Heading Off Correlated Failures through Independence-as-a-Service

Ennan Zhai[†], Ruichuan Chen[§], David Isaac Wolinsky[†], Bryan Ford[†]
[†]*Yale University* [§]*Bell Labs / Alcatel-Lucent*

Abstract

Today's systems pervasively rely on redundancy to ensure reliability. In complex multi-layered hardware/software stacks, however – especially in the clouds where many independent businesses deploy interacting services on common infrastructure – seemingly independent systems may share deep, hidden dependencies, undermining redundancy efforts and introducing unanticipated correlated failures. Complementing existing post-failure forensics, we propose *Independence-as-a-Service* (or INDaaS), an architecture to audit the independence of redundant systems proactively, thus avoiding correlated failures. INDaaS first utilizes pluggable dependency acquisition modules to collect the structural dependency information (including network, hardware, and software dependencies) from a variety of sources. With this information, INDaaS then quantifies the independence of systems of interest using pluggable auditing modules, offering various performance, precision, and data secrecy tradeoffs. While the most general and efficient auditing modules assume the auditor is able to obtain all required information, INDaaS can employ private set intersection cardinality protocols to quantify the independence even across businesses unwilling to share their full structural information with anyone. We evaluate the practicality of INDaaS with three case studies via auditing realistic network, hardware, and software dependency structures.

1 Introduction

Cloud services normally require high reliability, and pervasively rely on redundancy techniques to ensure this reliability [7, 10, 12, 29]. Amazon S3, for example, replicates each data object across multiple racks in an S3 region [3]. iCloud rents infrastructures from multiple cloud providers – both Amazon EC2 and Microsoft Azure – for redundancy [28]. Seemingly independent infrastructure components, however, may share deep, hidden dependencies. Failures in these shared dependencies may lead to unexpected *correlated failures*, undermining redundancy efforts [19, 27, 34, 44, 47, 74, 75].

In redundant systems, a *risk group* [35] or RG is a set of components whose simultaneous failures could cause a 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 3-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 failure 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. 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 that undermined the EC2's redundancy efforts.

Discovering unexpected common dependencies is extremely challenging [20,22]. Many diagnostic and forensic approaches attempt to localize or tolerate such failures after they occur [5, 12, 15, 24–27, 31, 37, 43]. These retroactive approaches, however, still require human intervention, leading to prolonged failure recovery time [68]. 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 [20].

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 [19]. Application-level cloud services such as iCloud [28] often redundantly rely on *multiple* cloud providers, *e.g.*, Amazon EC2 and Microsoft Azure. However, a storm in Dublin recently took down a local power source and its backup generator, disabling *both* the Amazon and Microsoft clouds in that region for hours [16]. 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 [19, 69, 74].

We propose *Independence-as-a-Service* or INDaaS, a novel architecture that aims to address the above problems proactively. Rather than localizing and tolerating failures after an outage, INDaaS collects and audits structural dependency data to evaluate the independence of redundant systems before failures occur. In particular, INDaaS consists of a pluggable set of *dependency acquisition modules* that collect dependency data, and an auditing agent that employs a similarly pluggable set of *auditing modules* to quantify the independence of redundant systems and identify 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 [52, 60], 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 [72], which offers the best generality in principle but performs adequately only on small dependency datasets [69]. We therefore focus here on the second approach, based on private set intersection cardinality [38, 58]. 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 [23], emergent risks due to overwhelming complexity [44], and proprietary information barriers that naturally arise in the cloud ecosystem [19].

In summary, this paper's contributions are: 1) the first 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.

2 Architecture Overview

We now present a high-level overview of the INDaaS architecture, deferring details to subsequent sections. Figure 1 illustrates the basic INDaaS workflow, which involves three main roles or types of entities: auditing client, dependency data source, and auditing agent.

The *auditing client*, *i.e.*, Alice in Figure 1, requests an audit of the independence of two or more cloud systems, which may either be operated by Alice herself or rented from other cloud providers, and which she believes to be independent so as to offer redundancy. For example, Alice 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 [28]. Alice might also request periodic audits on a deployed configuration to identify correlated failure risks that configuration changes or evolution might introduce.

