Abstract

A Flexible Architecture for Structural Reliability Auditing to the Clouds

Ennan Zhai

2015

Today's cloud computing systems pervasively rely on redundancy techniques to enhance reliability and availability. In complex multi-layered hardware/software stacks, however, seemingly independent components used redundantly might share deep, hidden dependencies. These common dependencies may potentially result in unexpected correlated failures, thus undermining redundancy efforts.

Heading off correlated failures is extremely challenging in cloud-scale systems for the following three reasons. First, infrastructure components and their dependencies underlying cloud-scale redundant systems are typically very complex. Second, it is non-trivial to efficiently and accurately determine the common dependencies potentially resulting in correlated failures within cloud-scale system environment. Third, application-level cloud services (*e.g.*, iCloud) renting multiple cloud providers for redundancy cannot identify dependencies shared by these rented cloud providers, since no cloud provider is willing to disclose such sensitive information.

This thesis proposes *Independence-as-a-Service* (or INDaaS), a practical and flexible architecture to audit the independence of redundant systems proactively, thus avoiding potential correlated failures fundamentally. INDaaS first utilizes pluggable dependency acquisition modules to automatically collect the structural dependency information (including network, hardware, and software level dependencies) from a variety of sources underlying the audited cloud services. With this information in hand, INDaaS then evaluates the independence of redundant systems of interest using pluggable auditing modules, offering various performance, precision, and data secrecy trade-offs. For redundant systems across multiple cloud providers unwilling to share their full structural information with others, INDaaS can leverage privacy-preserving set intersection cardinality protocols to quantify the independence of the redundant systems without leaking any sensitive information of the cloud providers. We evaluate the practicality of INDaaS with three case studies that audit realistic network, hardware, and software dependency structures, and evaluate the effectiveness and efficiency of INDaaS through large-scale datasets.

A Flexible Architecture for Structural Reliability Auditing to the Clouds

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Ennan Zhai

Dissertation Director: Bryan A. Ford

December 2015

To my parents and my wife

Acknowledgments

I owe a great amount of gratitude to my advisor, Bryan A. Ford, for his constant guidance, intellectual insight and extraordinary patience, and for giving me the freedom to explore my own route. His impact on my research is beyond words.

I would like to express my sincere gratitude to my committee members, Joan Feigenbaum, Wenjun Hu and Ruichuan Chen, for many insightful discussions on my research. I also wish to thank other faculty members at Yale for their additional guidances and stimulating discussions on specific research topics: Dana Angluin, James Aspnes, Ruzica Piskac, and Zhong Shao.

This research work benefited a lot from the collaboration with David Isaac Wolinsky, and was improved by the valuable feedback of Gustavo Alonso, Jiewen Huang, Hongqiang Liu, Jeff Mogul, Timothy Roscoe, Xueyuan Su, Hongda Xiao, and the thirty one anonymous reviewers who pored over many drafts of this work.

I was fortunate to work alongside amazing colleagues of the DeDiS group at Yale, including Henry Corrigan-Gibbs, Michael F. Nowlan, Liang Gu, Daniel Jackowitz, John Maheswaran, Ewa Syta and Weiyi Wu. I enjoyed every minute with them.

Last but not least, I want to thank my parents for their unending encouragement and unwavering support, and thank my wife, Lewie, for her love, support and understanding.

Previous Publications

This thesis incorporates and extends work previously published in the following papers:

- Ennan Zhai, Ruichuan Chen, David Isaac Wolinsky, and Bryan Ford. Heading off correlated failures through Independence-as-a-Service. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, CO, Oct 2014. [88]

- Ennan Zhai, Ruichuan Chen, David Isaac Wolinsky, and Bryan Ford. An untold story of redundant clouds: Making your service deployment truly reliable. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems (HotDep)*, Farmington, PA, Nov 2013. [87]

- Ennan Zhai, David Isaac Wolinsky, Hongda Xiao, Hongqiang Liu, Xueyuan Su, and Bryan Ford. Auditing the structural reliability of the clouds. In *Yale University Technical Report TR-1479*, Jul 2013. [89]

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

— Leslie Lamport

Today's cloud services explicitly require high reliability. Typically, service providers pervasively rely on redundancy techniques to achieve this requirement [21,23,26,42]. Amazon S3, for example, replicates each data object across multiple racks in an S3 region [1]. iCloud, as a representative application-level cloud service, rents infrastructures from multiple cloud providers – both Amazon EC2 and Microsoft Azure – for redundancy [2]. Seemingly independent infrastructure components underlying redundant systems, however, may share deep, hidden dependencies. Failures in these shared dependencies may lead to unexpected *correlated failures* across the whole redundant systems, thus undermining redundancy efforts [32,41,46,56,60,87,89].

In a typical cloud service's redundant system, a *risk group* [47] (or RG) is defined as a set of components whose simultaneous failures could make the redundant system unavailable, thus causing the service outage. Suppose some service A replicates critical state across independent servers B, C and D located in three separate racks.

The intent of this three-way redundancy configuration is for all RGs to be of size three, *i.e.*, three servers must fail simultaneously to cause an outage. Unbeknownst to the service provider, however, the three racks share an infrastructure component such as an aggregation switch S. If the switch S fails for whatever reason, B, C and D could become unavailable at the same time, causing the service A to fail. We say such common dependency introduces an *unexpected RG*, defined as a smaller than expected RG, whose failures could disable the whole service despite redundancy efforts.

This example, while simplistic, nevertheless illustrates documented failures. In an Amazon AWS event [4], a glitch on one Amazon Elastic Block Store (EBS) server disabled the EBS service across Amazon's US-East availability zones, due to a latency bug in the data collection agent running on that server. The failure of the EBS service caused correlated failures across multiple Elastic Compute Cloud (EC2) instances utilizing that EBS for storage, and in turn disabled applications designed for redundancy across these EC2 instances. The EBS server in this example was a single common dependency, *i.e.*, an unexpected RG, that undermined the EC2's redundancy efforts.

Discovering and eliminating unexpected common dependencies is extremely challenging in practice [33, 35]. Many diagnostic and forensic approaches attempt to localize or tolerate such failures after they occur [19, 26, 29, 38–41, 44, 49, 55]. These retroactive approaches, however, still require human intervention, leading to prolonged failure recovery time [81]. Google has estimated that "close to 37% of failures are truly correlated" within its global storage system, but they lack the tools to identify these failure correlations systematically [33].

Even worse, correlated failures can be hidden not just by inadequate tools or analysts within *one* cloud provider, but also by non-transparent business contracts between cloud providers forming complex multi-level service stacks [32]. Application-level cloud services such as iCloud often redundantly rely on *multiple* cloud providers, *e.g.*, Amazon EC2 and Microsoft Azure [2]. However, a storm in Dublin recently took down a local power source and its backup generator, thus leading to *both* the Amazon and Microsoft clouds in that region to be unavailable for many hours, potentially undermining the reliability efforts of applications attempting to achieve redundancy across those two cloud infrastructure providers [30]. Providers of higher-level cloud services cannot readily know how independent the lower-level services they build on redundantly really are, since the relevant common dependencies (*e.g.*, power sources) are often proprietary internal information, which cloud providers do not normally share [32, 82, 87].

This thesis proposes *Independence-as-a-Service* or INDaaS, a practical and flexible architecture that aims to tackle the above problems proactively. Rather than localizing and tolerating failures after a service outage, INDaaS collects and audits structural dependency information to evaluate the independence of redundant systems before failures occur. Because INDaaS's clients (*e.g.*, cloud operators and cloud application developers) may have different requirements on the accuracy, performance, and secrecy of independence auditing, INDaaS is designed to support these needs through multiple pluggable modules, thus exhibiting its flexibility. In particular, INDaaS consists of a pluggable set of *dependency acquisition modules* that are responsible for collecting various dependency data, and an auditing agent that employs a similarly pluggable set of *auditing modules* to not only quantify the independence of redundant systems but also determine common dependencies that may introduce unexpected correlated failures.

In the dependency acquisition phase, we introduce a uniform representation for different types of dependency data, enabling dependency acquisition modules to be tailored and reused for a particular cloud provider's infrastructure. As an example, our experimental prototype was able to collect dependency data from various sources with respect to network topologies, hardware components, and software packages.

To represent this collected dependency data, INDaaS builds on the traditional fault analysis techniques [66, 74], and further adapts these techniques to audit the independence of redundant systems. Our fault graph representation supports three levels of detail appropriate in different situations: *component-sets*, *fault-sets*, and *fault graphs*. INDaaS can use component-sets to identify shared components even if no failure likelihood information is available. With fault-sets, INDaaS can take failure likelihood information into account. Fault graphs further enable INDaaS to account for deep internal structures involving multiple levels of redundancy.

In its auditing phase, INDaaS offers multiple auditing modules to address tradeoffs among performance, precision, and data secrecy. Our most powerful and informative auditing methods assume that a single independent auditing agent is able to obtain all the required structural dependency data in the clear. This assumption may hold if the agent is a system run by and within a single cloud provider, or if the agent is run by a trusted third party such as a cloud insurance company or a non-profit underwriting agency.

To support independence auditing even across mutually distrustful cloud providers who may be unwilling to share full dependency data with anyone, INDaaS offers *private independence auditing* (or PIA). We have explored two approaches to PIA. The first uses secure multi-party computation [85], which offers the best generality in principle but performs adequately only on small dependency datasets [82]. We therefore focus here on the second approach, based on private set intersection cardinality [51, 72]. This approach restricts INDaaS's auditing to the component-set level of detail, but we find it to be practical and scalable to large datasets.

We have developed a prototype INDaaS auditing system, and evaluated its performance with three small but realistic case studies. These case studies exercise INDaaS's two capabilities: 1) proactively quantifying the independence of given redundancy configurations, and 2) identifying potential correlated failures. We find that the prototype scales well. For example, the prototype can audit a cloud dependency structure containing $27,648$ servers and $2,880$ switches/routers, and identify about 90% of relevant dependencies, within 3 hours.

Our INDaaS prototype has many limitations, and would need to be refined and customized to particular cloud environments before real-world deployment. Nevertheless, even as a proof-of-concept, we feel that INDaaS represents one step towards building reliable cloud infrastructures whose redundancy structures can avoid various types of unexpected common-mode failures [36], emergent risks due to overwhelming complexity [56], and proprietary information barriers that naturally arise in the cloud ecosystem [32].

In summary, this thesis makes the following contributions:

1. the first practical and flexible architecture designed to audit the independence of redundant cloud systems before or during deployment;

2. adaptation of fault graph analysis techniques to support multiple levels of detail in explicit dependency structures;

3. an efficient fault graph analysis technique that scales to large datasets representing realistic cloud infrastructures;

4. an application of private set intersection cardinality techniques to enable efficient private independence auditing;

5. a prototype implementation and evaluation of INDaaS's practicality with small

but realistic case studies and larger-scale simulations.

Roadmap. This thesis is organized as follows. The next chapter clarifies our target problem and potential challenges. Then, we sketch INDaaS architecture and deployment in §3. Following the architecture, detailed designs of dependency acquisitions, structural independence auditing, and privacy-preserving independence auditing are given in §4, §5 and §6 respectively. After that, INDaaS's limitations and practical issues are discussed in §7. Next, we present the implementation of INDaaS and evaluate this prototype system in §8 and §9, respectively. Finally, §10 summarizes and discusses related work, and §11 concludes this work and presents future work.

Chapter 2

Problem Statement and Challenges

This chapter first clarifies our target problem (§2.1) and then lists associated technical challenges (§2.2).

2.1 Problem Statement

Under the assumption of failure independence, cloud providers typically leverage redundancy techniques to reduce the likelihood of failures, thus enhancing the reliability of their services.

Risk group. For a given service or application's redundant system, we define a *risk group* [47], RG, of this redundant system as a collection of the underlying components whose simultaneous failures would lead to the service outage.[1] For example suppose a cloud storage service employs three-way redundancy configuration upon its storage components. Its purpose is for all RGs to be of size at least three. In other words, at least three components must fail simultaneously to result in the service outage. However, if these three higher-level components unexpectedly depend on some *single*

[1]Failure of some component means the component does not function due to software, network or hardware problems.

lower-level component, *e.g.*, an aggregation switch, then this lower-level component represents a RG of size one in that its failure would make the whole service become unavailable. Heading off *unexpected RGs*, defined as smaller than expected RGs, is our goal.

Unexpected RGs within the clouds. Cloud service outages in reality have resulted from unexpected RGs due to common dependencies [64]. Well-known cloud IaaS services, such as Amazon EC2 and Microsoft Azure, try to utilize redundancy to ensure service reliability, *e.g.*, by introducing backup servers and switches in their data centers [16, 37, 57, 91]. However, additional redundancy may not mitigate the likelihood of failure due to a failed RGs, derailing providers' efforts to improve reliability [42]. Amazon, for example, experienced a significant disruption in the Northern Virginia EC2 data center due to correlated failures resulting from incorrect configuration on a few aggregation switches [5]. While Amazon repaired the failure after it occurred, the general problem, *i.e.*, an unexpectedly failed RG, would likely recur in the future.

Unexpected RGs across multiple cloud providers. As the cloud diversifies, some cloud service providers no longer depend upon only a single cloud service and have begun redundantly using other cloud providers for enhanced reliability [23]. For example, as a well-known cloud storage and computing platform, iCloud [2] redundantly builds its service upon infrastructure services from both Amazon EC2 and Microsoft Azure. Zynga [15], developer of many Facebook games, uses both EC2 and an internal "cloud" for redundancy. Despite the efforts, these providers may be unaware and unable to mitigate failures due to unexpected RGs shared by distinct cloud providers. For instance, a recent ferocious lightning storm in Northern Virginia [8,9] took out local primary and backup power supplies, resulting in unavailability of all the IaaS services in the region.

2.2 Technical Challenges

Several technical challenges appear in detailed analysis of our target problem. The following list focuses on those we view as critical toward building a practical solution on our problem.

Challenge 1: Acquiring dependencies. In order to discover and eliminate unexpected RGs within a cloud service deployment, we must be able to acquire the information about relevant components and their associated dependencies supporting this service. Because such infrastructures underlying the cloud tend to be very complex in practice, it is likely to be infeasible for the service provider or administrator to populate this information manually. Therefore, acquiring dependencies automatically becomes an important and challenging problem. Existing efforts towards this target found in monitoring and diagnosis systems have been limited to networking ignoring hardware and software level dependencies [29]. Recent reports have shown unsuitability of networking alone, as failures resulting from software and hardware become rather commonplace [13].

Challenge 2: Determining and evaluating RGs. Even with a set of components and their dependencies, it is still not obvious how best to determine unexpected RGs and evaluate their importance in a useful way. Within this challenge, there exists the need to construct an useful dependency graph and instrumenting it with potential failures. Determining RGs collected from target services provides a difficult challenge due to potentially complex dependencies. Existing efforts [19,52] have tried to solve similar problems with diagnosis analysis; however, these approaches may fail to offer accurate results when confronted with complex dependencies [81].

Challenge 3: Privacy Preservation. For a cloud service that rents infrastructure from other different cloud providers, performing structural auditing is almost impossible in practice. Since the infrastructure information is high secret proprietary information to every cloud provider, no provider is willing to share it. Auditing such multi-level cloud services, therefore, introduces another complex challenge: performing auditing of the target service without compromising the privacy of its cloud infrastructure providers. Two possible solutions would be: 1) to introduce a trusted third party, who collects dependency information from multiple cloud providers and performs auditing for the service; or 2) using secure multi-party computation (SMPC) [85] to compute the overlap privately, without exposing any information from which those results are computed. Unfortunately, in the former case cloud providers may be hesitant to trust a third-party, while the latter option has been proven to be time-consuming and does not scale well [82, 89].

Chapter 3

INDaaS Architecture Overview

This chapter presents a high-level overview of the INDaaS architecture including deployment (§3.1) and a typical workflow (§3.2). We defer details of each component of INDaaS to subsequent chapters.

3.1 INDaaS Deployment

Figure 3.1 shows deployment of INDaaS architecture. In practice, INDaaS architecture mainly involves three roles or types of entities: auditing client, dependency data source, and auditing agent.

The *auditing client* requests an audit of the independence of two or more cloud systems, which may either be operated by the auditing client herself or rented from other cloud providers, and which she believes to be independent so as to offer redundancy. For example, an auditing client may request a one-time independence audit prior to deploying a new service onto multiple redundant clouds, like iCloud's use of both Amazon EC2 and Microsoft Azure [2]. The auditing client may also request periodic audits on a deployed configuration to identify correlated failure risks that configuration changes or evolution might introduce. In practice, auditing clients

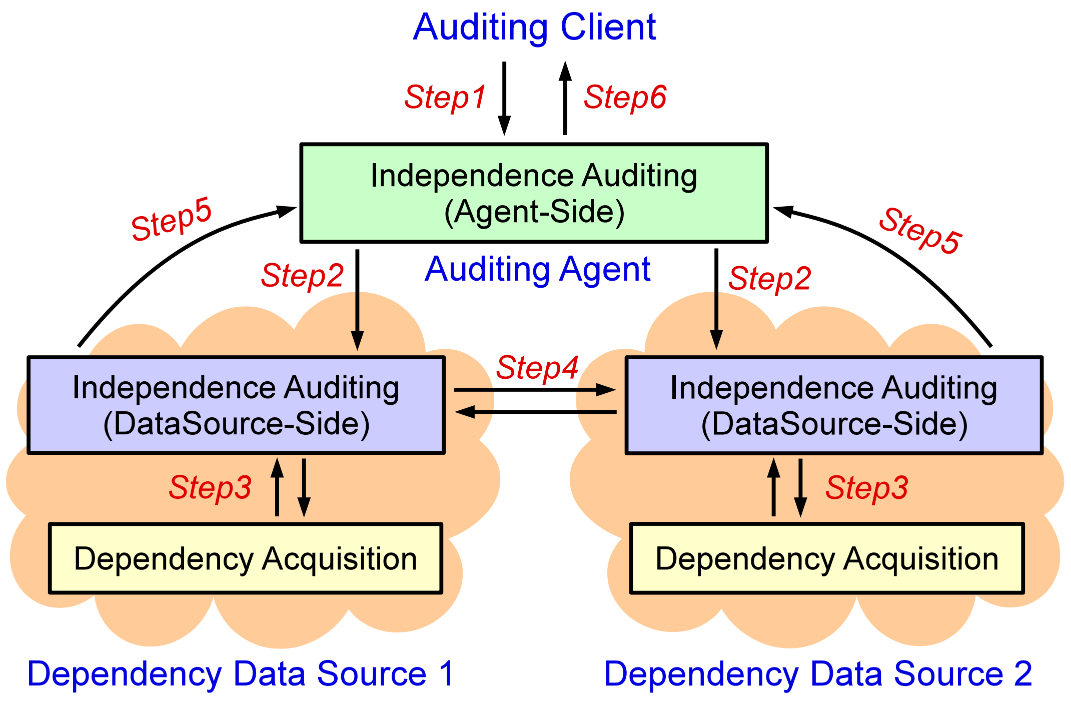


Figure 3.1: An example for INDaaS's auditing process. There are three main roles: auditing client, dependency data source, and auditing agent. In this example, an auditing client wishes to audit the independence of a two-way redundancy deployment.

could be the cloud operators of large organizations with vast resources and global interests such as Amazon EC2, Microsoft Azure, and Google Engine, or application developers with limited infrastructures resources such as Netflix, iCloud, and Zynga.

