[93] Shamir, A. (1979) "How to share a secret," *Commun. ACM*, **22**(11), pp. 612–613.
URL `http://doi.acm.org/10.1145/359168.359176`

[94] Capkun, S. and J.-P. Hubaux (2005) "Secure positioning of wireless devices with application to sensor networks," in *Proc. of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 3, pp. 1917–1928 vol. 3.

[95] of Standards, N. I. and Technology (2013), "Cryptographic Module Validation Program Management," .
URL `http://csrc.nist.gov/groups/STM/cmvp/index.html`

[96] Dutta, D., A. Goel, R. Govindan, and H. Zhang (2003) "The design of a distributed rating scheme for peer-to-peer systems," in *Workshop on Economics of Peer-to-Peer Systems*, vol. 264, pp. 214–223.

[97] Saroiu, S., K. P. Gummadi, and S. D. Gribble (2001) "Measurement study of peer-to-peer file sharing systems," in *Electronic Imaging 2002*, pp. 156–170.

[98] Stoica, I., R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan (2001) "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proc. of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, ACM, pp. 149–160.
URL `http://doi.acm.org/10.1145/383059.383071`

[99] Rabin, M., R. Servedio, and C. Thorpe (2007) "Highly Efficient Secrecy-Preserving Proofs of Correctness of Computations and Applications," in *Proc. of 22nd Annual Symposium on Logic in Computer Science*, ACM, pp. 63–76.

[100] "Flickr," .
URL `http://www.flickr.com/`

[101] Trak.in, "Very Interesting Social Media Statistics: Facebook, Twitter, Flickr, Linkedin and more," .
URL `http://trak.in/tags/business/2010/02/01/social-media-statistics-facebook-twitter-flickr-linkedin/`

[102] Emulab, "Emulab Network Emulation Testbed," .
URL `www.emulab.net`

[103] Systems, E., "Eucalyptus Cloud Computing Software," .
URL `http://www.eucalyptus.com/`

[104] Boneh, D. and M. K. Franklin (2001) "Identity-Based Encryption from the Weil Pairing," in *Proc. of the International Cryptology Conference on Advances in Cryptology*, pp. 213–229.

[105] Committee, O. S. S. T., "Security Assertion Markup Language (SAML) 2.0," .
URL `https://www.oasis-open.org/committees/security/`

[106] Schneier, B. (1993) *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, John Wiley & Sons, Inc., New York, NY, USA.

[107] NTP, "The Network Time Protocol," .
URL http://www.ntp.org/

[108] Hightower, J. and G. Borriello (2001) "Location systems for ubiquitous computing," *Computer*, **34**(8), pp. 57 –66.

[109] Ammann, P. and S. Jajodia (1993) "Distributed timestamp generation in planar lattice networks," *ACM Trans. Comput. Syst.*, **11**, pp. 205–225.
URL http://doi.acm.org/10.1145/152864.152865

[110] Wicker, S. B. and V. K. Bhargava (eds.) (1999) *Reed-Solomon Codes and Their Applications*, John Wiley & Sons, Inc., USA.

[111] Roman, S. (1992) *Coding and information theory*, Springer-Verlag New York, Inc., New York, NY, USA.

[112] Chen, Y., R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. Jakubowski (2003) "Oblivious Hashing: A Stealthy Software Integrity Verification Primitive," in *Information Hiding* (F. Petitcolas, ed.), vol. 2578 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 400–414.

[113] Oaks, S. (2001) *Java security*, O'Really.

[114] Google, "Google Application Engine," .
URL http://code.google.com/appengine/

[115] "Salesforce," .
URL http://www.salesforce.com/

[116] "Proofpoint," .
URL http://www.proofpoint.com/

[117] RightScale, "Cloud Computing. Delivered," .
URL http://www.rightscale.com/

[118] Workday, "Cloud Computing," .
URL http://www.workday.com/

[119] Wustenhoff, E. (2010) "Top Ten Things to Consider About Cloud Services: Find the right services for your cloud computing project," *Cloudbook*, **2**.