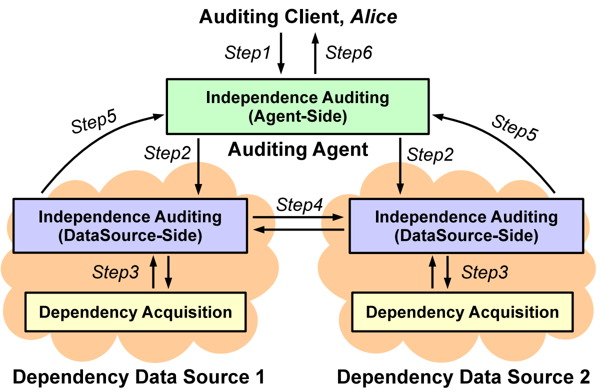


Figure 1: An example INDaaS auditing process, where an auditing client Alice wishes to audit the independence of a two-way redundancy deployment.

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 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 [23]).

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 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.

We briefly summarize the independence auditing process as illustrated in Figure 1:

Step 1: The auditing client, Alice, specifies to the auditing agent what services she wishes to audit and in what way. This specification 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.

Table 1: 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"/>

Step 2: The auditing agent issues a request to each data source Alice specified.

Step 3: Each specified data source uses one or more dependency acquisition modules to collect the dependency data for future independence auditing (see §3).

Step 4: In the private independence auditing (or PIA) case, the data sources collaborate to obtain the auditing results without revealing the proprietary dependency data to each other (see §4.2).

Step 5: Each data source returns to the auditing agent either the full dependency data for structural independence auditing (see §4.1), 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.

3 Dependency Acquisition

Acquiring accurate structural dependency data within heterogeneous cloud systems is non-trivial, and realistic solutions would need to be adapted to different cloud environments. As many dependency acquisition tools have been deployed in today's clouds for various purposes (*e.g.*, system diagnosis) [2, 5, 6, 14, 15, 18, 31, 36, 37, 39], we expect such tools can be adapted and reused to collect the dependency data required by INDaaS.

Towards this end, INDaaS leverages pluggable dependency acquisition modules (DAM), and maintains a uniform representation of different types of dependency data. 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 1. Finally, the DAM stores the adapted dependency data in a database, DepDB, for further processing.

Table 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 [22, 68]: incorrect network