Dependency data sources (or data sources for brevity) represent the providers of cloud systems whose independence the auditing client wishes to check. The data sources in practice may be providers of computation, storage and networking components to be used redundantly by the auditing client. INDaaS might be deployed so as to utilize data sources either from a single provider or across multiple providers. In the first case, a storage service like Amazon S3 might provide data sources for each of multiple Amazon data centers offering intra-provider redundancy for S3. In the second, inter-provider scenario, Amazon EC2 and Microsoft Azure might serve as distinct data sources for redundant services rented by iCloud. Either way, as shown in Figure 3.1, each data source employs pluggable *dependency acquisition* modules to collect structural dependency data on its components such as network topology, hardware devices, or even software packages whose dependencies could lead to common-mode failures (*e.g.*, Heartbleed [36]).

The *auditing agent* mediates the interaction between the auditing client and the data sources. In the case where the auditing agent can obtain the dependency data from all the relevant data sources, the auditing agent constructs a dependency graph based on the data from these data sources. Then, the agent processes the dependency graph and quantifies its independence, or identifies any unexpected common dependencies which may potentially cause correlated failures, using a set of pluggable *independence auditing* modules. In the case of private independence auditing, the agent cannot obtain the full dependency data from data sources in cleartext, but supervises a private set intersection cardinality protocol performed by the data sources collaboratively.

3.2 INDaaS Workflow

We exhibit a typical independence auditing process as illustrated in Figure 3.1:

Step 1. An auditing client, say Alice, needs to specify to the auditing agent what services or redundant systems she wishes to audit and in what way. In particular, this specification normally includes: a) the relevant data sources; b) the level of redundancy desired; c) the types of components and dependencies to be considered; and d) the metrics used to quantify independence.

Step 2. By receiving the auditing client's specification, the auditing agent issues a request to each data source mentioned in the specification.

Step 3. Each specified data source uses one or more dependency acquisition modules to collect different types of the dependency data (*e.g.*, network, hardware and software level dependencies) for future independence auditing (see §4).

Step 4. For the situation that data sources are not willing to share dependency information with others, *i.e.*, private independence auditing (PIA) case, the data sources collaborate to compute the auditing results without revealing the proprietary dependency data to each other (see §6).

Step 5. Each of data sources returns to the auditing agent either the full dependency data for structural independence auditing (see §5), or in the PIA case, returns the collaboratively computed independence auditing results.

Step 6. The auditing agent returns to Alice an auditing report quantifying the independence of various redundancy deployments, optionally computing some useful information such as the estimates of correlated failure probabilities and ranked lists of potential risk groups.

Chapter 4

Dependency Acquisition

Acquiring accurate structural dependency data within heterogeneous cloud systems is our first technical challenge. Typically, realistic solutions would need to be adapted to different cloud environments and platforms. As many dependency acquisition tools have been deployed in today's clouds for various purposes (*e.g.*, system diagnosis) [18–20, 28, 29, 31, 44, 48, 49, 52], we expect such tools can be adapted and reused to collect the dependency data required by INDaaS. Towards this end, we propose INDaaS to leverage pluggable dependency acquisition modules (DAM), and maintains a uniform representation of different types of dependency data.

In this chapter, we first exhibit our design on flexible dependency acquisition (§4.1), and then describe a network dependency and failure probability acquisition tool developed by us to handle the situation that cloud providers do not have their own dependency acquisition tools (§4.2).

4.1 Flexible Dependency Acquisition in INDaaS

The basic idea of the design on flexible dependency acquisition is to introduce a uniform format (or representation) for different types of dependency data, thus enabling

Table 4.1: Uniform format definition of various dependencies.

Type	Dependency Expression
Network	`<src="S" dst="D" route="x,y,z"/>`
Hardware	`<hw="H" type="T" dep="x"/>`
Software	`<pgm="S" hw="H" dep="x,y,z"/>`

different dependency acquisition modules to be tailored and reused for a particular cloud providers' infrastructure.

In particular, different data sources first collect dependency data through their dependency acquisition systems or service monitoring systems, and then adapt the collected data to a common XML-based format illustrated in Table 4.1. Finally, the DAM stores the adapted dependency data in a database, DepDB, for further processing. This design supports different cloud environments as long as their providers can adapt the dependencies underlying their systems to our defined formats.

Table 4.1 shows how our prototype expresses network, hardware, and software dependencies. Each such dependency corresponds to one of the three most common causes of correlated failures [35, 81]: incorrect network configurations, faulty hardware components, and buggy or insecure software packages.[1]

A *network dependency* describes a route from source `S` to destination `D` via various network components in between, such as routers and/or switches `x`, `y`, and `z`.

A *hardware dependency* describes a physical component, *e.g.*, a disk or CPU of a server. The `hw` field denotes a physical component, and `type` specifies the type of this component such as CPU, disk, RAM, etc. The `dep` field specifies the model number of the component.

A *software dependency* describes the package information of a software component. The `pgm` field denotes the software component `S` itself, `hw` specifies the hardware

[1] We also considered power supply dependencies, which are also critical and can lead to correlated failures, but were unable to identify a systematic approach to collect power dependency information with widely-available tools.

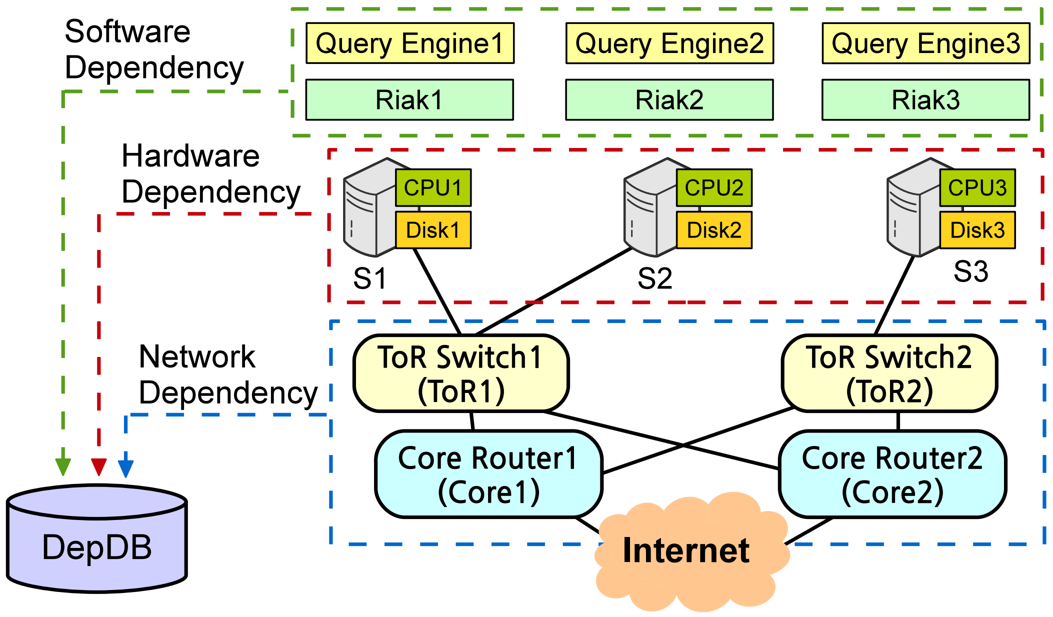


Figure 4.1: Example for a simple distributed storage system using three servers S1-S3. Riak i denotes individual Riak instance running on each machine, and, similarly, Query Engine i means the application which is responsible for querying the corresponding instance i.

`H` on which the `S` runs, and `dep` specifies various packages `x`, `y` and `z` used by `S`.

Our INDaaS prototype currently includes three dependency acquisition modules employing existing tools to collect various raw dependency data, then adapt them into the common format as discussed above. In particular, we employ NSDMiner [44, 59] to collect network dependencies, HardwareLister [6] to collect hardware dependencies, and apt-rdepends [3] to collect software dependencies. These first-cut INDaaS modules are in no way intended to be definitive but merely aim to provide some examples of realistic dependency acquisition methods.

NSDMiner is a traffic-based network data collector, which discovers network dependencies by analyzing network traffic flows collected from network devices or individual packets [44, 59]. HardwareLister (lshw) extracts a target machine's detailed hardware configuration including CPUs, disks and drivers [6]. The apt-rdepends tool

```
Network dependencies of S1 and S2:
<src="S1" dst="Internet" route="ToR1,Core1"/>
<src="S1" dst="Internet" route="ToR1,Core2"/>
<src="S2" dst="Internet" route="ToR1,Core1"/>
<src="S2" dst="Internet" route="ToR1,Core2"/>
------------------------------------
Hardware dependencies of S1 and S2:
<hw="S1" type="CPU" dep="S1-Intel(R)X5550@2.6GHz"/>
<hw="S1" type="Disk" dep="S1-SED900"/>
<hw="S2" type="CPU" dep="S2-Intel(R)X5550@2.6GHz"/>
<hw="S2" type="Disk" dep="S2-SED900"/>
------------------------------------
Software dependencies of S1 and S2:
<pgm="QueryEngine1" hw="S1" dep="libc6,libgccl">
<pgm="Riak1" hw="S1" dep="libc6,libsvn1">
<pgm="QueryEngine2" hw="S2" dep="libc6,libgccl">
<pgm="Riak2" hw="S2" dep="libc6,libsvn1">
```

Figure 4.2: A sample of the collected dependency data.

extracts the software package and library dependencies for popular Linux software distributions [3].

Figure 4.1 illustrates a sample distributed storage system. Suppose an auditing client desires two-way redundancy for her service running on two of the three servers S1-S3 within her cloud. She submits to the auditing agent a specification indicating: 1) IP addresses of the three servers, and 2) relevant software components running on these servers. Our current prototype requires the auditing client to list software components of interest manually – *e.g.*, Query Engine and Riak [11] (a distributed database) in this example. With this specification, the auditing agent invokes the dependency acquisition modules (*i.e.*, NSDMiner, lshw, and apt-rdepends) on each server to collect the network, hardware, and software dependencies, and store them in the DepDB, as shown in Figure 4.2.

4.2 Developed Network Dependency Acquisition

We also have developed a new network dependency acquisition tool, named NetAcq. The basic idea of NetAcq is to determine network component dependencies by analyzing flows of network traffic that are either collected from communication devices (*e.g.*, switches and routers) or created from individual packets. The output of NetAcq is a set of dependency information (formated as defined in Table 4.1) about the target redundant system.

In NetAcq, a network flow is defined as a continuous, directed stream of packets between two networking infrastructure components (*e.g.*, two servers). A network flow is formated as a tuple of seven attributes $\langle$ $StartTime$, $EndTime$, $SourceIP$, $SourcePort$, $Protocol$, $DestIP$, $DestPort$ $\rangle$ in NetAcq. Our insight on designing NetAcq is: when a client accesses a cloud or web service or its replication, the network traffic flow between them would be open for the entire duration of this service session. In other words, without interferences from other service requests, if we can capture all the networking packets between the start time and end time of accessing target service, we would be able to extract dependency information of this service from the collected packets.

Therefore, based on the above insight, NetAcq plays as a client role to send request to target service. During the whole session, NetAcq uses probing techniques (*e.g.*, snort [12]) to capture all the traffic flows. Although such a way has been enough to collect dependencies in a small-scale service, nevertheless, for typical large-scale service that depends on many components and other services, the probing process would capture a lot of different flows (including noised flows) within the whole service running procedure. For the complex case, therefore, NetAcq first tries to capture all the possible traffic flows among different components. Regarding each captured flow, NetAcq counts and records the time of the flow occurring during the whole service session, and then models these interactions in the form of a directed dependency edge (*i.e.*, $A \rightarrow B$), which represents a item of network dependency information. Two nodes (*i.e.*, A and B) in this directed dependency edge are used to denote components as the origination and destination of this flow respectively. The directed edge (*i.e.*, the arrow) represents a *dependency candidate* between two network components (A and B) with a *weight* indicating the number of times it lasts.

Because there are many noised traffic flows in large-scale service, we need to use an intuitive metric to decide which dependency candidates should be looked as actual dependency items. Thus, inspired by NSDMiner [44], we propose to adopt a logarithm-based metric to determine "real" dependency from candidate pool. In particular, we compute *confidence of a dependency candidate* $A \rightarrow B$ as:

$$C(A \rightarrow B) = \log_{weight(A)} weight(A \rightarrow B) \tag{4.1}$$

Where, $weight(A)$ denotes the number of times node A is accessed. If a confidence is higher than 50%, then we believe this is a real dependency between two devices, and NetAcq stores such an item in dependency information database.

Two types of flows, TCP and UDP flows, are mainly considered by NetAcq. In particular, the former represents the traffic of a TCP session and can be identified by locating the 3-way handshake and 4-way handshake or reset that signal the beginning and the end of the session, and the latter one consists of a continuous stream of UDP packets where the delay between two consecutive packets is no greater than a certain threshold.

By finishing the process of acquiring all the network dependency information, NetAcq stores the collected information in the format as shown in Table 4.1.

Failure probability acquisition. In order to achieve the goal of determining the likelihood of RG failures, dependency collectors can be extended to obtain failure probability. Any communication devices, for instance, have mean time to failure and mean time between failures (MTTF) and many hardware components have warranties during which the expected likelihood of failure is low. Using this information combined with the use time of a device, which may be accessible via firmware, a collector could estimate the likelihood of a failure. To further enhance these types of predictions, we can make use of an online crowd-sourced database in which individuals post failure (or non-failure) information regarding various infrastructure components. In that fashion, if a particular hard drive model has been found faulty, the likelihood of failure would be far greater than computing some probability based upon the MTTF and the current use time.

Failure probability measured in this context tends to be based upon purely mechanical objects. For each component, an administrator can extract how much time the component was off-line, τ over a given time period T and obtain the failure probability by $F = \tau/T$. Gill et al. [35] have successfully obtained the failure probability of each of component failure events within an enterprise network of Microsoft based on this approach.

Chapter 5

Structural Independence Auditing

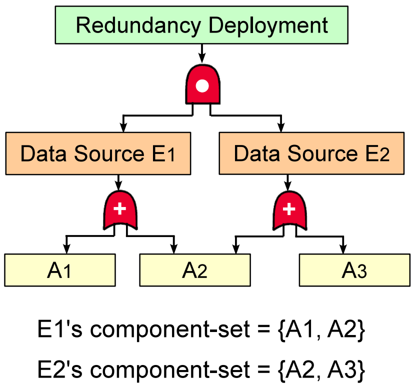
After dependency data acquisition phase, INDaaS performs independence auditing to generate auditing reports.

As described in §3, INDaaS supports two scenarios. We first present a *structural independence auditing* protocol in this chapter, which assumes data sources are willing to provide the auditing agent with the full dependency data, *e.g.*, for auditing a common cloud provider. The auditing agent executes SIA protocol to generate the dependency graph (§5.2), determine the RGs (§5.3), rank the RGs (§5.4), and eventually generates an auditing report (§5.5).

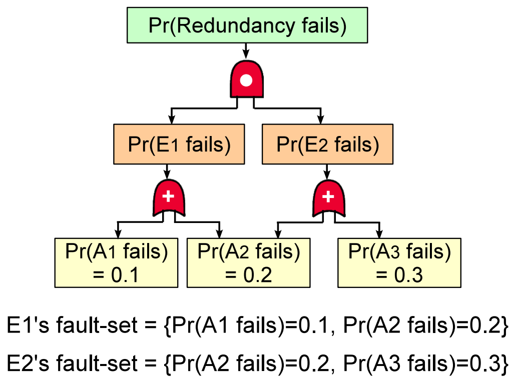
In the next chapter, we present a *private independence auditing* protocol to support auditing across multiple cloud providers unwilling to reveal the full dependency data to anyone.