[120] IBM, "Trusted Java Virtual Machine," .
URL http://www.almaden.ibm.com/cs/projects/jvm/

[121] Aviram, A., S. Hu, B. Ford, and R. Gummadi (2010) "Determinating timing channels in compute clouds," in *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, CCSW '10, ACM, New York, NY, USA, pp. 103–108.
URL http://doi.acm.org/10.1145/1866835.1866854

[122] Zander, S., G. Armitage, and P. Branch (2007) "A Survey of Covert Channels and Countermeasures in Computer Network Protocols," *Commun. Surveys Tuts.*, **9**(3), pp. 44–57.
URL http://dx.doi.org/10.1109/COMST.2007.4317620

[123] Black Duck Koders.com, "Pbkdf2 java," .
URL http://forums.sun.com/thread.jspa?threadID=5306039

[124] Agesen, O., D. Detlefs, and J. E. Moss (1998) "Garbage collection and local variable type-precision and liveness in Java virtual machines," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, ACM, New York, NY, USA, pp. 269–279.
URL http://doi.acm.org/10.1145/277650.277738

[125] Sun (2001), "Java Authentication and Authorization Service (JAAS) reference guide for the Java™2 SDK," .
URL http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/JAASRefGuide.html

[126] Bishop, D. (2003) *Introduction to Cryptography with Java Applets*, Cryptography Series, Jones and Bartlett.
URL http://books.google.com/books?id=yxPnt4S3mFMC

[127] Horstmann, C. S. and G. Cornell (2008) *Core Java: Volume II - Advanced Features, Eight Edition*, Prentice Hall.

[128] Capkun, S., M. Hamdi, and J.-P. Hubaux (2001) "GPS-free positioning in mobile ad-hoc networks," in *Proc. of the 34th Annual Hawaii International Conference on System Sciences*, IEEE, pp. 10–pp.

Vita

Smitha Sundareswaran

Smitha Sundareswaran received her bachelors degree in electronics and communications engineering in 2005 from Jawaharlal Nehru Technological University, Hyderabad, India. She started pursuing her Master of Science In Electrical Engineering at the Pennsylvania State University in 2006. She then joined the PhD program in the College of Information Sciences and Technology at the Pennsylvania State University. Her research interests include security, accountability, policy formulation and enforcement for Distributed computing architectures, particularly Cloud Computing. She has also worked on security and privacy in Social Networks, and security in web applications during her PhD. During her PhD she worked as an intern at Microsoft Research.

Selected Publications

- Smitha Sundareswaran, and Anna Squicciarini. "Detecting Malicious Coresident Virtual Machines Indulging in Load-Variation Attacks". To Appear in Fourth International Conference on Information and Communication Systems (ICICS 2013).
- Smitha Sundareswaran, Anna Squicciarini, Dan Lin, and Shuo Huang. "Ensuring Distributed Accountability for Data Sharing in the Cloud". IEEE Transaction on Dependable and Secure Computing (TDSC),Vol. 9(4), Pages 556-568. 2012.
- Smitha Sundareswaran, Anna Squicciarini, and Dan Lin. "A Brokerage-Based Approach for Cloud Service Selection". 5th IEEE International Conference on Cloud Computing (CLOUD), June 2012.
- Smitha Sundareswaran, Anna Squicciarini, and Ranjani Sundareswaran. "JUiCE: Java-based Distributed Content Protection". 35th IEEE Annual International Computer Software and Applications Conference (COMPSAC), July 2011.
- Smitha Sundareswaran, Anna Squicciarini, Dan Lin, and Shuo Huang. "Promoting Distributed Accountability in the Cloud". 4th IEEE International Conference on Cloud Computing (CLOUD), July 2011.
- Anna Squicciarini,Smitha Sundareswaran, and Dan Lin. "Preventing Information Leakage from Indexing in the Cloud". IEEE International Conference on Cloud Computing, July 2010.