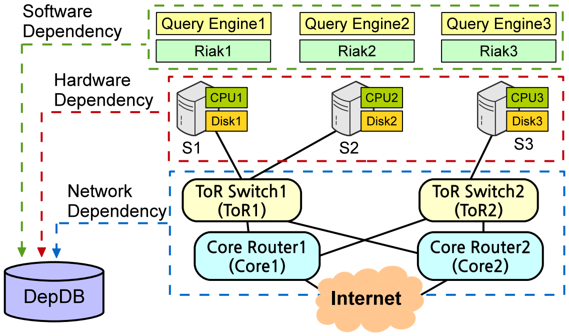


Figure 2: A sample distributed storage system.

```
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 3: A sample of the collected dependency data.

configurations, faulty hardware components, and buggy or insecure software packages.

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

Dependency acquisition examples. 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 [31, 46] to collect network dependencies, HardwareLister [61] to collect hardware dependencies, and apt-rdepends [17] 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 [31,46]. HardwareLister (lshw) extracts a target machine's detailed hardware configuration including CPUs, disks and drivers [61]. The apt-rdepends tool extracts the software package and library dependencies for popular Linux software distributions [17].

Figure 2 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 [8] (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 3.

4 Independence Auditing

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

As described in §2, INDaaS supports two scenarios. We first present a *structural independence auditing* protocol in §4.1, which assumes data sources are willing to provide the auditing agent with the full dependency data, *e.g.*, for auditing a common cloud provider. We later present a *private independence auditing* protocol in §4.2 to support analysis across multiple cloud providers unwilling to reveal the full dependency data to anyone.

4.1 Structural Independence Auditing

Upon acquiring full dependency data from the data sources, the auditing agent executes our structural independence auditing (SIA) protocol to generate the dependency graph, determine the risk groups, rank the risk groups, and eventually generate an auditing report.

4.1.1 Generating Dependency Graph

To implement 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 traditional fault tree models [52, 60] to a directed

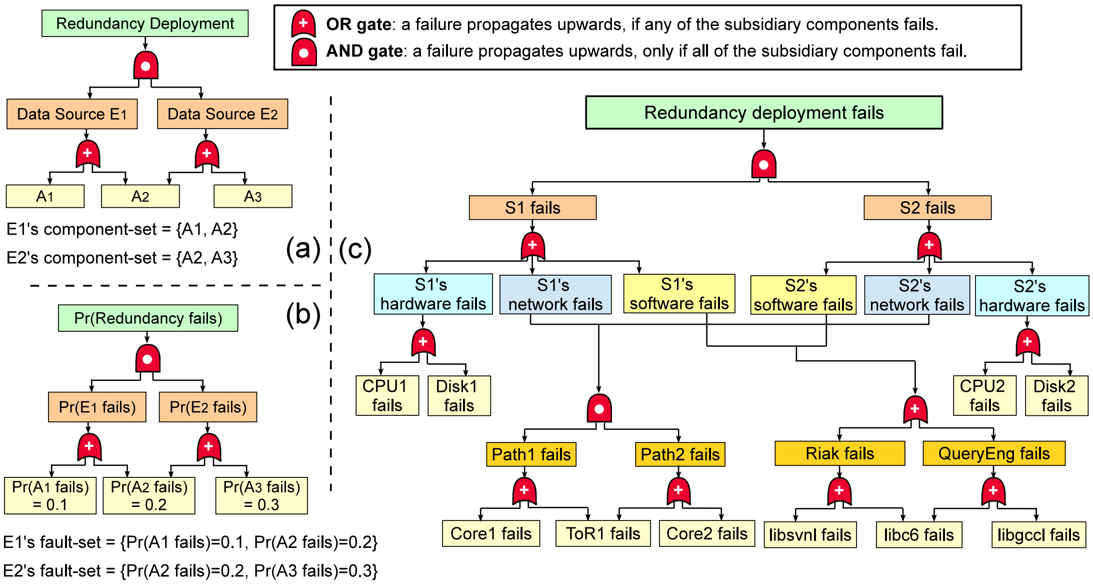


Figure 4: Dependency graphs represented at three different levels of detail: (a) component-set level of detail, (b) fault-set level of detail, and (c) fault graph level of detail.

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 4(a), 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 4(a) 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 4(b), 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 §5.1.

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 4(c). 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 4(c), 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.

Building the 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 §2), 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 2 may be represented by the fault graph in Figure 4(c). 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). Details on dependency graph composition may be found in the associated technical report [75].

4.1.2 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 4(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 4(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 4(c) are {ToR1 fails}, {Core1 fails, Core2 fails}, etc.

Minimal RG algorithm. The first algorithm for determining RGs is adapted from classic fault tree analysis techniques [52, 60]. 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 [59].

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. This failure sampling algorithm is similar in principle to heuristic SAT algorithms such as ApproxCount [67], and these methods may offer ways to improve INDaaS failure sampling.

4.1.3 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 4(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 ranks RGs by their relative importance. Here, for a given RG's failure event (say, C), its relative importance, I_C, is computed using the probability of C, $\Pr(C)$, in comparison to the probability of the top event T, $\Pr(T)$: $I_C = \Pr(C)/\Pr(T)$. 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. In Figure 4(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$\}$.