5.1 Dependency Graph

To implement the most general structural independence auditing, the auditing agent first generates an explicit dependency graph representation, which will later be used by the pluggable auditing modules. In designing this representation, we adapt tradi-

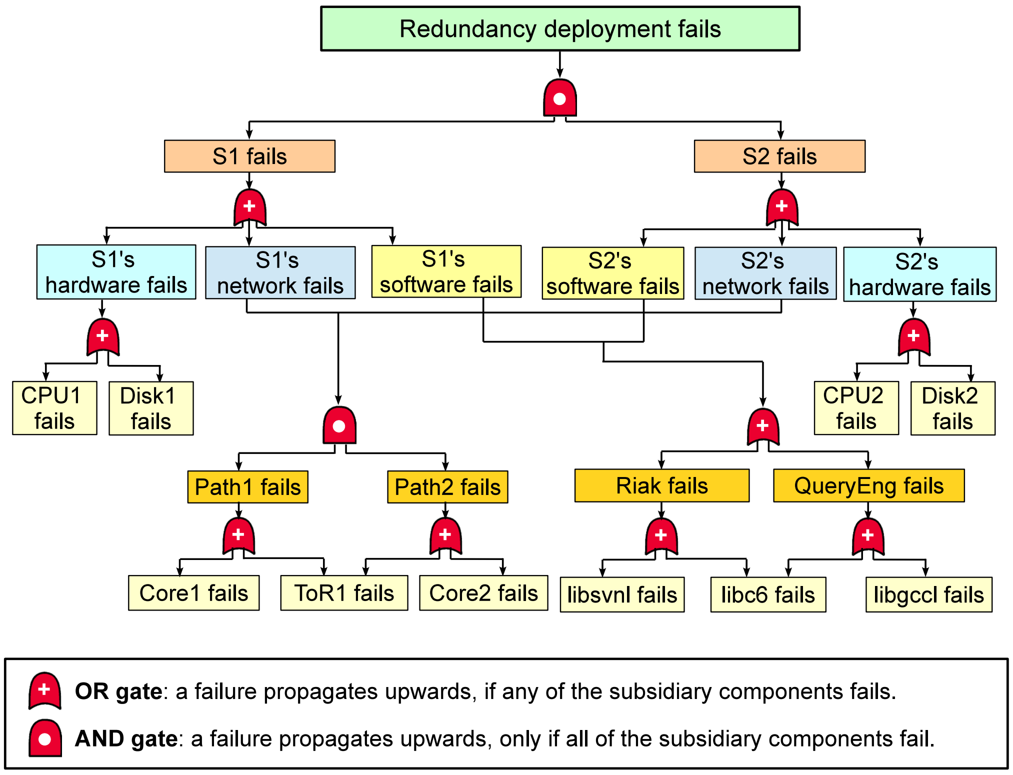


(a) Component-set level of detail.



Pr(Redundancy fails)
Pr(E1 fails)
Pr(E2 fails)
Pr(A1 fails) = 0.1
Pr(A2 fails) = 0.2
Pr(A3 fails) = 0.3
E1's fault-set = {Pr(A1 fails)=0.1, Pr(A2 fails)=0.2}
E2's fault-set = {Pr(A2 fails)=0.2, Pr(A3 fails)=0.3}

(b) Fault-set level of detail.



(c) Fault graph level of detail.

Figure 5.1: Dependency graphs represented at three different levels of detail.

tional fault tree models [66, 74] to a directed acyclic graph structure, and generalize the representation to express dependencies at any of three different levels of detail: *component-set*, *fault-set* and *fault graph*.

Component-set. At the most basic level of detail, we organize dependencies in terms of component-sets. As shown in Figure 5.1a, if a system E_1 depends on components A_1 and A_2, and another system E_2 depends on components A_2 and A_3, then the two relevant component-sets are $\{A_1, A_2\}$ and $\{A_2, A_3\}$, respectively. E_1 and E_2 are the data sources. At this level of detail, for independence reasoning, we focus only on the presence of shared components – *e.g.*, A_2 – that may lead to correlated failures.

As Figure 5.1a illustrates, we express component-sets in a two-level "*AND*-of-*OR*s" dependency graph. This structure consists of two types of nodes: *components* and *logic gates*. If a component fails (or not), it outputs a 1 (or 0) to its higher-layer logic gate. The two types of logic gates, *AND* and *OR*, depict the different logical relationships among components' failures. For an *OR* gate, if any of its subsidiary components fails, this failure propagates upwards. For an *AND* gate, only if *all* of its subsidiary components fail, the gate propagates a failure upwards. The top-level *AND* gate thus represents redundancy across the data sources (*e.g.*, E_1 and E_2), each of which uses an *OR* gate to connect all its dependent components. Our representation also supports n-of-m redundant deployments ($n \leq m$) via n-of-m *AND* gates.

Fault-set. At the fault-set level of detail, we additionally assign some form of *weight* to each component, *e.g.*, probability of failure over some time period. As shown in Figure 5.1b, the failure of A_1 or A_2 leads to the outage of system E_1; thus, the two failure events $\{A_1$ fails, A_2 fails$\}$ form a fault-set. Hereafter, when reasoning at the fault-set level, we assign each failure event a failure probability between 0 and 1. Approaches to obtaining realistic failure probabilities are discussed later in §7.1.

Fault graph. The component-set and fault-set levels of detail assume a single level of redundancy across data sources (*e.g.*, E_1 and E_2), each depending on a "flat" set of components among which any failure causes the respective data source to fail. The fault graph, the richest level of detail INDaaS supports, can describe more complex dependency structures as shown in Figure 5.1c. In a fault graph, event nodes having no child nodes are called *basic events*, the root node is called the *top event*, and the remaining nodes are *intermediate events*. Each node in a fault graph has a weight expressing the failure probability of the associated event. A fault graph is evaluated from basic events to the top event. Each top and intermediate event has an *input gate* connecting the lower-layer events. For example, in Figure 5.1c, the top event's input gate is an *AND* gate representing top-level redundancy, but the fault graph also expresses internal redundancy via the internal *AND* gates at lower levels.

Transformation between different details. Any fault-set model can be trivially transformed into a fault graph simply by enumerating all the members of a given subsystem's component-set under an OR gate, and enumerating all the subsystems whose independence is to be analyzed under a top-layer AND gate. Any fault graph can be transformed into a fault-set level of detail that "conservatively approximates" the fault graph, simply by converting all AND nodes other than the one for the top event into OR, and combining adjacent OR, resulting in a simple 2-layer (AND-of-ORs).

Converting an AND into an OR is a conservative approximation in that doing so may increase the estimated failure probability of the overall system (top event) and hence overestimate the probability of failure, but the transformation can never decrease or underestimate the ultimate probability of failure. This transformation basically amounts to ignoring or throwing away information about the failure-suppressing effects of any deep, internal levels of redundancy.

5.2 Generating Dependency Graph

Any dependency graph, at whichever level of detail, in principle represents the underlying structure of a top-level service across a number of redundant systems. Each such system is a data source where the auditing agent can obtain the dependency data. Automatically building a fault graph with the dependency data is non-trivial in practice. We summarize here how the auditing agent builds a dependency graph at the fault graph level of detail from top to bottom.

- **Step 1:** The fault graph's top event is the failure of the entire redundancy deployment R.

- **Step 2:** According to the auditing client's specification (see Step 1 in §3), the auditing agent sends a query to the dependency information database DepDB for information about all servers given in the specification. Each server's failure event then becomes a child node of the top event, and an *AND* gate connects the top event with its child nodes to express the servers' redundancy.

- **Step 3:** The auditing agent then queries DepDB for each server's network, hardware, and software dependencies. As a result, each server's failure event has three child nodes, *i.e.*, network, software, and hardware failure events. An *OR* gate connects the server failure event with its three child nodes, since the failure of any of these dependencies effectively causes the server to fail.

- **Step 4:** For the hardware failure event of each server, the auditing agent gets its dependency data from DepDB, then uses an *OR* gate to connect the hardware failure event with its dependencies' failure events.

- **Step 5:** For each server's network failure event, the auditing agent queries DepDB for network paths relevant to the server, then connects them as child nodes to the server's network failure event. The agent puts an *AND* gate between the network failure event and child nodes representing redundant paths, while network devices comprising each path are connected by an *OR* gate.

- **Step 6:** The auditing agent repeats Step 5 to construct the child nodes for each server's software failure event. Different layers of software components are connected by an *OR* gate, and all packages underlying a software component are connected by an *OR* gate.

As an example, the redundancy deployment in Figure 4.1 may be represented by the fault graph in Figure 5.1c. An information-rich fault graph may be "downgraded" to the lower fault-set or component-set levels of detail, by discarding partial information in a fault graph.

Our INDaaS prototype can also compose individual dependency graphs collected from multiple services into more complex aggregate dependency graphs (*e.g.*, EC2 instances depending on services offered by EBS and ELB). The basic idea is to perform the dependency graph generation multiple times, treating each service deployment as a dependency sub-graph and then recursively linking them together to construct the complete dependency graph.

5.3 Determining Risk Groups

After building a dependency graph, SIA needs to determine risk groups (RGs) of interest in the dependency graph. The SIA provides two pluggable auditing algorithms which make trade-offs between accuracy and efficiency. The *minimal RG* algorithm computes precise results, but its execution time increases exponentially with the size of dependency graph, making it impractical on large datasets. The *failure sampling* algorithm, in contrast, runs much faster but scarifies accuracy. Both algorithms operate on dependency graphs represented at any level of detail. Without loss of generality, hereafter we elaborate on the algorithms at the fault graph level.

Minimal RGs. An RG within a dependency graph is a group of basic failure events with the property that if all of them occur simultaneously, then the top event occurs as well. For example, in Figure 5.1(a), if A_1 and A_3 fail simultaneously, the redundancy deployment fails. Here, $\{A_1, A_3\}$, $\{A_1, A_2\}$, $\{A_1, A_2, A_3\}$, $\{A_2\}$, and $\{A_2, A_3\}$ are five RGs. Some RGs, however, are more critical than others. We define an RG as a *minimal RG* if the removal of any of its constituent failure events makes it no longer an RG. Consider the following two RGs: $\{A_1, A_2\}$ and $\{A_2, A_3\}$ in Figure 5.1(a). Neither are minimal RGs because $\{A_2\}$ alone is sufficient to cause the top event to occur; thus, the minimal RGs should be $\{A_2\}$ and $\{A_1, A_3\}$. As another example, the minimal RGs in Figure 5.1(c) are {ToR1 fails}, {Core1 fails, Core2 fails}, etc.

5.3.1 Minimal RG Algorithm

The first algorithm for determining RGs is adapted from classic fault tree analysis techniques [66, 74]. The pseudocode of this algorithm is presented in Algorithm 1.

Algorithm 1: Minimal RG algorithm

```
Input: fault graph $T$
Output: minimalRG
foreach $event_i \in T$ by reversed breadth-first traversal do
    if $event_i$ is basic event then
        $event_i.RG.append(event_i)$;
    else
        if $event_i.InputGate$ is OR then
            foreach $c_i \in event_i.ChildList$ do
                foreach $cs_i \in c_i.RG$ do
                    $event_i.RG.append(cs_i)$
        else
            /* $event_i.InputGate$ is AND */
            foreach $c_i \in event_i.ChildList$ do
                foreach $c_j \in event_i.ChildList$ and $c_i \neq c_j$ do
                    /* list is the Cartesian product of $c_i.RG$ and $c_j.RG$ */
                    $list \leftarrow c_i.RG \times c_j.RG$;
                    $event_i.RG.append(list)$;
/* reduce redundant items in $root.RG$ and assign the result to
   $minimalRG$, and then simplify minimalRG. */
$minimalRG \leftarrow reduce\_redundancy(root.RG)$;
$minimalRG \leftarrow simplify(minimalRG)$;
return minimalRG;
```

This algorithm traverses a dependency graph G in a reverse breadth-first order (from basic events to the top event). Basic events first generate RGs containing only themselves, while non-basic events produce RGs based on their child events' RGs and their input gates. For a non-basic event, if its input gate is an *OR* gate, the RGs of this event include all its child events' RGs; otherwise, if its input gate is an *AND* gate, each RG of this event is an element of the cartesian product among the RGs of its child events. Traversing the dependency graph G generates all the RGs, and in turn all the minimal RGs through simplification procedures. This algorithm produces precise results, but is NP-hard [73].

5.3.2 Failure Sampling Algorithm

To address the efficiency issue, we developed an RG detection algorithm based on random sampling, which makes a trade-off between accuracy and efficiency. This algorithm uses multiple sampling rounds, each of which performs a breadth-first traversal of the dependency graph G. Within each sampling round, the algorithm assigns either a 1 or a 0 to each basic event of G based on random coin flipping, where 1 represents failure and 0 represents non-failure. Starting from such an assignment, the algorithm assigns 1s and 0s to all non-basic events from bottom to top based on their logic gates and the values of their child events. After each sampling round, the algorithm checks whether the top event fails. If it fails (*i.e.*, its value is 1), then the algorithm generates an RG consisting of all the basic events being assigned a 1 in this sampling round. The algorithm executes a large number of sampling rounds and aggregates the resulting RGs in all rounds. The failure sampling algorithm offers the linear time complexity, but is non-deterministic and cannot guarantee that the resulting RGs it identifies are minimal RGs. We provide provable guarantees on failure sampling algorithm in the next section, which shows the algorithm is capable of finding out minimal RGs with relatively high probability. This failure sampling algorithm is similar in principle to heuristic SAT algorithms such as ApproxCount [80], and these methods may offer ways to improve INDaaS failure sampling.

5.3.3 Failure Sampling Algorithm Analysis

Because failure sampling algorithm is a randomized approach, we are interested in what type of guarantees it can provide. We provide provable guarantees for failure sampling algorithm in this section.

Algorithm 2: Failure sampling algorithm

```
Input: fault graph $G$ and the number of samples $N$
Output: RG
for $i \leftarrow 1$ to $N$ do
    foreach $event_i \in G$ by reversed breadth-first traversal do
        if $event_i$ is basic event then
            $event_i.occurrence \leftarrow$ 0 or 1 based on random flipping a fair coin
        else
            $event_i.occurrence \leftarrow 0$;
            if $event_i.InputGate$ is OR then
                foreach $c_i \in event_i.ChildList$ do
                    if $c_i.occurrence$ is 1 then
                        $event_i.occurrence \leftarrow 1$;
                        break;
            else                                        /* $event_i.InputGate$ is AND */
                foreach $c_i \in event_i.ChildList$ do
                    if $c_i.occurrence$ is 0 then
                        break;
                $event_i.occurrence \leftarrow 1$;
    if $root.occurrence$ is 1 then
        $TmpSet \leftarrow \emptyset$;
        foreach $event_i \in G$ do
            if $event_i.occurrence$ is 1 then
                $TmpSet.append(event_i)$;
        RG.append(TmpSet);
return RG;
```

5.3.3.1 Preliminaries

Let $f : \{0,1\}^k \to \{0,1\}$ be a k-ary Boolean function. An *assignment* is a vector $\vec{a} \in \{0,1\}^k$. A *target* is a set of assignments $T \subseteq \{0,1\}^k$. Assignment $\vec{a}$ is a *satisfying* assignment for f if and only if $f(\vec{a}) = 1$. The *size* of a satisfying assignment $\vec{a}$ is the number of 1's in $\vec{a}$. A satisfying assignment is *minimal* if it has minimum size over all satisfying assignment.

For example, Maj_5 is a Boolean function that maps any assignments with at least three 1's to 1 and others to 0, and $\min_{maj}$ is a target that consists of all assignments with minimum number of 1's that evaluate to 1, i.e., those with exactly three 1's. Then any assignment is the target $\min_{maj}$ is a minimal satisfying assignment.

5.3.3.2 Hardness of Finding A Minimal Satisfying Assignment

A Boolean function is *simple* if it consists of only *AND* and *OR* operators. We show that there is no efficient algorithm that computes a minimal satisfying assignment for a simple Boolean function unless $\mathcal{P} = \mathcal{NP}$. The idea is by reduction from the set cover problem.

An instance σ of the set cover problem (SCP) consists an universe $U = \{1, 2, \cdots, m\}$, and n subsets $S = \{s_1, \cdots, s_n\}$, such that $s_i \subseteq U$ for any $1 \leq i \leq n$ and $\bigcup_{1 \leq i \leq n} s_i = U$. A *cover* of U is a subset of S such that their union equals U. The set cover problem requires to find a cover with minimum size, which is known to be $\mathcal{NP}$-hard.

Theorem 1 *Finding a minimal satisfying assignment for a simple Boolean function is $\mathcal{NP}$-hard.*

Proof 1 *Given an instance σ of SCP, we construct an instance ϕ of the minimal satisfying assignment problem (MSA). We first describe the construction of a simple Boolean function in the conjunctive normal form (CNF). A Boolean function is CNF if it is a conjunction (*AND*) of clauses, where a clause is a disjunction (*OR*) of literals. The Boolean function has m clauses, $C_1, \cdots, C_m$, corresponding to the m elements in U, and n variables, $x_1, \cdots, x_n$, corresponding to the n subsets in S. For each element $j \in s_i$, add variable x_i to clause C_j. Apparently, this construction $g(\sigma) = \phi$ can be computed in polynomial time.*

Now we show that there is one-to-one correspondence between covers in SCP and satisfying assignments in MSA. In one direction, given a cover in SCP, setting

TRUE *all variables x_i corresponding to subsets s_i in the cover results in a satisfying assignment in MSA. In the other direction, given a satisfying assignment in MSA, including all subsets s_i corresponding to* TRUE *variables x_i in the satisfying assignment results in a cover in SCP. In addition, the size of the cover equals the size of the satisfying assignment in both directions. A direct consequence of this observation is that finding a cover with minimum size in σ is then equivalent to finding a satisfying assignment with minimum size in ϕ. As SCP is $\mathcal{NP}$-hard, the MSA problem is also $\mathcal{NP}$-hard.*

5.3.3.3 Target Cover

Uniform sampling. Consider the following random process. Given a k-ary Boolean function f and a target T, we randomly sample assignments. For each trial of sampling, flip a sequence of k independent *fair* coins. Let random variable X be the number of samples when all assignments in T are covered by the random sampling process. Then we want to bound the following two problems: (1) What is the expected number of samples in order to cover the target? (2) What is the probability of covering the target if the number of samples is m?