4.1.4 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.

4.2 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 approach, explored by Xiao et al. [69], is to use secure multi-party computation (SMPC) [72] 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 [21, 38, 58, 73]. 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 [32] using a private set intersection cardinality protocol [58] to quantify the independence of redundancy configurations.

4.2.1 Trust Assumptions

As described in §2, our architecture consists of entities filling three roles: auditing client, cloud providers (*i.e.*, data sources in Figure 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. We discuss some potential solutions to dealing with dishonest parties in §5.2.

4.2.2 Technical Building Blocks

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

Jaccard similarity. Jaccard similarity [32] is a widely-adopted metric for measuring similarity across multiple datasets. Jaccard similarity is defined as $J(S_0,\cdots,S_{k-1}) = |S_0 \cap \cdots \cap S_{k-1}|/|S_0 \cup \cdots \cup S_{k-1}|$ 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 [62]. While there are many other similarity metrics, *e.g.*, the Sørensen-Dice index [57], we choose Jaccard similarity because it is efficient, easy to understand, and extends readily to more than two datasets.

MinHash. Computing the Jaccard similarity incurs a complexity linear with the dataset sizes. In the presence of large datasets, an approximation of the Jaccard similarity based on MinHash is often preferred [11]. The MinHash technique [13] extracts a vector $\{h^{(i)}_{min}(S)\}^m_{i=1}$ of a dataset S through deterministic sampling, where $h^{(1)}(\cdot),\cdots,h^{(m)}(\cdot)$ denote m different hash functions, and $h^{(i)}_{min}(S)$ denotes the item $e \in S$ with the minimum value $h^{(i)}(e)$. Let δ denote the number of datasets satisfying $h^{(i)}_{min}(S_0) = \cdots = h^{(i)}_{min}(S_{k-1})$. 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 [13] proves that the expected error of MinHash-based Jaccard similarity estimation is $O(1/\sqrt{m})$.

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 [58]. 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 [50, 56]. 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 $|\cup_i S_i|$.

4.2.3 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 [19]. 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.

4.2.4 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 §4.2.2, 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.

4.2.5 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 redundancy 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.

5 Limitations and Practical Issues

This section discusses a few INDaaS's limitations and areas for further exploration, as well as some practical issues regarding INDaaS deployment.

5.1 Limitations and Potential Solutions

Failure probability acquisition. Part of INDaaS's utility depends on the acquisition of accurate failure probability information. Without this, we cannot perform some auditing operations, *e.g.*, dependency graph generation at the fault-set level and failure probability based ranking. Collecting failure probabilities automatically is a challenging problem in practice, however. Gill et al. proposed one approach [22]: they estimate failure probability by dividing the number of components of a given type that have ever failed during a time period, by the total component population of that given type. They successfully provide the failure probabilities of various network devices (*e.g.*, aggregation switches and core routers) during a one year period. Regarding the failure probabilities of software dependencies, the Common Vulnerability Scoring System (or CVSS) [48] can be used to provide vulnerability-related failure probabilities for many software libraries and packages.

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 [5,40,47,51]. 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 [70]. Joukov et al. [33] 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. [41, 42], can greatly reduce the workload of the network dependency acquisition.

5.2 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.

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.

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 [53] 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.

6 Implementation and Evaluation

This section first describes our INDaaS prototype implementation (§6.1), then evaluates its practicality (§6.2) and performance (§6.3).

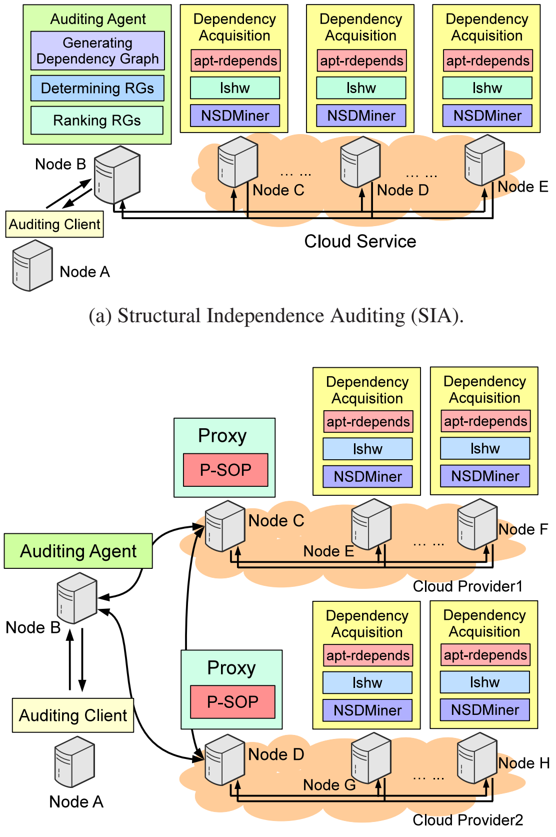
6.1 Implementation and Deployment

We have built an INDaaS prototype system written in a mix of Python and Java. For clarity, this section focuses first on our implementation of SIA, followed by PIA.

6.1.1 Structural Independence Auditing

Figure 5a shows the key components of an INDaaS prototype in the SIA scenario.

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 5a. The auditing client communicates with the auditing agent to send the specification and receive the auditing report.



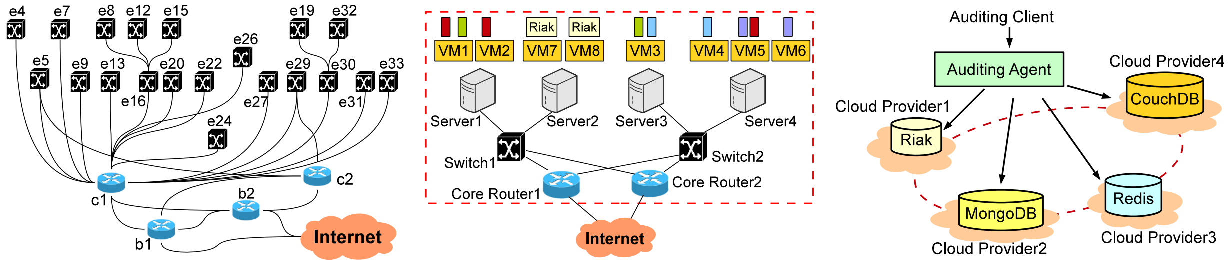
(a) Structural Independence Auditing (SIA).

(b) Private Independence Auditing (PIA).

Figure 5: INDaaS implementation and deployment.

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 5a. Our current dependency acquisition implementation includes three open-source tools: NSDMiner [46], lshw [61], and apt-rdepends [17], which are used to collect network, hardware, and software dependencies, respectively. 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 [49] library, is deployed on another machine, *e.g.*, Node B in Figure 5a. 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.



(a) Common network dependency.

(b) Common hardware dependency.

(c) Common software dependency.

Figure 6: Practicality evaluation through three case studies: (a) common network dependency, (b) common hardware dependency, and (c) common software dependency.

6.1.2 Private Independence Auditing

Figure 5b 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.

Dependency acquisition and proxy. For each cloud provider, there are three dependency acquisition modules deployed on each of its worker machines, as in SIA. 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 [56].

6.2 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.

6.2.1 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 [9] to model Alice's data center network. As shown in Figure 6a, this data center has many Top-of-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.

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 correlated 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.

6.2.2 Common Hardware Dependency

As shown in Figure 6b, we have built a simple IaaS cloud in the lab with four servers and four switches. We used OpenStack to support the automatic virtual machine (VM) management, and deployed various services on VMs for different uses. In particular, we deployed an S3-like Riak [8] cloud storage service. For redundancy, Riak was run on two VMs (VM7 and VM8).

Before releasing the Riak storage service for public use, we ran SIA to check whether there would be any unexpected RGs. 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

Table 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

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.

6.2.3 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 [23].

Here, we chose four popular key-value storage systems, *i.e.*, Riak, MongoDB, Redis, and CouchDB. As shown in Figure 6c, 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 2 shows the ranking lists of various two- and three-way redundancy deployments.

Table 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

6.3 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.

6.3.1 SIA: Efficiency v.s. Accuracy

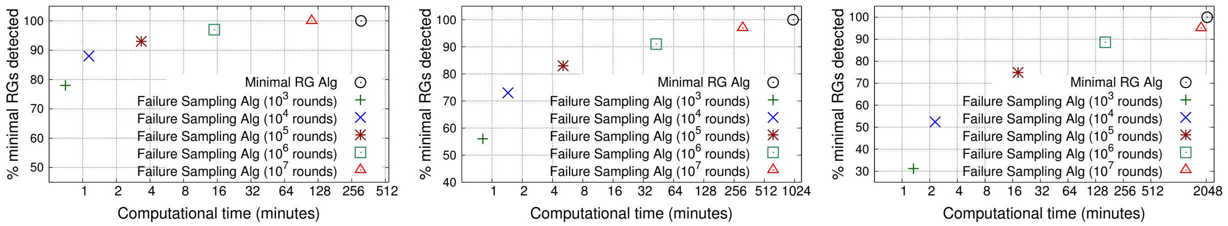