Lemma 2 *The expected number of uniform samples to cover the target is* $\mathbf{E}[X] = 2^k H_t = \Theta(2^k \log t)$, *where* $t = |T|$ *and* $H_n = \sum_{i=1}^{n}(1/i)$ *is the harmonic number.*

Proof 2 *The probability for a random sample to cover any assignment in T is $t/2^k$. After i assignments in T has been covered, a random sample to cover an additional assignment in T is $(t-i)/2^k$. Let random variable X_i be the number of samples used to hit the i-th assignment in T, then the subsequence of random sampling process to cover the i-th assignment are Bernoulli trials with success probability $(t-i+1)/2^k$.*

Therefore, for any $1 \leq i \leq d$,

$$\mathbf{E}[X_i] = \frac{2^k}{t-i+1}$$

Thus, by the linearity of expectation, we have

$$\begin{aligned}\mathbf{E}[X] &= \mathbf{E}\left[\sum_{i=1}^{t} X_i\right] = \sum_{i=1}^{t} \mathbf{E}[X_i] \\ &= \sum_{i=1}^{t} \frac{2^k}{t-i+1} = 2^k H_t\end{aligned}$$

Then $\mathbf{E}[X] = \Theta(2^k \log t)$ *follows from the fact that* $H_t = \Theta(\log t)$.

Remark: we can regard the target cover problem as a variant of the coupon collector's problem, where t specific coupons out of 2^k possible ones need to be collected.

Then we bound the probability of covering the target with m samples.

Lemma 3 *The probability to cover the target with* m *uniform samples is at least* $1 - m/(2^k H_t)$.

Proof 3 *By Markov's inequality, the probability that more than* m *uniform samples are need to cover the target is*

$$\Pr(X \geq m) \leq \frac{\mathbf{E}[X]}{m}$$

Therefore, following Lemma 2, the probability to cover the target with m *samples is*

$$\Pr(X \leq m) = 1 - \Pr(X \geq m) \geq 1 - \frac{2^k H_t}{m}$$

Take the Boolean function Maj_5 and the target $\min_{maj}$ as an example. The expected number of uniform samples needed to cover all $\binom{5}{3} = 10$ target assignments is $2^5 \cdot H_{10} \approx 94$. The probability to cover the target with 188 uniform samples is at least 0.5.

Biased sampling. Consider the following random process with biased sampling. For each trial of sampling, instead of flipping fair coins, we flip a sequence of k independent *biased* coins, such that each assignment $\vec{a}_i$ is cover with probability p_i and $\sum_{i:\vec{a}_i \in \{0,1\}^k} p_i = 1$.

Without loss of generality, let $T = \{a_1, a_2, \cdots, a_t\}$ and $p_1 \leq p_2 \cdots \leq p_t$. Let $(q_1, q_2, \cdots, q_t)$ be the sequence of prefix sums for the sequence $(p_1, p_2, \cdots, p_t)$, i.e., for any $1 \leq i \leq d$,

$$q_i = \sum_{j=1}^{i} p_j$$

Let $(q'_1, q'_2, \cdots, q'_t)$ be the sequence of prefix sums for the sequence $(p_t, p_{t-1}, \cdots, p_1)$[1], i.e., for any $1 \leq i \leq t$,

$$q'_i = \sum_{j=t-i+1}^{t} p_j$$

Then by definition, it follows that $q_i \leq q'_i$ for any $1 \leq i \leq t$. We will bound the expected number of biased samples needed to cover the target with q_i and q'_i.

Lemma 4 *The expected number of biased samples to cover the target is* $\sum_{i=1}^{t}(1/q'_i) \leq \mathbf{E}[X] \leq \sum_{i=1}^{t}(1/q_i)$*, where* $t = |T|$.

Proof 4 *The analysis is similar to that for Lemma 2, with the only distinction that the success probability for each subsequence of Bernoulli trials depends on* p_i *now. Let* π *be a permutation of the sequence* $\{1, 2, \cdots, t\}$*, representing the order in which assignments in target* T *are covered in the sampling process. Let the sequence* $(\hat{q}_1, \cdots, \hat{q}_t)$ *be the prefix sums of* $(p_{\pi(1)}, p_{\pi(2)}, \cdots, p_{\pi(t)})$*, i.e.,* $\hat{q}_i = \sum_{j=\pi(1)}^{\pi(i)} p_i$ *for all*

[1]The sequence of q'_i can also be regarded as suffix sums for the sequence $(p_1, p_2, \cdots, p_t)$, although this definition is not standard.

$1 \leq i \leq t$. Let $p = \sum_{i=1}^{t} p_i$. It follows the definition that for any $1 \leq i \leq t$,

$$q_i \leq \hat{q}_i \leq q'_i \tag{5.1}$$

Let random variable X_i be the number of samples used to cover assignment $\vec{a}_{\pi(i)}$. Then the subsequence of random sampling process to cover $\vec{a}_{\pi(i)}$ are Bernoulli trials with success probability $\sum_{j=\pi(i)}^{\pi(t)} p_j = p - \hat{q}_{i-1}$, where $\hat{q}_0 = 0$ by convention. Therefore, for any $1 \leq i \leq t$,

$$\mathbf{E}[X_i] = \frac{1}{p - \hat{q}_{i-1}}$$

Thus, by the linearity of expectation, we have

$$\begin{aligned} \mathbf{E}[X] &= \mathbf{E}\left[\sum_{i=1}^{t} X_i\right] = \sum_{i=1}^{t} \mathbf{E}[X_i] \\ &= \sum_{i=1}^{t} \frac{1}{p - \hat{q}_{i-1}} \end{aligned} \tag{5.2}$$

Define $q_0 = q'_0 = 0$ for convention. Combining (5.1) and (5.2) gives

$$\sum_{i=1}^{t} \frac{1}{p - q_{i-1}} \leq \mathbf{E}[X] \leq \sum_{i=1}^{t} \frac{1}{p - q'_{i-1}} \tag{5.3}$$

Note that by definition, for all $1 \leq i \leq d$,

$$p - q_{i-1} = q'_{t-i+1} \tag{5.4}$$

Finally, combining (5.3) and (5.4) gives

$$\sum_{i=1}^{t} \frac{1}{q'_i} \leq \mathbf{E}[X] \leq \sum_{i=1}^{t} \frac{1}{q_i}$$

Take the Boolean function Maj_5 and the target $\min_{maj}$ as an example. Assume the ten target assignments have probability $p_1 = \cdots = p_5 = 1/16$ and $p_6 = \cdots = p_{10} = 1/8$. Then the expected number of biased samples needed to cover all target assignments is bounded by $24.49 \leq \mathbf{E}[X] \leq 44.35$.

5.3.3.4 (d, t)-Target Cover

In some applications, covering the entire target is too expensive. In such cases, it might be desirable to cover at least d out of all t target assignments. We call this problem the (d, t)-target cover problem.

Lemma 5 *The expected number of uniform samples to cover at least d members in the target is $\mathbf{E}[X] = 2^k(H_t - H(t-d)) = \Theta(2^k \log \frac{t}{t-d})$, where $t = |T|$ and $H_n = \sum_{i=1}^{n}(1/i)$ is the harmonic number.*

Proof 5 *The proof is similar to that of Lemma 2, with the distinction that the counting stops when we cover the d-th assignment in T. Let random variable X_i be the number of samples used to cover the i-th assignment in T. For any $1 \leq i \leq d$,*

$$\mathbf{E}[X_i] = \frac{2^k}{t - i + 1}$$

Therefore, we have

$$\begin{aligned}
\mathbf{E}[X] &= \mathbf{E}\left[\sum_{i=1}^{d} X_i\right] = \sum_{i=1}^{d} \mathbf{E}[X_i] \\
&= \sum_{i=1}^{d} \frac{2^k}{t-i+1} = \sum_{i=t-d+1}^{t} \frac{2^k}{i} \\
&= 2^k \left(\sum_{i=1}^{t} \frac{1}{i} - \sum_{i=1}^{t-d} \frac{1}{i} \right) \\
&= 2^k(H_t - H_{t-d}) = \Theta\left(2^k \log \frac{t}{t-d}\right)
\end{aligned}$$

The result in Lemma 5 indicates that when t is a significant fraction of 2^k or d is a small fraction of t, the expected number of samples for (d, t)-target cover is not overwhelming. Take the Boolean function Maj_5 and the target $\min_{maj}$ as an example. The expected number of uniform samples to cover at least 3 out of the 10 target assignments is $2^5 \cdot (H_{10} - H_7) \approx 10.76$.

5.4 Ranking Risk Groups

After determining RGs, we have two algorithms to rank them and generate the RG-ranking list.

Size-based ranking. To rank RGs at the component-set level or at the unweighted fault graph level, we use a simple size-based ranking algorithm which ranks RGs based on the number of components in each RG. While this algorithm cannot distinguish which potential component failures may be more or less likely, identifying RGs with fewer components – especially any of size 1 indicating no redundancy – can point to areas of the system that may warrant closer manual inspection. For example, in Figure 5.1(c), the RGs {ToR1} and {libc6} are ranked highest since they have the least size.

Failure probability ranking. In cases where the probabilities of failure events can be estimated, we provide a probability-based ranking algorithm to evaluate RGs at the levels of fault-set and weighted fault graph. This algorithm can rank RGs by their relative importance. For a given RG's failure event (say, C), its relative importance, I_C, can be computed based on Equation 5.5.

$$I_C = \frac{\Pr(C)}{\Pr(T)} \tag{5.5}$$

Where, T is the top event. Thus, the relative importance I_C is actually computed using the probability of C, $\Pr(C)$, in comparison to the probability of the top event T, $\Pr(T)$ Specifically, $\Pr(C)$ is the probability that all the events in C occur simultaneously, and $\Pr(T)$ is computed by the inclusion-exclusion principle where the involved sets are all the minimal RGs of T. Equation 5.6 shows how to compute the probability of the top event T.

$$\begin{aligned}\Pr(T) = &\sum_{i=1}^{n} \Pr(MC_i) - \sum_{i<j=2}^{n} \Pr(MC_i \cdot MC_j) \\ &+ \sum_{i<j<k=2}^{n} \Pr(MC_i \cdot MC_j \cdot MC_k) \\ &+ \cdots + (-1)^{n-1} \Pr(MC_1 \cdot MC_2 \cdots MC_n)\end{aligned} \tag{5.6}$$

Where, MC_i means minimal RG i. For example in Figure 5.1(b), since the probabilities of events A_1, A_2 and A_3 are 0.1, 0.2 and 0.3, respectively, we have: $\Pr(T) = 0.1 \cdot 0.3 + 0.2 - 0.1 \cdot 0.3 \cdot 0.2 = 0.224$. Therefore, the relative importances of the minimal RGs $\{A_2$ fails$\}$ and $\{A_1$ fails, A_3 fails$\}$ are: $0.2/0.224 = 0.8929$ and $0.03/0.224 = 0.1339$, respectively. As a result, $\{A_2$ fails$\}$ is ranked higher than $\{A_1$ fails, A_3 fails$\}$.

5.5 Generating the Auditing Report

Upon getting the RG-ranking lists for all redundancy deployments, SIA computes an independence score for each of them. If the size-based ranking algorithm is used, a given redundancy deployment R's independence score is computed as $indep(R) = \sum_{i=1}^{n} size(c_i)$, where c_i denotes the ith RG in the R's RG-ranking list, and n denotes the number of top RGs in the RG-ranking list used for this independence evaluation. If the failure probability based ranking algorithm is used, a given redundancy deployment R's independence score is then $indep(R) = \sum_{i=1}^{n} I_{c_i}$, where I_{c_i} denotes the relative importance of c_i.

The auditing agent generates an auditing report by ranking all the redundancy deployments based on their independence scores, and finally sends the report back to the auditing client for reference. With the auditing report, the auditing client might for example select the most independent redundancy deployment for her service.

The auditing report can also help an auditing client understand unexpected common dependencies to focus further analysis. In the case of one documented Amazon EC2 outage, for example [4], we speculate that the availability of an INDaaS auditing report might have enabled the operators to notice that a specific EBS server had become a common dependency, and fix it, thus avoiding the outage.

Chapter 6

Private Independence Auditing

We now address the challenge of independence auditing across mutually distrustful data sources, *e.g.*, multiple cloud providers, who may be unwilling to share dependency data with each other or any third-party auditor. To reflect the motivating deployment model, we use the term *cloud providers* instead of data sources when describing the private independence auditing (PIA) protocol.

The most general and direct private auditing approach, explored by Xiao *et al.* [82], is to use secure multi-party computation (SMPC) [85] to compute and reveal overlap among the datasets of multiple cloud providers while keeping the data themselves private. This approach works in theory, but scales poorly in practice due to its inherent complexity. We find SMPC to be impractical currently even for datasets with only a few hundreds of components.

We thus focus henceforth on a more scalable approach built on private set intersection cardinality techniques [34, 51, 72, 86]. This approach sacrifices generality and dependency graph expressiveness, operating only at the component-set level of detail. The basic idea is to evaluate Jaccard similarity [45] using a private set intersection cardinality protocol [72] to quantify the independence of redundancy configurations.

To support the fault-set level of detail, our PIA protocol uses duplicate set elements to implement weighted private set intersection. Although our design is more restricted in that it can only reason at "coarse-grained" level of detail rather than arbitrary one, it is efficient to implement practically at larger scales.

This chapter starts with the trust assumptions of each entity within PIA working process (§6.1) and needed technical building blocks (§6.2). Then, we present PIA design according to the following phases: generating dependency graph (§6.3), privately auditing independence (§6.4), and generating auditing report (§6.5).

6.1 Trust Assumptions

As described in §3, our architecture consists of entities filling three roles: auditing client, cloud providers (*i.e.*, data sources in Figure 3.1), and auditing agent.

We assume that auditing clients are potentially malicious and wish to learn as much as possible about the cloud providers' private dependency data. We assume cloud providers and the auditing agent are honest but curious: they run the specified PIA protocol faithfully but may try to learn additional information in doing so. We assume there is no collusion among cloud providers and the auditing agent.

Dishonest behavior includes producing a partial set for dependency information in an attempt to discover the dependency components of another cloud provider. We wish to emphasize that we believe that our honest-but-curious trust assumption on both cloud providers and the auditing agent is appropriate in the realistic scenario. Previous efforts, *e.g.*, PSI-CA [34], Kissner and Song [51], and DJoin [58] also assume parties in their protocols are honest but curious. We discuss some potential solutions to dealing with dishonest parties in §**??**. Finally, we assume that all the cryptographic operations are operated correctly and computationally secure.

6.2 Technical Building Blocks

There are four technical building blocks that we utilize throughout the PIA design.

6.2.1 Jaccard Similarity

Jaccard similarity [45] is a widely-adopted metric for measuring similarity across multiple datasets. Jaccard similarity is defined as

$$J(S_0, \cdots, S_{k-1}) = \frac{|S_0 \cap \cdots \cap S_{k-1}|}{|S_0 \cup \cdots \cup S_{k-1}|} \tag{6.1}$$

Where S_i denotes the ith dataset. A Jaccard similarity J close to 1 indicates high similarity, whereas a J close to 0 indicates the datasets are almost disjoint. In practice, datasets with similarity $J \geq 0.75$ are considered significantly correlated [75]. While there are many other similarity metrics, *e.g.*, the Sørensen-Dice index [71], we choose Jaccard similarity because it is efficient, easy to understand, and extends readily to more than two datasets.

6.2.2 MinHash Technique

Computing the Jaccard similarity incurs a complexity linear with the dataset sizes. In the context of a large number of big sets, its computation might be a relatively expensive. In the presence of large datasets, an approximation of the Jaccard similarity based on MinHash is often preferred [24, 25]. The MinHash technique [27] extracts a vector $\{h_{min}^{(i)}(S)\}_{i=1}^{m}$ of a dataset S through deterministic sampling, where $h^{(1)}(\cdot), \cdots, h^{(m)}(\cdot)$ denote m different hash functions, and $h_{min}^{(i)}(S)$ denotes the item $e \in S$ with the minimum value $h^{(i)}(e)$. Let δ denote the number of datasets satisfying $h_{min}^{(i)}(S_0) = \cdots = h_{min}^{(i)}(S_{k-1})$. Then, the Jaccard similarity $J(S_0, \cdots, S_{k-1})$ can be approximated as δ/m. Here, the parameter m correlates to the expected error to the precise Jaccard similarity — a larger m (*i.e.*, more hash functions) yields a smaller approximation error. Broder [27] proves that the expected error of MinHash-based Jaccard similarity estimation is $O(1/\sqrt{m})$.

6.2.3 Commutative Encryption

In a commutative encryption system, for any given plaintext message M, multiple parties encrypting M in any order always produce the same ciphertext. Suppose E and D are commutative encryption and decryption functions, respectively. K and J are public keys (or, private keys where applicable) used by different parties. According to the definition given In a commutative encryption system, for any given plaintext message M, multiple parties encrypting M in any order always produce the same ciphertext. Suppose E and D are commutative encryption and decryption functions, respectively. K and J are public keys (or, private keys where applicable) used by different parties. According to the definition given by Shamir *et al.* [70], a commutative encryption system should have the following five properties:

- $E_K(M)$ is the ciphertext of a given message M under the key K.
- $D_K(E_K(M)) = M$ for all messages M and keys K.
- $E_K(E_J(M)) = E_J(E_K(M))$ for all messages M as well as keys J and K.
- Given M and $E_K(M)$, it is computationally impossible for a cryptanalyst to derive K for all M and K.
- Given any messages M_1 and M_2, it is computationally impossible to find keys J and K such that $E_J(M_1) = E_K(M_2)$.

Typical commutative encryption systems employ encryption function $E_K(M) \equiv M^K (\text{mod } p)$, where p is a large and secure composite number shared by all parties, and $\gcd(K, \varphi(p)) = 1$. Here, $\varphi(p)$ is Euler's totient function, which can be easily computed from the prime factorization of p. For two-party case, we can call an encryption function as commutative if $E_K(E_J(M)) \equiv M^{JK} (\text{mod } p) \equiv M^{KJ} (\text{mod } p)$ $\equiv E_J(E_K(M))$. More detailed explanations about commutative encryption can be found in Pohlig-Hellman (PH) [63] and commutative RSA [70].

6.2.4 Private Set Intersection Cardinality

A private set intersection cardinality protocol allows a group of $k \geq 2$ parties, each with a local dataset S_i, to compute the number of overlapping elements among them privately without learning any elements in other parties' datasets. We adopt P-SOP, a private set intersection cardinality protocol based on commutative encryption. In P-SOP, all parties form a logical ring, and agree on the same deterministic hash function (*e.g.*, SHA-1 or MD5). In addition, each party has its own permutation function used to shuffle the elements in its local dataset, as well as its own public/private key pair used for commutative encryption [63, 70]. Commutative encryption offers the property that $E_K(E_J(M)) = E_J(E_K(M))$ where E_X denotes using X's public key to encrypt the message M.