We first explore the efficiency/accuracy trade-off between SIA's two algorithms for analyzing a dependency graph: the minimal RG algorithm and the failure sampling algorithm (see §4.1.2). We generate three topologies from a small-scale cloud deployment to a large-scale deployment, based on the three-stage fat tree model [45]. These topologies include the typical components within a commercial data center: servers, Top-of-Rack (ToR) switches, aggregation switches, and core routers. Table 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 7 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.

6.3.2 PIA: System Overheads

To better understand the performance of PIA, we implemented a comparable private independence auditing system based on another private set intersection cardinality protocol, Kissner and Song (KS) [38], and then compared this system with our PIA system.

For a private independence auditing system, the cryptographic operations tend to be the major computational bottleneck. Thus, we evaluate PIA by comparing PIA's P-SOP protocol with the comparable system's 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.

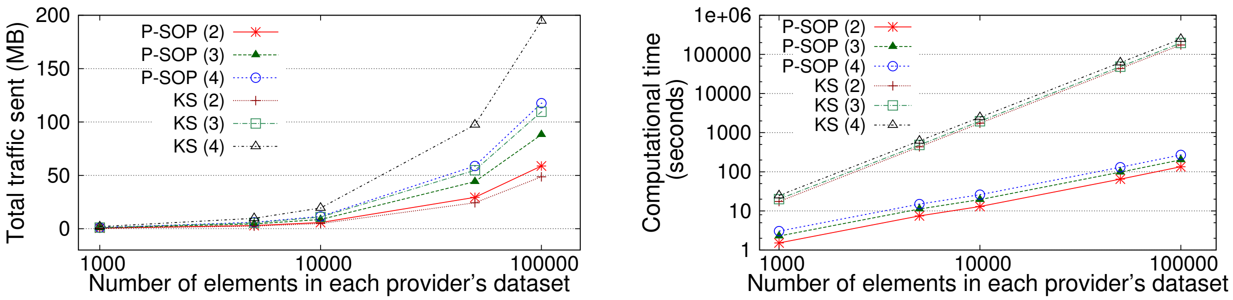


(a) Topology A: 1,344 devices.

(b) Topology B: 4,176 devices.

(c) Topology C: 30,528 devices.

Figure 7: Performance evaluation of the minimal RG algorithm and the failure sampling algorithm in SIA.



(a) Bandwidth overhead.

(b) Computational overhead.

Figure 8: System 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.

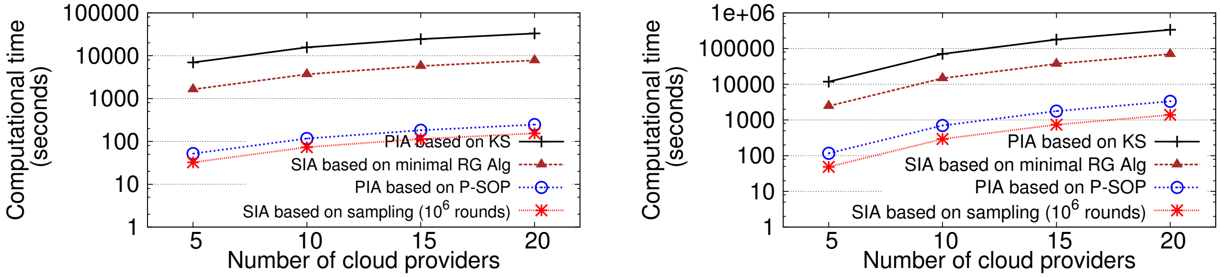
In the evaluation, there are k cloud providers with n elements in each provider's local dataset. We set k to 2, 3 and 4, and vary n between $1,000$ and $100,000$ to cover a wide range of real-world settings. We measure and compare P-SOP with KS in terms of their bandwidth and computational overheads at each such cloud provider. Figure 8a and 8b 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 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.

6.3.3 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 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.

Figure 9a and 9b show the computational overheads of these independence calculations for all potential two- and three-way redundancies, respectively. As expected, 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.



(a) Two-way redundancy.

(b) Three-way redundancy.

Figure 9: Performance comparison between SIA and PIA. Each cloud provider maintains a 10,000-element dataset.

7 Related Work

Providing audits for clouds is a well-known approach to increase reliability [54]. 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.

Privacy-preserving auditing systems. Following the auditing concept proposed by Shah et al. [54], many privacy-preserving auditing systems have been proposed extending this approach [55, 63–66, 71].

Similar to PIA, iRec [74] and Xiao et al. [69] 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 [21] and the secure multi-party computation protocol [72] to perform the dependency analysis in a privacy-preserving fashion, respectively. These initial efforts did not scale to handle realistically large cloud datasets, however,

Diagnosis & accountability systems. Diagnosis systems, unlike auditing, attempt to discover failures after they occur. For example, many inference-based diagnosis systems [5, 15, 31, 37] have been proposed to obtain the network dependencies of a cloud service when a failure occurs. Unlike existing diagnosis systems, NetPilot [68] 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 [24] proposed using third-party verifiable evidence to determine whether the cloud customer or the cloud provider should be held liability when a failure occurs.