In P-SOP, each party first makes every element in its own dataset S_i identical. Specifically, any element e appearing t times in S_i is represented as t unique elements $\{e\|1, \cdots, e\|t\}$, with '$\|$' being a concatenation operator. Each party then hashes and encrypts every individual element in its dataset, and randomly permutes all the encrypted elements. Afterwards, each party sends the encrypted and permuted dataset to its successor in the ring. Next, once the successor receives the dataset, it simply encrypts each individual element in the received dataset, permutes them, and sends the resulting dataset to its successor. The process repeats until each party receives its own dataset whose individual elements have been encrypted and permuted by all parties in the ring. Finally, all parties share their respective encrypted and permuted datasets, so that they can count the number of common elements in these datasets, *i.e.*, $|\cap_i S_i|$, as well as the number of unique elements in these datasets $|\cup_i S_i|$ due to the commutative property.

We now use a simple but illustrative example to clarify P-SOP protocol. Suppose there are three providers Alice, Bob and Carlo. They have data sets S_A, S_B and S_C, respectively. For clarity, we define the following symbols:

- $\widehat{H}(S)$: generate a hash value for *each* individual item in the data set S using an agreed-upon hash function H.
- $\widehat{E_i}(S)$: encrypt *each* individual item in the data set S under the i-th provider p_i's commutative public key.
- $P_i(S)$: permute *all* items in the data set S using the i-th provider p_i's permutation function.

In order to obtain the set intersection and union cardinality, the three parties perform the following steps:

- **Step 1:** Alice hashes and encrypts each individual item in her data set S_A, and then randomly permutes all the encrypted items to obtain $P_A(\widehat{E_A}(\widehat{H}(S_A)))$. Afterwards, Alice sends the encrypted and permuted data set to Bob.
- **Step 2:** Once receiving the data set, Bob simply encrypts each individual item in the received data set, permutes all the double-encrypted items, and then sends the resulting data set $P_B(\widehat{E_B}(P_A(\widehat{E_A}(\widehat{H}(S_A)))))$ to Carlo.

- **Step 3:** Once receiving the data set, Carlo encrypts each individual item in the received data set, permutes all the (triple-)encrypted items, and then sends the resulting data set $P_C(\widehat{E_C}(P_B(\widehat{E_B}(P_A(\widehat{E_A}(\widehat{H}(S_A)))))))$ back to Alice.

- **Step 4:** The above three steps also apply to Bob and Carlo's data sets S_B and S_C, so that they eventually receive the processed data sets $P_A(\widehat{E_A}(P_C(\widehat{E_C}(P_B(\widehat{E_B}(\widehat{H}(S_B)))))))$ and $P_B(\widehat{E_B}(P_A(\widehat{E_A}(P_C(\widehat{E_C}(\widehat{H}(S_C)))))))$, respectively.

- **Step 5:** Alice, Bob and Carlo share their respective data sets whose individual items have been encrypted and permuted by all parties. Finally, they count the number of common (or total) unique items in these data sets to learn the set intersection (or union) cardinality.

6.3 Generating Dependency Graph

To perform private independence auditing, each cloud provider p_i (holding an individual data source) within a given redundancy deployment R first generates its local dependency graph at the component-set level. In addition, each p_i needs to *normalize* its generated component-set. This normalization ensures that the same component shared across different cloud providers always has the same identifier.

Common sources of correlated failures across cloud providers are third-party components such as routers and software packages [32]. Therefore, our current PIA prototype normalizes two types of components: 1) third-party routing elements (*e.g.*, ISP routers), and 2) third-party software packages (*e.g.*, the widely-used OpenSSL toolkit). PIA normalizes these components as follows: 1) for routers, PIA uses their accessible IP addresses as unique identifiers, and 2) for software packages, PIA uses their standard names plus version numbers as unique identifiers. In so doing, any given component in all cloud providers' generated component-sets has a unique normalized identifier.

6.4 Auditing Independence Privately

If cloud providers involved in a potential redundancy deployment have relatively small component-sets, PIA takes these (normalized) component-sets S_i directly as input to the private set intersection cardinality protocol (P-SOP) to compute the number of common components $|\cap_i S_i|$ and the number of unique components $|\cup_i S_i|$ across cloud providers. With the two numbers, PIA can compute the Jaccard similarity as $|\cap_i S_i|/|\cup_i S_i|$ to evaluate the independence of this redundancy deployment.

Otherwise, if cloud providers in a potential redundancy deployment have large component-sets, PIA uses m hash functions based on the MinHash technique to map each such component-set to a much smaller dataset S_i, and then takes these MinHash-generated datasets as input to the P-SOP as normal to get the number of common components across cloud providers, *i.e.*, $|\cap_i S_i|$. As discussed in §6.2, the Jaccard similarity can then be approximated as $|\cap_i S_i|/m$. This MinHash-based approach leads to much higher efficiency but lower accuracy. To increase the accuracy, we can use more hash functions in MinHash. How to make the trade-off between efficiency and accuracy depends on the application domain. Algorithm 3 presents the protocol in the form of pseudocode.

6.5 Generating the Auditing Report

In the design as so far, each cloud provider p_i has computed the Jaccard similarities (or estimated Jaccard similarities using MinHash) corresponding to all the redun-

Algorithm 3: Private independence auditing (PIA) process

Input: There are $k \geq 2$ cloud providers, each with a local standardization data set defined as D_i, which contains all the dependency information needed by PIA process. All providers form a logical ring, and agree upon the same deterministic hash function (*e.g.*, SHA-1), and m hash functions (for MinHash). Each provider i has its own public key used for commutative encryption, and its own permutation function used to shuffle the items in its local data set.

Steps:

1. Each provider p_i $(i = 0, ..., k-1)$
   - computes $S_i = \{\langle h_{min}^{(j)}(D_i), j\rangle\}_{j=1}^{m}$ using the m hash functions, if MinHash option is on;
     generates $S_i = D_i$, if MinHash option is off.
   - makes elements in S_i unique. Any element $a \in S_i$ appearing b times in S_i is encoded as $\{a \parallel 1, ..., a \parallel b\}$, with element included only once.
2. Each provider p_i $(i = 0, ..., k-1)$
   - generates a hash value for each individual item in its local data set S_i, obtaining $\widehat{H}(S_i)$;
   - encrypts each individual hash value in $\widehat{H}(S_i)$ under its commutative public key, obtaining $\widehat{E_i}(\widehat{H}(S_i))$;
   - randomly permutes all the encrypted items in $\widehat{E_i}(\widehat{H}(S_i))$, obtaining $P_i(\widehat{E_i}(\widehat{H}(S_i)))$;
   - sends $P_i(\widehat{E_i}(\widehat{H}(S_i)))$ to the successor in the ring, *i.e.*, $p_{(i+1) \bmod k}$, who has not yet encrypted and permuted the data set.
3. Each provider $p_{(i+1) \bmod k}$, $(i = 0, ..., k-1)$
   - receives the data set $P_i(\widehat{E_i}(\widehat{H}(S_i)))$ from its predecessor p_i;
   - encrypts each individual item in $P_i(\widehat{E_i}(\widehat{H}(S_i)))$ under its commutative public key, obtaining $\widehat{E_{(i+1) \bmod k}}(P_i(\widehat{E_i}(\widehat{H}(S_i))))$;
   - randomly permutes all the encrypted items in $\widehat{E_{(i+1) \bmod k}}(P_i(\widehat{E_i}(\widehat{H}(S_i))))$, obtaining $P_{(i+1) \bmod k}(\widehat{E_{(i+1) \bmod k}}(P_i(\widehat{E_i}(\widehat{H}(S_i)))))$;
   - sends $P_{(i+1) \bmod k}(\widehat{E_{(i+1) \bmod k}}(P_i(\widehat{E_i}(\widehat{H}(S_i)))))$ to the successor in the ring, *i.e.*, $p_{(i+2) \bmod k}$, who has not yet encrypted and permuted the data set.
4. The process repeats until each provider p_i receives its own data set whose individual items have been encrypted and permuted by all providers in the ring.
5. All providers share their respective encrypted and permuted data sets, so that they can count the number of common unique items in these data sets to compute the set intersection cardinality, $|\cap_i S_i| = \delta$.
6. Each provider p_i finally computes Jaccard similarity by $J(S_0, S_1, ..., S_{i-1}) = \delta/m$.

dancy deployments involving p_i. After collecting these Jaccard similarities from all cloud providers, the auditing agent generates an auditing report ranking all the redundancy deployments based on the Jaccard similarities, and finally sends this report to the auditing client. For an n-of-m redundancy deployment ($n \leq m$), the auditing agent needs to obtain the Jaccard similarity across all the n cloud providers and the similarity across all the m cloud providers, then generate the auditing report.

At the client side, since the auditing client receives only a list ranking all potential redundancy deployments, she obtains no proprietary information about the participating cloud providers' internal infrastructures other than the information produced intentionally to describe their degree of independence.

Chapter 7

Limitations and Practical Issues

This chapter discusses a few INDaaS's limitations and areas for further exploration (§7.1). In addition, we also describe some practical issues regarding INDaaS deployment (§7.2).

7.1 Limitations and Potential Solutions

7.1.1 Complex Dependency Acquisition

Our current software dependency collector takes only static software dependency data into account. In practice, many cloud outages have been caused by more tricky bugs within complex cloud software stacks [13, 19, 60, 64]. Collecting such software dependency data would be challenging, and we are not aware of any existing systematic solutions. A potential solution may need to access the logs generated by various cloud components, and their configuration scripts. For example, we might be able to adapt software failure detection techniques based on mining console logs [83]. Zhao *et al.* [90] developed a tool that discovers static dependencies between Java programs by parsing these programs' code. In addition, traffic-aware optimizations, *e.g.*, the

UDS, BDS and ASD mechanisms proposed by Li *et al.* [53, 54], can greatly reduce the workload of the network dependency acquisition.

7.1.2 Dependencies Outside Control

INDaaS's utility directly relates to the level of detail and accuracy of the dependency information. Some important dependencies (*e.g.*, fibre-optic, power cables, as well as network and power loads) may lie outside cloud providers' controls or may be inaccessible due to insufficient privilege. Capturing these dependencies is critical to discovering unknown RGs. For example, the Baltimore tunnel fire in 2001 [65] had a significant effect on Internet traffic due to the wide-spread use of the passage way for laying fibre-optic cables. One possible solution to this limitation may be open standards for sharing information on third-parties infrastructures that manually collect this type of information.

7.2 Practical Issues

The motivation for auditing clients to use INDaaS is straightforward: they can choose redundancy deployments with better independence property, and can understand unexpected common dependencies which may lead to correlated failures. On the other hand, especially in the PIA case the cloud providers who offer data sources may not explicitly benefit from honestly participating in such a process. We now discuss what incentives the cloud providers have to join PIA, and how they deal with dishonest cloud providers.

7.2.1 Do cloud providers have incentives to join?

By participating in PIA, a cloud provider has the opportunity to better understand its potential dependency issues in relation to other cloud providers. While the cloud provider may not learn which specific components overlap with others, it can learn to what extent common dependencies exist between itself and other cloud providers. PIA thus gives cloud providers the opportunity to improve the independence of their deployments. Another potential incentive is that cloud providers not participating in PIA will not appear among the alternative cloud providers that PIA offers to auditing clients. As a result, the clients may be less likely to learn or consider these non-participating alternatives while evaluating various redundancy deployments. These non-participating cloud providers may lose potential customers due to the lack of the PIA "reliability label" or merely due to not being on the PIA "certified provider list". Finally, PIA offers cloud providers the opportunity to improve their reputation for transparency and reliability, without risking significant leaks of proprietary secrets about their infrastructure. Joining PIA offers cloud providers a privacy-preserving way to increase the effective transparency of their infrastructures.

7.2.2 Will cloud providers behave honestly?

Some cloud providers might execute PIA dishonestly, for example, by declaring a subset of their actual component-sets. In doing so, these providers might benefit from their dishonesty by appearing to have a smaller set intersection and hence greater independence than other providers. Thus, dishonest cloud providers might be ranked higher in the resulting ranking list. To address this issue, we could use the trusted hardware (*e.g.*, TPM) to remotely attest whether cloud providers are performing PIA as required. Recent efforts such as Excalibur [67] have deployed

TPM into some cloud platforms successfully.

A less technical solution is to rely on the common business practice of “trust but leave an audit trail.” For most executions of PIA, the auditing client simply trusts the participating cloud providers to feed honest and accurate information into the protocol, but the providers must also save and digitally sign the data they used. If an auditing client suspects dishonesty, or during occasional “spot-checks,” a specially-authorized independent authority – analogous to the IRS – might perform a “meta-audit” of the provider’s PIA records, so that a persistently dishonest participant risks eventually getting caught.

Chapter 8

Implementation and Deployment

We have built an INDaaS prototype system written in a mix of Python and Java. As written, the code is generally unoptimized and we make no claims that it is the "best" implementation of an INDaaS architecture and deployment. In this chapter, we first focus on the implementation and deployment of SIA (§8.1), followed by PIA (§8.2).

8.1 Structural Independence Auditing

Figure 8.1 shows the key components of an INDaaS prototype in the SIA scenario. SIA implementation uses three INDaaS prototype components: INDaaS client, INDaaS agent and dependency acquisition module.

Auditing client. Our auditing client software, currently written in Python, is deployed on a machine maintained by the cloud provider itself, *e.g.*, Node A in Figure 8.1. The auditing client communicates with the auditing agent to send the specification and receive the auditing report.

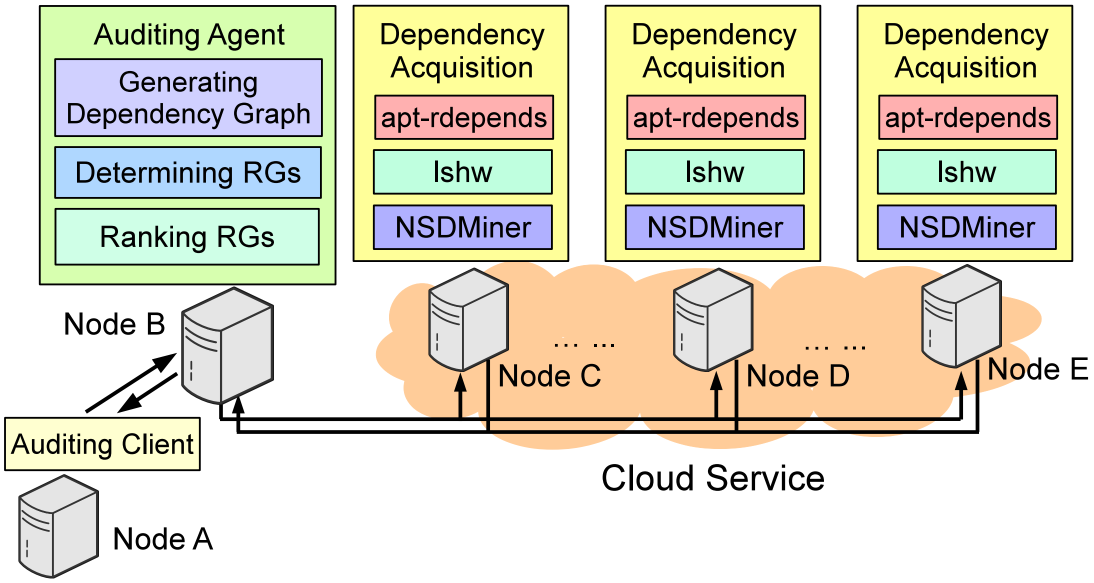


Figure 8.1: Implementation and deployment of SIA.

Dependency acquisition. The dependency acquisition modules, written in Python, are deployed on each worker machine to support the audited redundancy deployment in a cloud, *e.g.*, Node C-E in Figure 8.1. Our current dependency acquisition implementation includes three open-source tools: NSDMiner [59], lshw [6], and apt-rdepends [3], which are used to collect network, hardware, and software dependencies, respectively. Each node's dependency module is responsible for recording all of its dependency information in a XML file, and then sends the file to the agent. Since each worker machine executes its local dependency acquisition modules separately, the dependency acquisition process can be parallelized.

Auditing agent. The auditing agent, written in Python with the NetworkX library [10], is deployed on another machine, *e.g.*, Node B in Figure 8.1. The agent has implemented all the SIA designs mentioned in §5. It collects the dependency data from the dependency acquisition modules on each worker machine over the SSH channels. The agent then audits the collected dependency data, and returns the auditing report back to the auditing client. The current agent implementation is a centralized component running on a single machine.

Parallelizing SIA process. The data acquisition and analysis modules in SIA implementation may benefit from some form of parallelization. Each data source begins by locally generating sub-fault graph representing its own dependency structure. Then data sources in each redundancy deployment randomly elect a "leader" data source for collecting the sub-fault graphs of other data sources in the same redundancy deployment, establishing a whole fault graph, *i.e.*, using an AND gate to connect all the sub-fault graphs. Finally, the leader data source in each redundancy deployment sends the fault graph to the agent. This optimization in principle distributes fault graph generation tasks to each data source, thus parallelizing the fault graph generation and even analysis operations.

8.2 Private Independence Auditing

Figure 8.2 presents the key components of our INDaaS prototype in the PIA scenario.

Auditing client and auditing agent. In PIA, the auditing client and auditing agent are implemented and deployed in a similar way as in SIA, except that the auditing agent is deployed on a machine maintained by a third-party auditor, *i.e.*, not by any audited cloud provider. More specifically, the two differences are, first, the machine running the agent is maintained by a third-party auditor who should not be any of audited cloud providers, and, second, the agent in the PIA case does not need to use functions for the SIA case, *e.g.*, the RG determination algorithms.