Private set operations. Secure multi-party computation (SMPC) [72] is a general approach to supporting computation on private data including set operations. However, current circuit-based SMPC protocols are too expensive and scale poorly to large computations. Arawal et al. [1] proposed a private set intersection cardinality protocol based on commutative encryption. This protocol was limited to two-party cases, however. Vaidya and Clifton [58] extended this protocol to support more than two parties, and optimized its efficiency.

The first private set intersection cardinality protocol based on homomorphic encryption was proposed by Freedman et al. [21], which could privately compute the number of elements common to two datasets. Hohenberger et al. proposed enhancements to this protocol protocol [30]. Later, Kissner and Song proposed multi-party private set operations based upon homomorphic encryption and polynomial generation [38].

8 Conclusion

This paper has presented INDaaS, an architecture to audit the independence of future or existing redundant service deployments in the cloud. While only a start, our proof-of-concept prototype and experiments suggest that INDaaS could be both practical and effective in detecting and heading off correlated failure risks before they occur.

Acknowledgments

We thank our shepherd, Timothy Roscoe, and the anonymous reviewers for their insightful comments. We also thank Gustavo Alonso, Hongqiang Liu, Jeff Mogul, Ruzica Piskac, Xueyuan Su, Hongda Xiao, and Sebastian Zander for their valuable feedback on earlier drafts of this paper. This research was sponsored by the NSF under grants CNS-1017206 and CNS-1149936.

References

[1] Rakesh Agrawal, Alexandre V. Evfimievski, and Ramakrishnan Srikant. Information sharing across private databases. In *ACM SIGMOD*, June 2003.

[2] 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)*, October 2003.

[3] Amazon S3's redundant storage. http://aws.amazon.com/s3/, accessed on Sep 9, 2014.

[4] Amazon Web Services Team. Summary of the October 22, 2012 AWS service event in the US-East region. https://aws.amazon.com/message/680342/, accessed on Sep 9, 2014.

[5] 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*, August 2007.

[6] 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)*, December 2004.

[7] 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)*, June 2012.

[8] Basho Technologies. Riak. http://basho.com/riak/, accessed on Sep 9, 2014.

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

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

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

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

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

[14] 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)*, March 2004.

[15] 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)*, December 2008.

[16] 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.

[17] Debian. Package apt-rdepends: Recursively lists package dependencies. http://packages.debian.org/sid/apt-rdepends, accessed on Sep 9, 2014.

[18] 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)*, December 2004.

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

[20] 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)*, October 2010.

[21] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *EUROCRYPT*, May 2004.

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

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

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

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

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

[27] 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)*, May 2005.

[28] Devindra Hardawar. Apple's iCloud runs on Microsoft's Azure and Amazon's cloud. *VentureBeat News*, September 2011. http://venturebeat.com/2011/09/03/icloud-azure-amazon/, accessed on Sep 9, 2014.

[29] 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)*, October 2013.

[30] Susan Hohenberger and Stephen A. Weis. Honest-verifier private disjointness testing without random oracles. In *6th Workshop on Privacy Enhancing Technologies*, 2006.

[31] 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)*, December 2012.