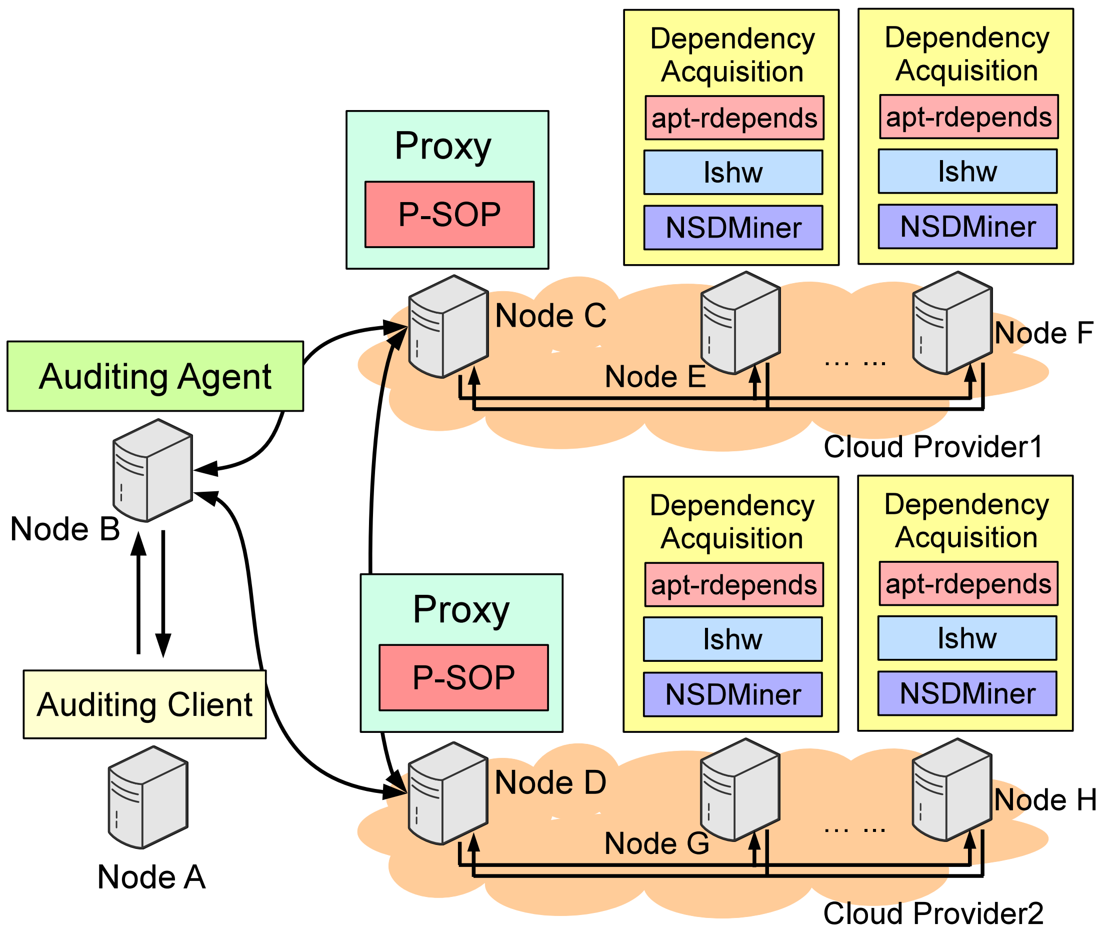


Figure 8.2: Implementation and deployment of PIA.

Dependency acquisition and proxy. For each cloud provider, there are three dependency acquisition modules deployed on each of its worker machines, as in SIA. The implementation of the dependency acquisition module in the PIA case is the same as the one in the SIA case. Moreover, we implemented a proxy in Java for each cloud provider. The proxy first collects dependency data from the dependency acquisition modules deployed in its own cloud, and then runs the private set intersection cardinality protocol (P-SOP) together with the proxies operated by other cloud providers. In particular, we implemented the P-SOP protocol with MD5, Java permutation, and the commutative RSA encryption scheme [70].

8.2.1 Comparison: Kissner and Song Protocol

In order to compare with P-SOP, we additionally implemented a well-known private set intersection cardinality protocol, Kissner and Song (KS) [51].

KS protocol. KS allows a group of k parties with multisets $S_1, ..., S_k$ to compute privately $|\cap_i S_i|$, the number of elements they have in common, without learning the specific elements in $\cap_i S_i$. In addition, KS needs to use a homomorphic encryption scheme such as the Paillier cryptosystem [62]. At the initial phrase of the protocol, the k parties use a homomorphic cryptosystem to share a secret key sk amongst themselves, while the corresponding public key pk is known to all parties. With the above keys in hands, the protocol computes $|\cap_i S_i|$ as follows. First, each party p_i encrypts a polynomial P_i whose roots are the elements of its local input data set S_i. The encrypted polynomials are essentially added together, thus yielding a polynomial P whose roots are the elements in the union sets of all the parties. Each party p_i, then, evaluates P on the elements e_{ij} of its local data set S_i, yielding values $v_{ij} = P(e_{ij})$; however, since sk is shared, no individual party can decrypt the v_{ij}. The k parties securely re-randomize and shuffle the v_{ij} based on the approach [61], such that each party learns all the v_{ij} but cannot tell which party it came from. Finally, The k parties jointly decrypt the v_{ij}. If there are n elements in the intersection, this would yield $n \cdot k$ zeros; thus, each party can compute the final result by dividing the number of zeros by k.

Implementation of KS. Following the above description, we have implemented KS in Java, and have deployed KS for comparing with P-SOP. To the best of our knowledge, this is the first effort implementing a practical system based on KS. In our implementation, we used MD5 for hashing operations, and use the `thep` library [7] to implement Paillier cryptosystem.

Chapter 9

Evaluation

This chapter evaluates the practicality (§9.1) and performance (§9.2) of INDaaS.

9.1 Practicality Evaluation: Case Studies

This section evaluates INDaaS's practicality through three small but realistic case studies with respect to unexpected common network, hardware, and software dependencies, respectively.

9.1.1 Common Network Dependency

Our first case study targets a scenario similar to the example given in the introduction. A data center operator, Alice, wants to deploy a new service S in her data center, and replicates the critical states of S across two servers within her data center. Before service deployment, Alice uses INDaaS to structurally audit the data center network in order to avoid potential correlated failures resulting from common network dependencies. We used a real data center topology [22] to model Alice's data center network. As shown in Figure 9.1, this data center has many Top-of-

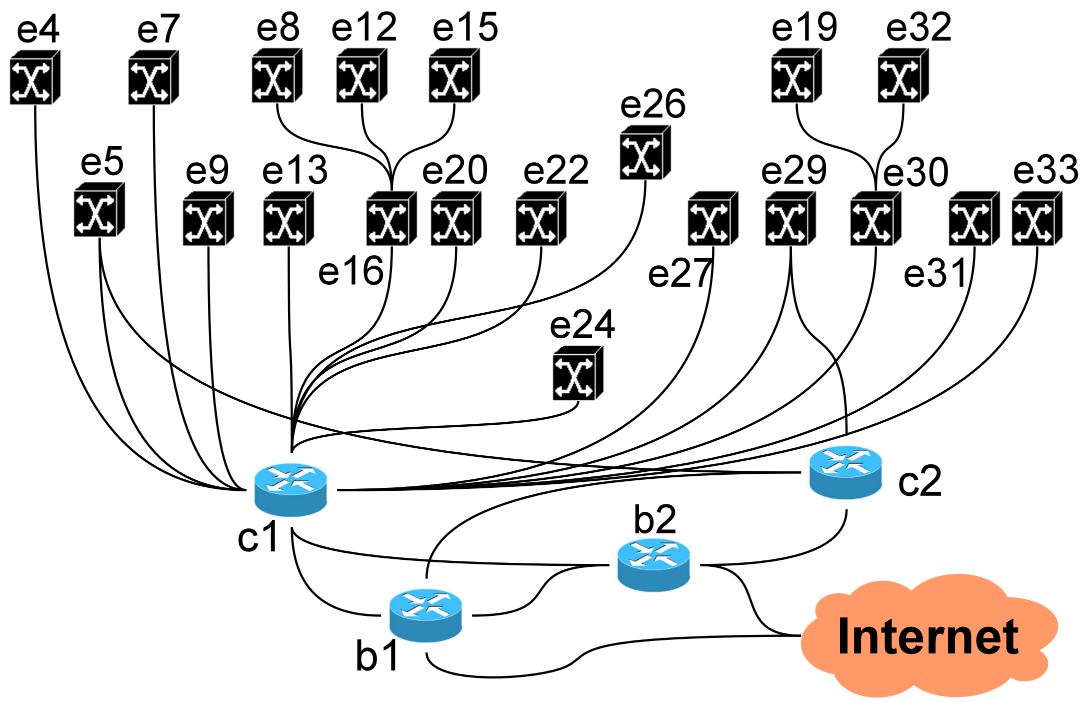


Figure 9.1: Case study for common network dependency.

Rack (ToR) switches (*i.e.*, e1-e33) each of which is connected to an individual rack. There are four core routers (*i.e.*, b1, b2, c1, and c2) connecting ToR switches to the Internet.

The INDaaS first collects network dependencies, and then executes the SIA protocol to provide auditing at the fault graph level. The auditing report generated by our prototype, based on the failure sampling algorithm (which we ran for 10^6 rounds) and the size-based ranking algorithm, suggests that {Rack 5, Rack 29} is the most-independent deployment in this scenario. Without INDaaS, it is highly possible that Alice builds a two-way replication with unexpected RGs. For instance, if she uses redundancy {Rack 4, Rack 7}, the failure of C1 would invalidate her efforts.

A formal analysis indicates that there are 190 different two-way redundancy deployments, among which 27 do not have unexpected RGs. This means, without INDaaS, a random selection leads to only 14% probability for Alice to avoid cor-

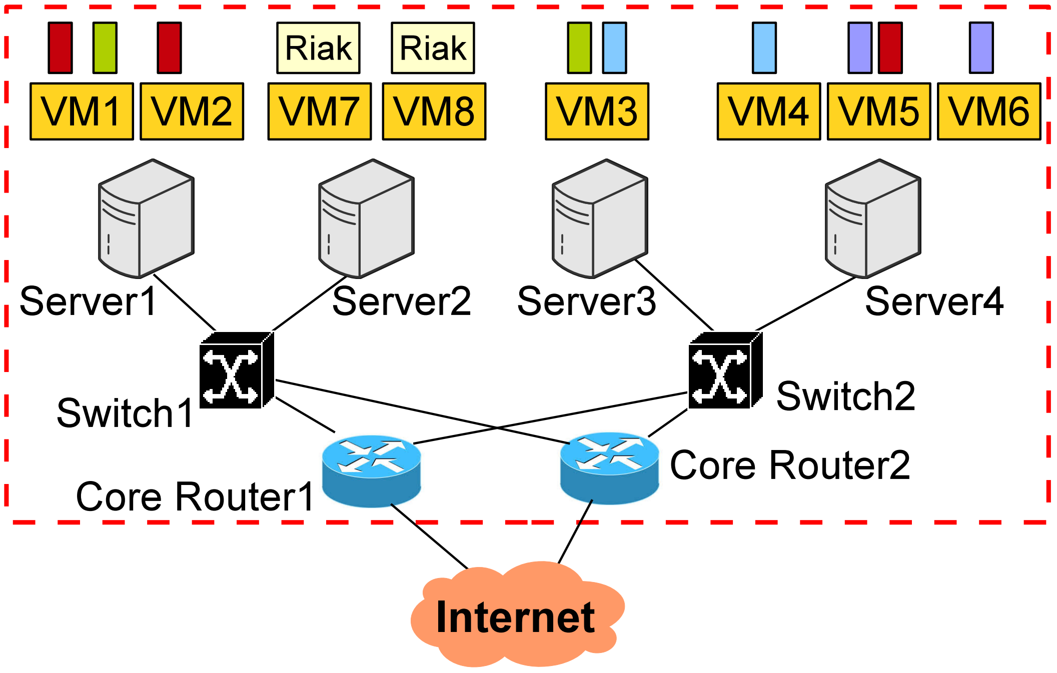


Figure 9.2: Case study for common hardware dependency.

related failures. Furthermore, if we assume the failure probability of all network devices is 0.1, the redundancy deployment {Rack 5, Rack 29} is indeed the one with the lowest failure probability.

9.1.2 Common Hardware Dependency

As shown in Figure 9.2, we have built a simple IaaS cloud in the lab with four servers and four switches. While perhaps not an impressive cloud, this configuration as will be shown sufficiently exercises the INDaaS prototype. We used OpenStack to support the automatic virtual machine (VM) management, and deployed various services on VMs for different uses. OpenStack automatically handles the allocation of resources, and thus Alice simply provides virtual machine images containing the services to the cloud. In particular, we deployed an S3-like Riak [11] cloud storage service. For redundancy, Riak was run on two VMs (VM7 and VM8).

Table 9.1: A RG-ranking list obtained by minimal RG algorithm and size-based ranking metric. SIA randomly orders RGs with the same size.

NO.	**Minimal RGs**	**RG-size**
1	{Server2}	1
1	{Switch1}	1
3	{Core1 & Core2}	2
3	{VM7 & VM8}	2
5	... ...	...

Before releasing the Riak storage service for public use, we ran SIA to check whether there would be any unexpected RGs. The SIA process is executed as follows: 1) collecting network, hardware and software dependency information; 2) building a fault graph with the information; and 3) generating a ranking list containing RGs for this service. We chose to use the minimal RG algorithm and the size-based ranking algorithm. The top 4 RGs in the RG ranking list generated by our prototype are: {Sever2}, {Switch1}, {Core1 & Core2}, and {VM7 & VM8}. Note that SIA randomly orders RGs with the same size. With this list, we noticed that we had failed to improve the reliability of Riak service via redundant VMs, because the automatic placement module in OpenStack placed the two redundant VMs on the same server (a shared hardware source). As a result, the failure of that server would undermine the redundancy effort. The fundamental cause is that the OpenStack's automatic virtual machine placement policy randomly selects from the least loaded resources to host a VM.

To make the most effective redundancy deployment, we consulted INDaaS for an auditing report on the independence of all potential redundancy deployments. According to the report, which suggests {Server2 and Server3}, we re-deployed the two redundant VMs for the Riak storage service.

Table 9.2: Ranking lists of two- and three-way redundancy deployments based on Jaccard similarities. Cloud1, 2, 3, and 4 are equipped with Riak, MongoDB, Redis, and CouchDB, respectively.

Rank	Two-Way Redundancy Deployment	Jaccard
1	Cloud2 & Cloud4	0.1419
2	Cloud2 & Cloud3	0.1547
3	Cloud1 & Cloud4	0.2081
4	Cloud1 & Cloud3	0.2939
5	Cloud3 & Cloud4	0.3489
6	Cloud1 & Cloud2	0.5059
Rank	Three-Way Redundancy Deployment	Jaccard
1	Cloud2 & Cloud3 & Cloud4	0.1128
2	Cloud1 & Cloud2 & Cloud4	0.1207
3	Cloud1 & Cloud3 & Cloud4	0.1353
4	Cloud1 & Cloud2 & Cloud3	0.1536

9.1.3 Common Software Dependency

The last case study targets a scenario where INDaaS offers private independence auditing across multiple cloud providers. In particular, a service provider, Alice, wants a reliable storage solution leveraging multiple cloud providers, *e.g.*, iCloud uses Amazon EC2 and Microsoft Azure for its reliable storage. Suppose Alice has found four alternative cloud providers: Cloud 1-4, each of which offers a key-value store. Alice then consults INDaaS for a redundancy deployment to avoid correlated failures caused by any shared software dependency [36].

Here, we chose four popular key-value storage systems, *i.e.*, Riak, MongoDB, Redis, and CouchDB. As shown in Figure 9.3, we assigned each one to a cloud provider as follows, Cloud1: Riak, Cloud2: MongoDB, Cloud3: Redis, and Cloud4: CouchDB. Suppose each cloud provider has used our prototype to automatically collect the software dependencies of the packages and libraries in its storage system. Our PIA protocol privately computes the Jaccard similarity for each potential redundancy deployment. Table 9.2 shows the ranking lists of various two- and three-

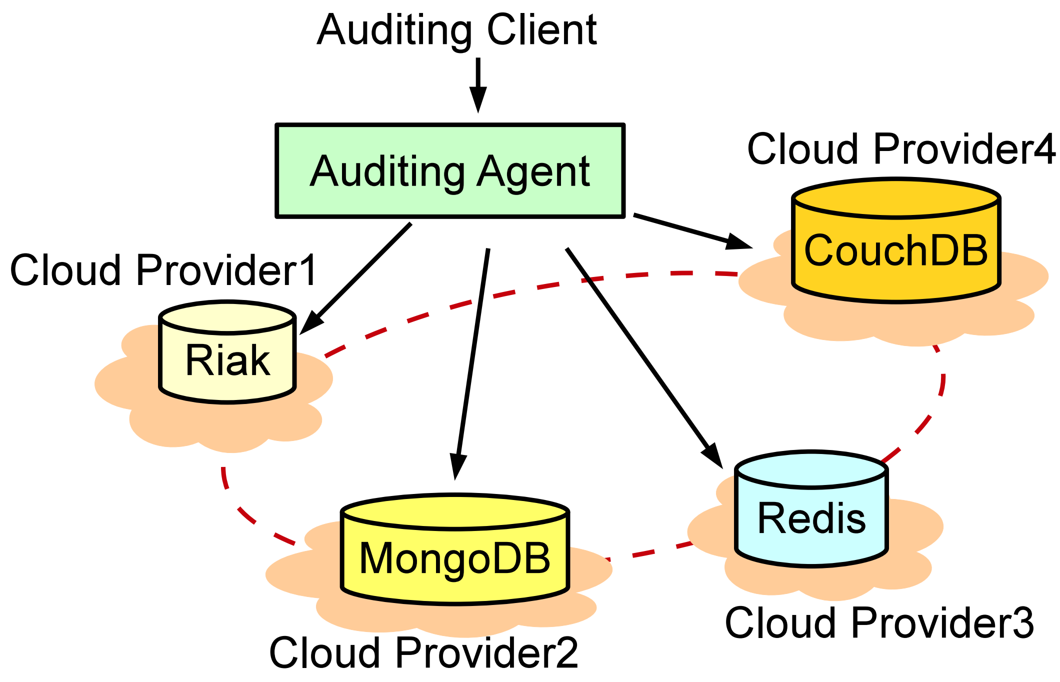


Figure 9.3: Case study for common software dependency.

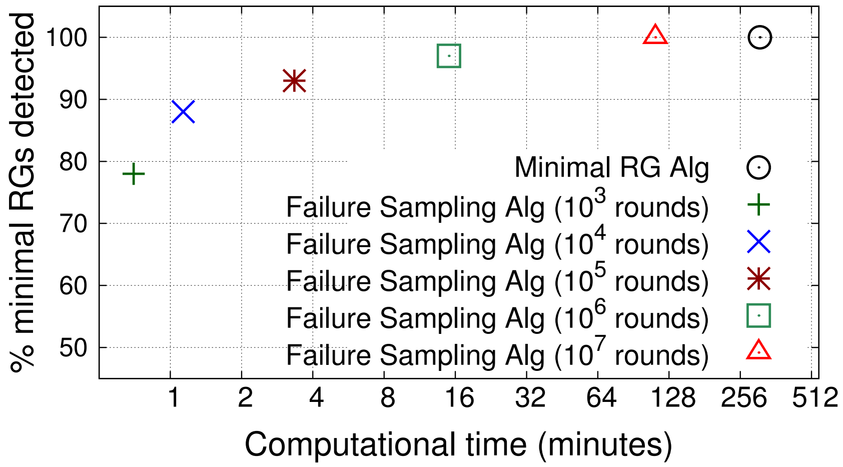
way redundancy deployments. We obtain Jaccard similarity without using MinHash option for accurate independence scores.

9.2 Performance Evaluation

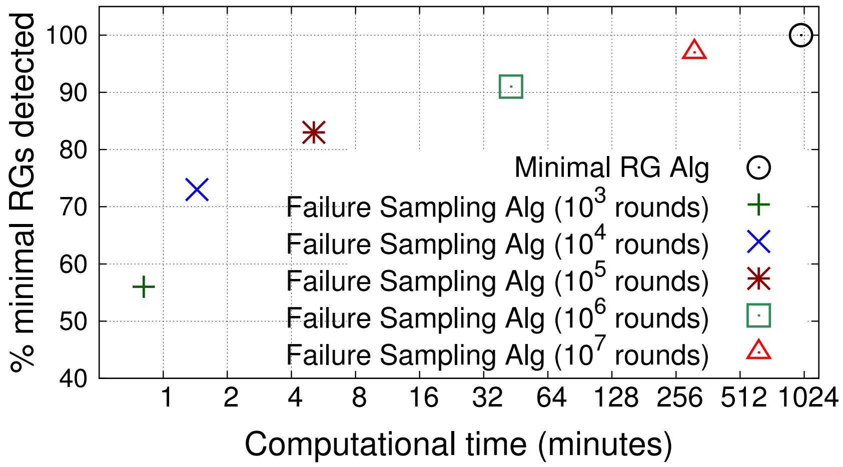
We evaluate INDaaS's two major components: SIA and PIA. The performance evaluation was conducted on a research cluster of 40 workstations equipped with Intel Xeon Quad Core HT 3.7 GHz CPU and 16 GB RAM [14].

9.2.1 SIA: Efficiency v.s. Accuracy

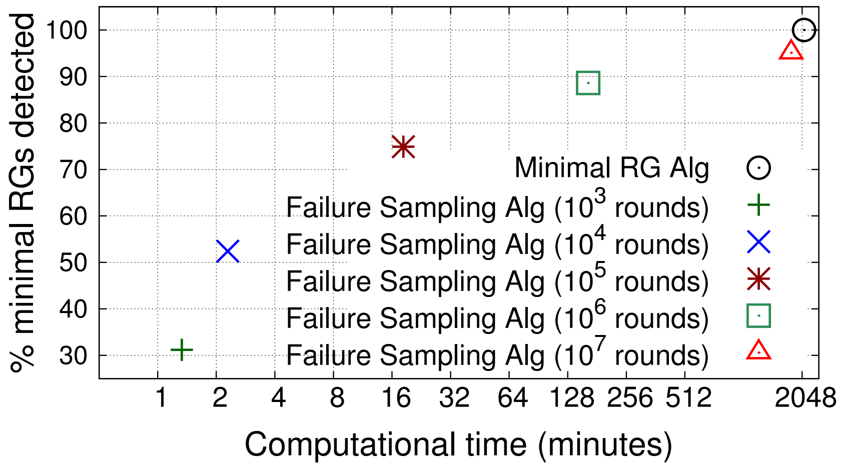
We first explore the efficiency/accuracy trade-off between SIA's two algorithms for analyzing a dependency graph: the minimal RG algorithm and the failure sampling algorithm (see §5.3). The former is able to produce a complete set of all minimal



(a) Topology A: $1,344$ devices.



(b) Topology B: $4,176$ devices.



(c) Topology C: $30,528$ devices.

Figure 9.4: Performance evaluation of the minimal RG algorithm and the failure sampling algorithm in SIA.

Table 9.3: Configurations of the generated topologies.

	Topology A	Topology B	Topology C
# switch ports	16	24	48
# core routers	64	144	576
# agg switches	128	288	1,152
# ToR switches	128	288	1,152
# servers	1,024	3,456	27,648
Total # devices	1,344	4,176	30,528

RGs but does so within NP-hard complexity. The failure sampling algorithm, on the other hand, cannot guarantee completeness but runs in linear time. We generate three topologies from a small-scale cloud deployment to a large-scale deployment, based on the three-stage fat tree model [57]. These topologies include the typical components within a commercial data center: servers, Top-of-Rack (ToR) switches, aggregation switches, and core routers. Table 9.3 gives the detail of these generated topologies.

We compare the computational overhead of the accurate but NP-hard minimal RG algorithm to that of the failure sampling algorithm with various sampling rounds (10^3 to 10^7). Figure 9.4 shows the result that the failure sampling algorithm runs much more efficiently than the minimal RG algorithm while achieving a reasonably high accuracy. For example, in topology B, the failure sampling algorithm uses 90 minutes to detect 92% of all the minimal RGs with 10^6 sampling rounds, in comparison to 1046 minutes for the minimal RG algorithm.

9.2.2 PIA: System Overheads

To better understand the performance of PIA, we compare P-SOP with KS from both aspects of bandwidth and computation.

For a private independence auditing system, the cryptographic operations tend to be the major bandwidth and computational bottleneck. Thus, we evaluate the

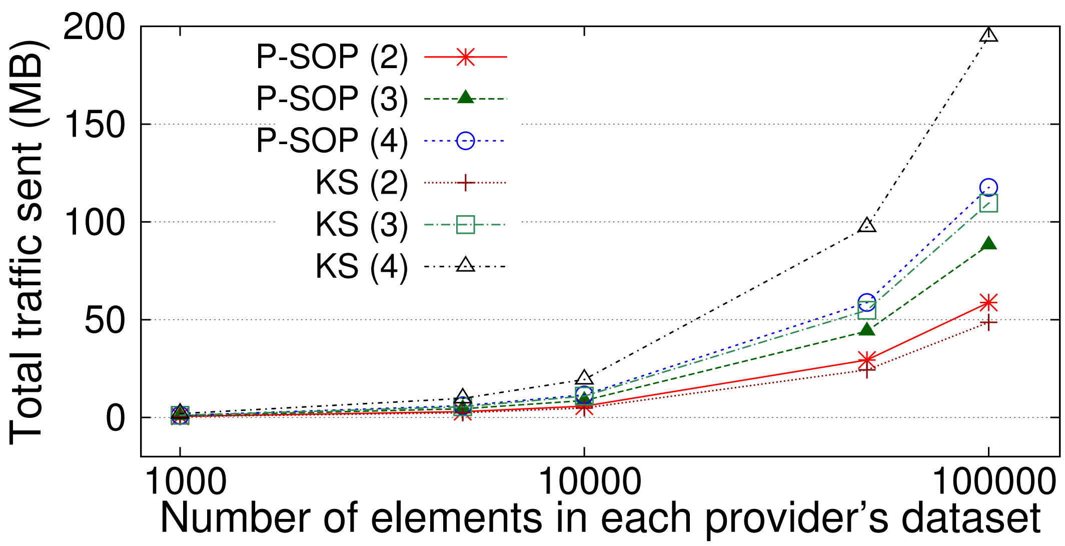


Figure 9.5: Bandwidth overhead evaluation of PIA. P-SOP (k) and KS(k) mean that there are k cloud providers participating in the P-SOP and KS protocols, respectively. The commutative encryption in P-SOP uses a 1024-bit key, and the homomorphic encryption in KS also uses a 1024-bit key.

performance of PIA by comparing the bandwidth and computational overheads of P-SOP protocol with KS protocol. Specifically, the cryptographic primitives of P-SOP are hashing, commutative encryption, and permutation. The KS protocol is mainly built on hashing, homomorphic crypto operations, and permutation.

In the evaluation, there are k cloud providers with n elements in each provider's local dataset. We set k to 2, 3 and 4, and vary n between $1,000$ and $100,000$ to cover a wide range of real-world settings. We measure and compare P-SOP with KS in terms of their bandwidth and computational overheads at each such cloud provider. Figure 9.5 and 9.6 show the bandwidth overhead and computational overhead, respectively.

With a small number of cloud providers ($e.g.$, $k = 2$), the bandwidth overhead of KS is comparable to that of P-SOP. However, with an increasing number of cloud providers, KS's bandwidth overhead increases much faster than P-SOP's. With respect to the computational overhead, P-SOP outperforms KS by a few orders of

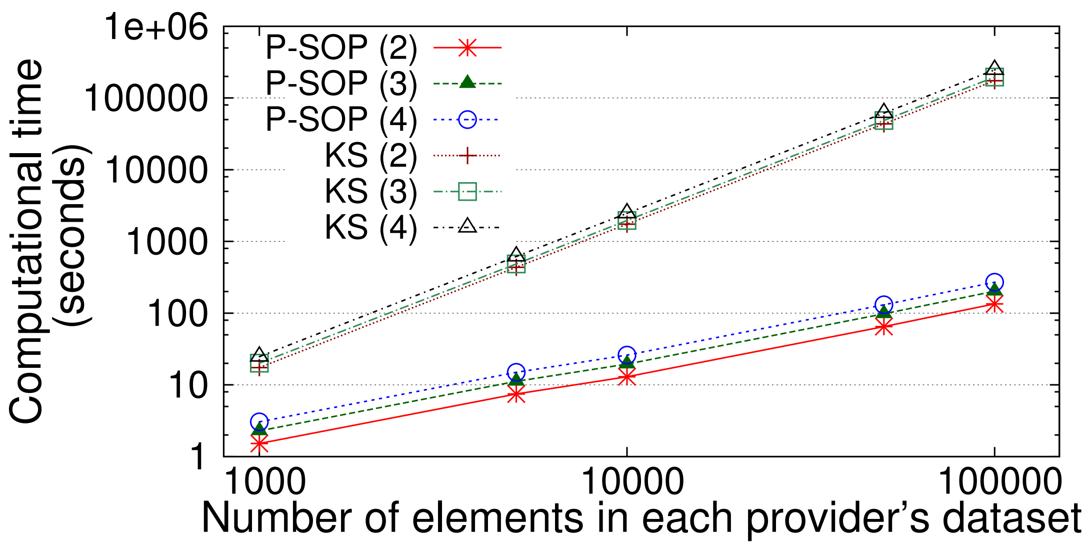


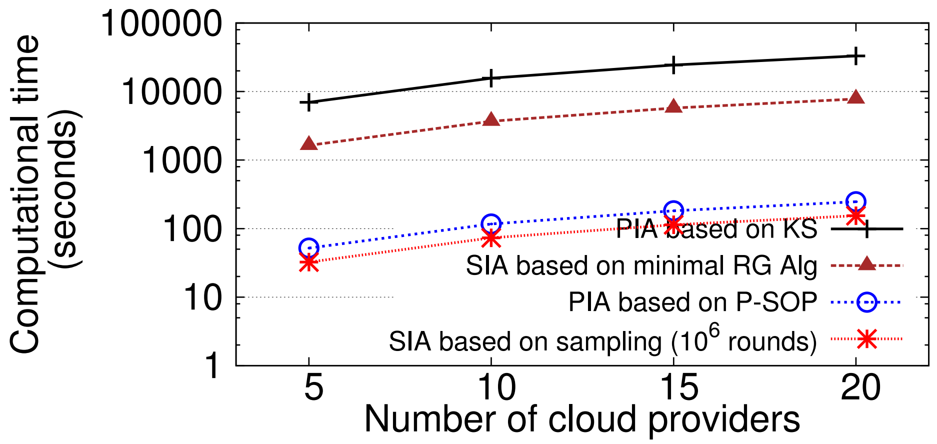
Figure 9.6: Computational overhead evaluation of PIA. P-SOP (k) and KS(k) mean that there are k cloud providers participating in the P-SOP and KS protocols, respectively. The commutative encryption in P-SOP uses a 1024-bit key, and the homomorphic encryption in KS also uses a 1024-bit key.

magnitude although both protocols' computational overheads increase almost linearly with the number of elements in each cloud provider's dataset. Altogether, the evaluation shows that our PIA system can efficiently handle large cloud providers each with even hundreds of thousands of system components.

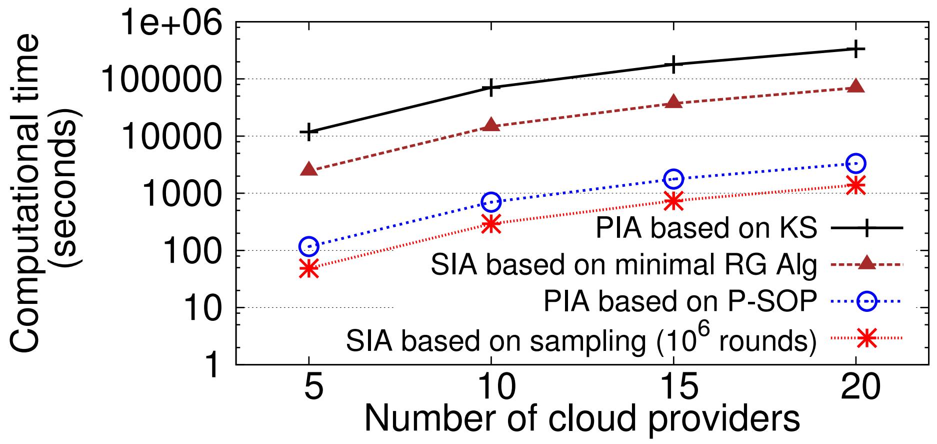
In particular, in a three redundancy deployment, an alternative service provider owning a local data set with $100,000$ elements can finish all the computation operations with about 200 seconds, if the provider is using P-SOP. Regarding bandwidth, we concede KS protocol is less than P-SOP in the two redundancy deployment case. However, P-SOP is more efficient than KS in other two cases.

9.2.3 Comparison: SIA Versus PIA

Compared with the SIA where there is a trusted auditor, we would also like to understand how much extra overhead the PIA approach incurs to preserve the secrecy of each participating cloud provider's data. Assume each cloud provider maintains a



(a) Two-way redundancy.



(b) Three-way redundancy.

Figure 9.7: Performance comparison between SIA and PIA. Each cloud provider maintains a 10,000-element dataset.

local dataset containing $10,000$ elements. To preserve secrecy for each cloud provider, an auditing client relies on either the PIA system or the comparable KS-based system to determine the most independent redundancy deployment. For a comparison, we also assume another setting where there exists a trusted auditor who knows all cloud providers' datasets. This trusted auditor runs SIA at the component-set level of detail based on the minimal RG algorithm or the failure sampling algorithm with 10^6 rounds.

The motivation we construct the above scenario is to compare our auditing algorithms in a "hybrid" auditing task case. Our experiments measure computational by changing $n = 5, 10, 15$ and 20 (n is the number of alternative cloud providers). We assume that the trusted auditor, can get all cloud providers' datasets, then generates a dependency graph at component-set level of detail with the collected data, and finally performs the auditing algorithms on the graph.

Figure 9.7a and 9.7b show the computational overheads of these independence calculations for all potential two- and three-way redundancies, respectively. As shown in Figure 9.7a and Figure 9.7b, preserving the secrecy of cloud providers' data does incur extra overhead. Surprisingly, this cost is not as high as might be expected: we see that the computational overhead of "PIA based on P-SOP" is less than twice that of "SIA based on sampling (10^6 rounds)". The SIA sampling scheme does implement a more general analysis than PIA, supporting fault graphs rather than just component sets. Unsurprisingly, both "PIA based on KS" and "SIA based on minimal RG Alg" do not scale well.

Chapter 10

Related Work

Providing audits for clouds is a well-known approach to increase reliability [68]. Practical and systematic cloud auditing, however, still remains an open problem. To the best of our knowledge, INDaaS is the first systematic effort to enable independence audits for cloud services.

10.1 Auditing Systems

As the earliest advocate of cloud auditing, Shah et al. [68] proposed auditing techniques could be used to build trust between cloud consumers and cloud storage services. They classify storage service auditing into two categories: *internal auditing* and *external auditing*. The former checks internal structures of services and assesses whether the services fail to meet their SLAs, while the latter makes use of privacy-preserving approaches to assist consumers to detect integrity of their stored data.

Privacy-preserving auditing systems. Following the auditing concept proposed by Shah *et al.* [68], many privacy-preserving auditing systems have been proposed extending this approach [69,76–79,84].

Similar to PIA, iRec [87] and Xiao *et al.* [82] also focused on analyzing correlated failures resulting from the common infrastructure dependencies across multiple cloud providers. These efforts proposed using the private set intersection cardinality protocol [34] and the secure multi-party computation protocol [85] to perform the dependency analysis in a privacy-preserving fashion, respectively. These initial efforts did not scale to handle realistically large cloud datasets, however, and have not been implemented or deployed, even in a lab environment.