[32] 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.

[33] Nikolai Joukov, Vasily Tarasov, Joel Ossher, Birgit Pfitzmann, Sergej Chicherin, Marco Pistoiz, and Takaaki Tateishi. Static discovery and remediation of code-embedded resource dependencies. In *Integrated Network Management*, pages 233–240, 2011.

[34] Flavio Paiva Junqueira, Ranjita Bhagwan, Alejandro Hevia, Keith Marzullo, and Geoffrey M. Voelker. Surviving Internet catastrophes. In *USENIX Annual Technical Conference*, pages 45–60, April 2005.

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

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

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

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

[39] 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)*, May 2005.

[40] Sarah Kuranda. The 10 Biggest Cloud Outages of 2013. *CRN*, January 2014. http://www.crn.com/slide-shows/cloud/240165024/the-10-biggest-cloud-outages-of-2013.htm, accessed on Sep 9, 2014.

[41] 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)*, November 2014.

[42] 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)*, December 2013.

[43] 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)*, April 2013.

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

[45] 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*, August 2009.

[46] Arun Natarajan, Peng Ning, Yao Liu, Sushil Jajodia, and Steve E. Hutchinson. NSDMiner: Automated discovery of network service dependencies. In *IEEE INFOCOM*, December 2012.

[47] 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)*, May 2006.

[48] National Institute of Standards and Technology. Common Vulnerability Scoring System (CVSS). http://nvd.nist.gov/cvss.cfm, accessed on Sep 9, 2014.

[49] NetworkX. http://networkx.github.com/, accessed on Sep 9, 2014.

[50] 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.

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

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

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

[54] 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)*, May 2007.

[55] Mehul A. Shah, Ram Swaminathan, and Mary Baker. Privacy-preserving audit and extraction of digital contents. Technical Report HPL-2008-32R1, HP Laboratories, April 2008.

[56] Adi Shamir, Ron Rivest, and Leonard Adleman. Mental poker. Technical Report LCS/TM-125, Massachusetts Institute of Technology, February 1979.

[57] 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.

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

[59] Leslie G. Valiant. The Complexity of Enumeration and Reliability Problems. *SIAM Journal of Computing*, 8(3):410–421, 1979.

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

[61] Lyonel Vincent. Hardware Lister (lshw). http://ezix.org/project/wiki/HardwareLiSter, accessed on Sep 9, 2014.

[62] 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)*, May 2006.

[63] 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.

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

[65] Cong Wang, Qian Wang, Kui Ren, and Wenjing Lou. Privacy-preserving public auditing for data storage security in cloud computing. In *IEEE INFOCOM*, March 2010.

[66] 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.

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

[68] 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*, August 2012.

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

[70] 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)*, October 2009.

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

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

[73] 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.

[74] 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)*, November 2013.

[75] 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, 2014. Available at `http://cpsc.yale.edu/sites/default/files/files/tr1479.pdf`, accessed on Sep 9, 2014.