10.2 Diagnosis and Accountability Systems

Diagnosis systems, unlike auditing, attempt to discover failures after they occur. For example, many inference-based diagnosis systems [19,29,44,49] have been proposed to obtain the network dependencies of a cloud service when a failure occurs. Unlike existing diagnosis systems, NetPilot [81] aimed to mitigate these failures rather than directly localize their sources.

Accountability systems attempt to place blame after failures occur, whereas our auditing system attempts to prevent failures in the first place. Haeberlen [38] proposed using third-party verifiable evidence to determine whether the cloud customer or the cloud provider should be held liability when a failure occurs.

10.3 Private Set Operations

There have been many private set operational protocols proposed in the past years. Secure multi-party computation (SMPC) [85] is a general approach to supporting computation on private data including set operations. However, current circuitbased SMPC protocols are too expensive and scale poorly to large computations. Arawal *et al.* [17] proposed a private set intersection cardinality protocol based on commutative encryption. This protocol was limited to two-party cases, however. Vaidya and Clifton [72] extended this protocol to support more than two parties, and optimized its efficiency. Similar to PIA, a two-party private Jaccard similarity computation protocol was proposed by Blundo *et al.* [24]. However, to the best of our knowledge, all the above protocols have not been implemented systematically yet.[1]

The first private set intersection cardinality protocol based on homomorphic encryption was proposed by Freedman *et al.* [34], which could privately compute the number of elements common to two datasets (called PSI-CA). The protocol has a communication complexity within $O(N)$ and a computational complexity within $O(N \ln \ln N)$ for each party and can be efficiently used only for two party set intersection cardinality computations. Nevertheless, a multi-party solution based on Freedman *et al.*is not obvious [51]. Hohenberger *et al.*proposed optimized version of PSI-CA [43], and Kiayias and Mitrofanova [50] proposed another version of PSI-CA to efficiently handle malicious provers [43]. Kissner and Song proposed multi-party private set operations based upon homomorphic encryption and polynomial generation [51].

[1]The only practical tool based on those techniques is SeFaSI developed by Zander *et al.* [86]; however, the effort only focuses on limited scenarios for particular purposes.

Chapter 11

Conclusion

This thesis has presented INDaaS, an architecture to audit the independence of future or existing redundant systems for services in the cloud. INDaaS employs pluggable dependency acquisition modules to collect the structural dependency information of redundant systems, and provides various auditing algorithms to quantify the independence of redundant systems with varying performance, precision, and data secrecy trade-offs. Our proof-of-concept prototype and experimental results suggest that INDaaS could be both practical and effective in detecting and heading off potential common dependencies resulting in unexpected correlated failures before the service outages occur.

Beyond our basic goal, INDaaS's techniques may be useful for improving diagnosis and recommendation systems as well as making clouds more easily insurable.

Bibliography

[1] Amazon S3's Redundant Storage. http://aws.amazon.com/s3/faqs/.

[2] Apple's iCloud runs on Microsoft's Azure and Amazon's cloud. http://venturebeat.com/2011/09/03/icloud-azure-amazon/.

[3] apt-rdepends. http://packages.debian.org/sid/apt-rdepends.

[4] Correlated Failures within EBS and EC2. https://aws.amazon.com/message/680342/.

[5] EC2 Crash in April of 2011. http://money.cnn.com/2011/04/21/technology/amazon_server_outage/index.htm.

[6] Hardware Lister (lshw). http://ezix.org/project/wiki/HardwareLiSter.

[7] Homomorphic encryption project. https://code.google.com/p/thep/.

[8] Internet Outages Highlight Problem for Cloud Computing: Actual Clouds. http://www.slate.com/blogs/future_tense/2012/07/02/amazon_ec2_outage_netflix_pinterest_instagram_down_after_aws_cloud_loses_power.html.

[9] Netflix Outage. http://www.huffingtonpost.com/2012/07/02/amazon-power-outage-cloud-computing_n_1642700.html.

[10] NetworkX. http://networkx.github.com/.

[11] Riak 1.3.1. http://docs.basho.com/riak/1.3.1/downloads/.

[12] snort. http://www.snort.org.

[13] The 10 Biggest Cloud Outages of 2013. http://www.crn.com/slide-shows/cloud/240158298/the-10-biggest-cloud-outages-of-2013-so-far.htm?pgno=1.

[14] Yale Cluster. http://zoo.cs.yale.edu/.

[15] Zynga. https://zynga.com/.

[16] Hussam Abu-Libdeh, Paolo Costa, Antony I. T. Rowstron, Greg O'Shea, and Austin Donnelly. Symbiotic routing in future data centers. In *ACM SIGCOMM (SIGCOMM)*, New Delhi, India, August 2010.

[17] Rakesh Agrawal, Alexandre V. Evfimievski, and Ramakrishnan Srikant. Information sharing across private databases. In *ACM International Conference on Management of Data (SIGMOD)*, San Diego, CA, June 2003.

[18] Marcos Kawazoe Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, October 2003.

[19] Paramvir Bahl, Ranveer Chandra, Albert G. Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *ACM SIGCOMM (SIGCOMM)*, Kyoto, Japan, August 2007.

[20] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, December 2004.

[21] Cristina Basescu, Christian Cachin, Ittay Eyal, Robert Haas, Alessandro Sorniotti, Marko Vukolic, and Ido Zachevsky. Robust data sharing with key-value stores. In *42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Boston, MA, June 2012.

[22] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Internet Measurement Conference (IMC)*, Melbourne, Australia, November 2010.

[23] Alysson Neves Bessani, Miguel P. Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. DepSky: Dependable and secure storage in a cloud-of-clouds. In *6th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, Salzburg, Austria, April 2011.

[24] Carlo Blundo, Emiliano de Cristofaro, and Paolo Gasti. EsPRESSo: Efficient privacy-preserving evaluation of sample set similarity. In *DPM/SETOP*, September 2012.

[25] Carlo Blundo, Emiliano de Cristofaro, and Paolo Gasti. EsPRESSo: Efficient privacy-preserving evaluation of sample set similarity. *Journal of Computer Security*, 22(3):355–381, 2014.

[26] Nicolas Bonvin, Thanasis G. Papaioannou, and Karl Aberer. A self-organized, fault-tolerant and scalable replication scheme for cloud storage. In *1st ACM Symposium on Cloud Computing (SoCC)*, Indianapolis, IN, June 2010.

[27] Andrei Z. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences (SEQUENCES)*, June 1997.

[28] Mike Y. Chen, Anthony Accardi, Emre Kiciman, David A. Patterson, Armando Fox, and Eric A. Brewer. Path-based failure and evolution management. In *1st USENIX Symposium on Networked System Design and Implementation (NSDI)*, San Francisco, CA, March 2004.

[29] Xu Chen, Ming Zhang, Zhuoqing Morley Mao, and Paramvir Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, December 2008.

[30] Jack Clark. Lightning strikes Amazon's European cloud. *ZDNet*, August 2011. http://www.zdnet.com/lightning-strikes-amazons-european-cloud-3040093641/, accessed on Sep 9, 2014.

[31] John Dunagan, Nicholas J. A. Harvey, Michael B. Jones, Dejan Kostic, Marvin Theimer, and Alec Wolman. FUSE: Lightweight guaranteed distributed failure notification. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, December 2004.

[32] Bryan Ford. Icebergs in the clouds: the *other* risks of cloud computing. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, Boston, MA, June 2012.

[33] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, October 2010.

[34] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *Conference on Theory and applications of cryptographic techniques (EUROCRYPT)*, Interlaken, Switzerland, May 2004.

[35] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *ACM SIGCOMM (SIGCOMM)*, Toronto, Canada, August 2011.

[36] Dan Greer. Heartbleed as metaphor. *Lawfare*, April 2014. http://www.lawfareblog.com/2014/04/heartbleed-as-metaphor/, accessed on Sep 9, 2014.

[37] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. DCell: A scalable and fault-tolerant network structure for data centers. In *ACM SIGCOMM (SIGCOMM)*, Seattle, WA, August 2008.

[38] Andreas Haeberlen. A case for the accountable cloud. In *3rd ACM SIGOPS International Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, Big Sky, MT, October 2009.

[39] Andreas Haeberlen, Paarijaat Aditya, Rodrigo Rodrigues, and Peter Druschelnd. Accountable virtual machines. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, BC, Canada, October 2010.

[40] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: Practical accountability for distributed systems. In *21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, October 2007.

[41] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *2nd Symposium*

on Networked Systems Design and Implementation (NSDI), Boston, MA, May 2005.

[42] Keqiang He, Alexis Fisher, Liang Wang, Aaron Gember, Aditya Akella, and Thomas Ristenpart. Next step, the cloud: Understanding modern web service deployment in EC2 and Azure. In *Internet Measurement Conference (IMC)*, Barcelona, Spain, October 2013.

[43] Susan Hohenberger and Stephen A. Weis. Honest-verifier private disjointness testing without random oracles. In *Privacy Enhancing Technologies (PETS)*, New York, NY, June 2006.

[44] Barry Peddycord III, Peng Ning, and Sushil Jajodia. On the accurate identification of network service dependencies in distributed systems. In *26th Large Installation System Administration Conference (LISA)*, San Diego, CA, December 2012.

[45] Paul Jaccard. Étude comparative de la distribution florale dans une portion des Alpes et du Jura. *Bulletin de la Société Vaudoise des Sciences Naturelles*, 37(142):547–579, June 1901.

[46] Flavio Paiva Junqueira, Ranjita Bhagwan, Alejandro Hevia, Keith Marzullo, and Geoffrey M. Voelker. Surviving Internet catastrophes. In *USENIX Annual Technical Conference (USENIX ATC)*, Anaheim, CA, April 2005.

[47] Ivan P Kaminow and Thomas L Koch. *Optical Fiber Telecommunications IIIA*. Academic Press, New York, 1997.

[48] Srikanth Kandula, Dina Katabi, and Jean-Philippe Vasseur. Shrink: A tool for failure diagnosis in IP networks. In *MineNet*, Philadephia, PA, August 2005.

[49] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Paramvir Bahl. Detailed diagnosis in enterprise networks. In *ACM SIGCOMM (SIGCOMM)*, Barcelona, Spain, August 2009.

[50] Aggelos Kiayias and Antonina Mitrofanova. Testing disjointness of private datasets. In *Financial Cryptography and Data Security*, 2005.

[51] Lea Kissner and Dawn Xiaodong Song. Privacy-preserving set operations. In *25th Annual International Cryptology Conference (CRYPTO)*, Santa Barbara, CA. Springer, August 2005.

[52] Ramana Rao Kompella, Jennifer Yates, Albert G. Greenberg, and Alex C. Snoeren. IP fault localization via risk modeling. In *2nd USENIX Symposium on Networked System Design and Implementation (NSDI)*, Boston, MA, May 2005.

[53] Zhenhua Li, Cheng Jin, Tianyin Xu, Christo Wilson, Yao Liu, Linsong Cheng, Yunhao Liu, Yafei Dai, and Zhi-Li Zhang. Towards network-level efficiency for cloud storage services. In *14th ACM Internet Measurement Conference (IMC)*, Vancouver, Canada, November 2014.

[54] Zhenhua Li, Christo Wilson, Zhefu Jiang, Yao Liu, Ben Y. Zhao, Cheng Jin, Zhi-Li Zhang, and Yafei Dai. Efficient batched synchronization in Dropbox-like cloud storage services. In *14th ACM/IFIP/USENIX International Middleware Conference (Middleware)*, Beijing, China, December 2013.

[55] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Lombard, IL, April 2013.

[56] Jeffrey C. Mogul. Emergent (mis)behavior vs. complex software systems. In *1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, Leuven, Belgium, April 2006.

[57] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. PortLand: A scalable fault-tolerant layer 2 data center network fabric. In *ACM SIGCOMM (SIGCOMM)*, arcelona, Spain, August 2009.

[58] Arjun Narayan and Andreas Haeberlen. DJoin: Differentially private join queries over distributed databases. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, October 2012.

[59] Arun Natarajan, Peng Ning, Yao Liu, Sushil Jajodia, and Steve E. Hutchinson. NSDMiner: Automated discovery of network service dependencies. In *31st IEEE INFOCOM (INFOCOM)*, Orlando, FL, March 2012.

[60] Suman Nath, Haifeng Yu, Phillip B. Gibbons, and Srinivasan Seshan. Subtleties in tolerating correlated failures in wide-area storage systems. In *3rd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006.

[61] C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *8th ACM Conference on Computer and Communications Security (CCS)*, November 2001.

[62] Pascal Paillier. Public-Key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology (EUROCRYPT)*, May 1999.

[63] Stephen C. Pohlig and Martin E. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance (Corresp.). *IEEE Transactions on Information Theory*, 24(1):106–110, 1978.

[64] Rahul Potharaju and Navendu Jain. When the network crumbles: An empirical study of cloud network failures and their impact on services. In *4th ACM Symposium on Cloud Computing (SoCC)*, Santa Clara, CA, October 2013.

[65] The Associated Press. Fire in baltimore snarls Internet traffic, too, July 2001. http://www.nytimes.com/2001/07/20/technology/20BALT.html.

[66] Chittoor V. Ramamoorthy, Gary S. Ho, and Yih-Wu Han. Fault tree analysis of computer systems. In *AFIPS National Computer Conference*, 1977.

[67] Nuno Santos, Rodrigo Rodrigues, Krishna P Gummadi, and Stefan Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *21st USECIX Security Symposium (USENIX Security)*, Bellevue, WA, August 2012.

[68] Mehul A. Shah, Mary Baker, Jeffrey C. Mogul, and Ram Swaminathan. Auditing to keep online storage services honest. In *11th Workshop on Hot Topics in Operating Systems (HotOS)*, San Diego, CA, May 2007.

[69] Mehul A. Shah, Ram Swaminathan, and Mary Baker. Privacy-preserving audit and extraction of digital contents. *IACR Cryptology ePrint Archive*, 2008:186, 2008.

[70] Adi Shamir, Ron Rivest, and Leonard Adleman. Mental poker. In *MIT Report, LCS/TM-125*, February 1979.

[71] T. Sørensen. *A Method of Establishing Groups of Equal Amplitude in Plant Sociology Based on Similarity of Species Content and Its Application to Analyses of the Vegetation on Danish Commons*. I kommission hos E. Munksgaard, 1948.

[72] Jaideep Vaidya and Chris Clifton. Secure set intersection cardinality with application to association rule mining. *Journal of Computer Security*, 13(4):593–622, 2005.

[73] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8(3):410–421, 1979.

[74] William E. Vesely, Francine F. Goldberg, Norman H. Roberts, and David F. Haasl. *Fault Tree Handbook*. U.S. Nuclear Regulatory Commission, January 1981.

[75] Kevin Walsh and Emin Gün Sirer. Experience with an object reputation system for Peer-to-Peer filesharing. In *3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006.

[76] Cong Wang, Sherman S. M. Chow, Qian Wang, Kui Ren, and Wenjing Lou. Privacy-preserving public auditing for secure cloud storage. *IEEE Transactions on Computers*, 62(2):362–375, 2013.

[77] Cong Wang, Kui Ren, Wenjing Lou, and Jin Li. Toward publicly auditable secure cloud data storage services. *IEEE Network*, 24(4):19–24, 2010.

[78] Cong Wang, Qian Wang, Kui Ren, and Wenjing Lou. Privacy-preserving public auditing for data storage security in cloud computing. In *29th IEEE INFOCOM (INFOCOM)*, San Deigo, CA, March 2010.

[79] Qian Wang, Cong Wang, Kui Ren, Wenjing Lou, and Jin Li. Enabling public auditability and data dynamics for storage security in cloud computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(5):847–859, 2011.

[80] Wei Wei and Bart Selman. A new approach to model counting. In *8th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, St. Andrews, Scotland, June 2005.

[81] Xin Wu, Daniel Turner, Chao-Chih Chen, David A. Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. NetPilot: Automating datacenter network failure mitigation. In *ACM SIGCOMM (SIGCOMM)*, Helsinki, Finland, August 2012.

[82] Hongda Xiao, Bryan Ford, and Joan Feigenbaum. Structural cloud audits that protect private information. In *ACM Cloud Computing Security Workshop (CCSW)*, Berlin, Germany, November 2013.

[83] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009.

[84] Kan Yang and Xiaohua Jia. Data storage auditing service in cloud computing: Challenges, methods and opportunities. *World Wide Web*, 15(4):409–428, 2012.

[85] Andrew Chi-Chih Yao. Protocols for secure computations (Extended abstract). In *23rd Annual Symposium on Foundations of Computer Science (FOCS)*, Chicago, IL, November 1982.

[86] Sebastian Zander, Lachlan L. H. Andrew, and Grenville Armitage. Scalable private set intersection cardinality for capture-recapture with multiple private

datasets. In *Centre for Advanced Internet Architectures, Technical Report 130930A*, 2013.

[87] Ennan Zhai, Ruichuan Chen, David Isaac Wolinsky, and Bryan Ford. An untold story of redundant clouds: Making your service deployment truly reliable. In *9th Workshop on Hot Topics in Dependable Systems (HotDep)*, Nemacolin Woodlands Resort, PA, November 2013.

[88] Ennan Zhai, Ruichuan Chen, David Isaac Wolinsky, and Bryan Ford. Heading off correlated failures through Independence-as-a-service. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, CO, October 2014.

[89] Ennan Zhai, David Isaac Wolinsky, Hongda Xiao, Hongqiang Liu, Xueyuan Su, and Bryan Ford. Auditing the Structural Reliability of the Clouds. Technical Report YALEU/DCS/TR-1479, Department of Computer Science, Yale University, 2013. Available at http://www.cs.yale.edu/homes/zhai-ennan/sra.pdf.

[90] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, CO, October 2014.

[91] Ming Zhong, Kai Shen, and Joel I. Seiferas. Replication degree customization for high availability. In *ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, Glasgow, Scotland, March 2008.